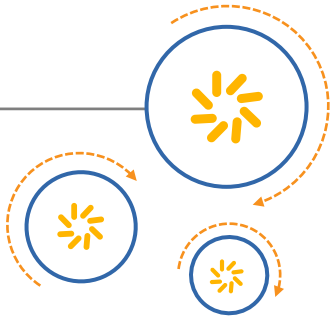
Qualcomm Technologies, Inc.



Hexagon V60/V61 Programmer's Reference Manual

80-N2040-33 Rev. D

March 9, 2016

Questions or comments: support.cdmatech.com

Qualcomm Hexagon is a product of Qualcomm Technologies, Inc. Other Qualcomm products referenced herein are products of Qualcomm Technologies, Inc. or its subsidiaries.

Qualcomm and Hexagon are trademarks of Qualcomm Incorporated, registered in the United States and other countries. Other product and brand names may be trademarks or registered trademarks of their respective owners.

This technical data may be subject to U.S. and international export, re-export, or transfer ("export") laws. Diversion contrary to U.S. and international law is strictly prohibited.

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

© 2015, 2016 Qualcomm Technologies, Inc. All rights reserved.

Contents

1 Introduction 17

1.1 Overview 17
1.2 Features 18
1.3 Functional units 19
1.3.1 Memory 20
1.3.2 Registers 20
1.3.3 Sequencer 20
1.3.4 Execution units 20
1.3.5 Load/store units 20
1.4 Instruction set 21
1.4.1 Addressing modes 21
1.4.2 Scalar operations 22
1.4.3 Vector operations 22
1.4.4 Floating-point operations 23
1.4.5 Program flow 24
1.4.6 Instruction packets 24
1.4.7 Dot-new instructions 25
1.4.8 Compound instructions 25
1.4.9 Duplex instructions 25
1.4.10 Instruction classes 25
1.4.11 Instruction intrinsics 26
1.5 Processor versions 27
1.6 Using the manual 28
1.7 Notation 28
1.7.1 Instruction syntax 28
1.7.2 Register operands 29
1.7.3 Numeric operands 31
1.8 Terminology 32
1.9 Feedback 32

2 Registers 33
2.1 Overview 33
2.2 General registers 34
2.3 Control registers 36
2.3.1 Program counter 38
2.3.2 Loop registers 39
2.3.3 User status register 39
2.3.4 Modifier registers 42
2.3.5 Predicate registers 43
2.3.6 Circular start registers 44
2.3.7 User general pointer register 44
2.3.8 Global pointer 44
2.3.9 Cycle count registers 45
2.3.10 Frame limit register 45
2.3.11 Frame key register 46
2.3.12 Packet count registers 46
2.3.13 Qtimer registers 47

3 Instructions 48
3.1 Overview 48
3.2 Instruction syntax 49
3.3 Instruction classes 50
3.4 Instruction packets 51
3.4.1 Packet execution semantics 52
3.4.2 Sequencing semantics 52
3.4.3 Resource constraints 53
3.4.4 Grouping constraints 54
3.4.5 Dependency constraints 55
3.4.6 Ordering constraints 55
3.4.7 Alignment constraints 56
3.5 Instruction intrinsics 56
3.6 Compound instructions 57
3.7 Duplex instructions 57

4 Data Processing 58
4.1 Overview 58
4.2 Data types 59
4.2.1 Fixed-point data 59
4.2.2 Floating-point data 59
4.2.3 Complex data 59
4.2.4 Vector data 59

4.3 Instruction options 61
4.3.1 Fractional scaling 61
4.3.2 Saturation 61
4.3.3 Arithmetic rounding 61
4.3.4 Convergent rounding 62
4.3.5 Scaling for divide and square-root 62
4.4 XTYPE operations 63
4.4.1 ALU 63
4.4.2 Bit manipulation 64
4.4.3 Complex 64
4.4.4 Floating point 65
4.4.5 Multiply 66
4.4.6 Permute 68
4.4.7 Predicate 68
4.4.8 Shift 69
4.5 ALU32 operations 70
4.6 Vector operations 71
4.7 CR operations 73
4.8 Compound operations 73
4.9 Special operations 73
4.9.1 H.264 CABAC processing 74
4.9.1.1 CABAC implementation 75
4.9.1.2 Code example 76
4.9.2 IP internet checksum 77
4.9.2.1 Code example 78
4.9.3 Software-defined radio 79
4.9.3.1 Rake despreading 79
4.9.3.2 Polynomial operations 80

5 Memory 82

5.1 Overview 82
5.2 Memory model 83
5.2.1 Address space 83
5.2.2 Byte order 83
5.2.3 Alignment 83
5.3 Memory loads 84
5.4 Memory stores 85
5.5 Dual stores 85
5.6 New-value stores 86
5.7 Mem-ops 86

5.8 Addressing modes 87
5.8.1 Absolute 87
5.8.2 Absolute-set 87
5.8.3 Absolute with register offset 88
5.8.4 Global pointer relative 88
5.8.5 Indirect 89
5.8.6 Indirect with offset 89
5.8.7 Indirect with register offset 90
5.8.8 Indirect with auto-increment immediate 90
5.8.9 Indirect with auto-increment register 90
5.8.10 Circular with auto-increment immediate 91
5.8.11 Circular with auto-increment register 93
5.8.12 Bit-reversed with auto-increment register 94
5.9 Conditional load/stores 95
5.10 Cache memory 96
5.10.1 Uncached memory 97
5.10.2 Tightly coupled memory 97
5.10.3 Cache maintenance operations 98
5.10.4 L2 cache operations 99
5.10.5 Cache line zero 99
5.10.6 Cache prefetch 100
5.11 Memory ordering 103
5.12 Atomic operations 104

6 Conditional Execution 106

6.1 Overview 106
6.2 Scalar predicates 107
6.2.1 Generating scalar predicates 107
6.2.2 Consuming scalar predicates 109
6.2.3 Auto-AND predicates 110
6.2.4 Dot-new predicates 111
6.2.5 Dependency constraints 112
6.3 Vector predicates 112
6.3.1 Vector compare 112
6.3.2 Vector mux instruction 114
6.3.3 Using vector conditionals 115
6.4 Predicate operations 116

7 Program Flow....117
7.1 Overview 117
7.2 Conditional instructions.... 118
7.3 Hardware loops.... 118
7.3.1 Loop setup.... 120
7.3.2 Loop end 121
7.3.3 Loop execution.... 122
7.3.4 Pipelined hardware loops 123
7.3.5 Loop restrictions 126
7.4 Software branches 126
7.4.1 Jumps 127
7.4.2 Calls 127
7.4.3 Returns 128
7.4.4 Extended branches 129
7.4.5 Branches to and from packets 129
7.5 Speculative jumps.... 130
7.6 Compare jumps.... 131
7.6.1 New-value compare jumps 132
7.7 Register transfer jumps.... 133
7.8 Dual jumps.... 134
7.9 Hint indirect jump target 135
7.10 Pauses 136
7.11 Exceptions 136

8 Software Stack 140
8.1 Overview 140
8.2 Stack structure 141
8.3 Stack frames 142
8.4 Stack protection 142
8.4.1 Stack bounds checking.... 142
8.4.2 Stack smashing protection 143
8.5 Stack registers.... 143
8.6 Stack instructions.... 144

9 PMU Events 145
9.1 Overview 145
9.2 V60 processor event symbols.... 146
9.3 V61 processor event symbols.... 154

10 Instruction Encoding 162
10.1 Overview 162
10.2 Instructions 163
10.3 Sub-instructions 164
10.4 Duplexes 167
10.5 Instruction classes 169
10.6 Instruction packets 170
10.7 Loop packets 171
10.8 Immediate values 172
10.9 Scaled immediates 172
10.10 Constant extenders 173
10.11 New-value operands 176
10.12 Instruction mapping 177

11 Instruction Set 178
11.1 Overview 178
11.2 Instruction categories 179
11.3 ALU32 182
11.3.1 ALU32/ALU 182
Add 183
Logical operations 185
Negate 187
Nop 188
Subtract 189
Sign extend 191
Transfer immediate 192
Transfer register 194
Vector add halfwords 195
Vector average halfwords 196
Vector subtract halfwords 197
Zero extend 199
11.3.2 ALU32/PERM 199
Combine words into doubleword 200
Mux 202
Shift word by 16 204
Pack high and low halfwords 206
11.3.3 ALU32/PRED 207
Conditional add 208
Conditional shift halfword 210
Conditional combine 212
Conditional logical operations 213
Conditional subtract 215

Conditional sign extend ... 216
Conditional transfer ... 218
Conditional zero extend ... 219
Compare ... 221
Compare to general register ... 223
11.4 CR ... 224
End loop instructions ... 225
Corner detection acceleration ... 226
Logical reductions on predicates ... 227
Looping instructions ... 228
Add to PC ... 230
Pipelined loop instructions ... 231
Logical operations on predicates ... 233
User control register transfer ... 235
11.5 JR ... 236
Call subroutine from register ... 237
Hint an indirect jump address ... 238
Jump to address from register ... 239
11.6 J ... 240
Call subroutine ... 241
Compare and jump ... 243
Jump to address ... 248
Jump to address conditioned on new predicate ... 250
Jump to address condition on register value ... 251
Transfer and jump ... 253
11.7 LD ... 253
Load doubleword ... 254
Load doubleword conditionally ... 256
Load byte ... 258
Load byte conditionally ... 260
Load byte into shifted vector ... 262
Load half into shifted vector ... 265
Load halfword ... 268
Load halfword conditionally ... 270
Load unsigned byte ... 272
Load unsigned byte conditionally ... 274
Load unsigned halfword ... 276
Load unsigned halfword conditionally ... 278
Load word ... 280
Load word conditionally ... 282
Deallocate stack frame ... 284
Deallocate frame and return ... 286
Load and unpack bytes to halfwords ... 288

11.8 MEMOP 297
Operation on memory byte 298
Operation on memory halfword 299
Operation on memory word 300
11.9 NV 301
11.9.1 NV/J 301
Jump to address condition on new register value 302
11.9.2 NV/ST 305
Store new-value byte 306
Store new-value byte conditionally 309
Store new-value halfword 312
Store new-value halfword conditionally 315
Store new-value word 318
Store new-value word conditionally 321
11.10 ST 323
Store doubleword 324
Store doubleword conditionally 326
Store byte 328
Store byte conditionally 330
Store halfword 333
Store halfword conditionally 336
Store word 340
Store word conditionally 342
Allocate stack frame 345
11.11 SYSTEM 346
11.11.1 SYSTEM/USER 346
Load locked 347
Store conditional 348
Zero a cache line 350
Memory barrier 351
Breakpoint 352
Data cache prefetch 353
Data cache maintenance user operations 354
Instruction cache maintenance user operations 355
Instruction synchronization 356
L2 cache prefetch 357
Pause 360
Memory thread synchronization 361
Send value to ETM trace 362
Trap 363

11.12 XTYPE 363
11.12.1 XTYPE/ALU 364
Absolute value doubleword 365
Absolute value word 366
Add and accumulate 367
Add doublewords 369
Add halfword 371
Add or subtract doublewords with carry 374
Logical doublewords 375
Logical-logical doublewords 377
Logical-logical words 378
Maximum words 380
Maximum doublewords 381
Minimum words 382
Minimum doublewords 383
Modulo wrap 384
Negate 385
Round 386
Subtract doublewords 388
Subtract and accumulate words 389
Subtract halfword 390
Sign extend word to doubleword 393
Vector absolute value halfwords 394
Vector absolute value words 395
Vector absolute difference bytes 396
Vector absolute difference halfwords 397
Vector absolute difference words 398
Vector add compare and select maximum bytes 399
Vector add compare and select maximum halfwords 400
Vector add halfwords 402
Vector add halfwords with saturate and pack to unsigned bytes 404
Vector reduce add unsigned bytes 405
Vector reduce add halfwords 407
Vector add bytes 409
Vector add words 410
Vector average halfwords 411
Vector average unsigned bytes 413
Vector average words 414
Vector conditional negate 416
Vector maximum bytes 418
Vector maximum halfwords 419
Vector reduce maximum halfwords 420
Vector reduce maximum words 422

Vector maximum words 424
Vector minimum bytes 425
Vector minimum halfwords 427
Vector reduce minimum halfwords 428
Vector reduce minimum words 430
Vector minimum words 432
Vector sum of absolute differences unsigned bytes 433
Vector subtract halfwords 435
Vector subtract bytes 437
Vector subtract words 438
11.12.2 XTYPE/BIT 438
Count leading 439
Count population 441
Count trailing 442
Extract bitfield 443
Insert bitfield 446
Interleave/deinterleave 449
Linear feedback-shift iteration 450
Masked parity 451
Bit reverse 452
Set/clear/toggle bit 453
Split bitfield 455
Table index 457
11.12.3 XTYPE/COMPLEX 459
Complex add/sub halfwords 460
Complex add/sub words 463
Complex multiply 465
Complex multiply real or imaginary 469
Complex multiply with round and pack 471
Complex multiply 32x16 473
Vector complex multiply real or imaginary 475
Vector complex conjugate 478
Vector complex rotate 479
Vector reduce complex multiply real or imaginary 481
Vector reduce complex multiply by scalar 484
Vector reduce complex multiply by scalar with round and pack 488
Vector reduce complex rotate 491
11.12.4 XTYPE/FP 494
Floating point addition 495
Classify floating-point value 496
Compare floating-point value 497
Convert floating-point value to other format 499
Convert integer to floating-point value 500

Convert floating-point value to integer ... 502
Floating point extreme value assistance ... 504
Floating point fused multiply-add ... 505
Floating point fused multiply-add with scaling ... 506
Floating point reciprocal square root approximation ... 507
Floating point fused multiply-add for library routines ... 508
Create floating-point constant ... 510
Floating point maximum ... 511
Floating point minimum ... 512
Floating point multiply ... 513
Floating point reciprocal approximation ... 514
Floating point subtraction ... 515
11.12.5 XTYPE/MPY ... 515
Multiply and use lower result ... 516
Vector multiply word by signed half (32x16) ... 519
Vector multiply word by unsigned half (32x16) ... 524
Multiply signed halfwords ... 529
Multiply unsigned halfwords ... 536
Polynomial multiply words ... 541
Vector reduce multiply word by signed half (32x16) ... 543
Multiply and use upper result ... 545
Multiply and use full result ... 548
Vector dual multiply ... 550
Vector dual multiply with round and pack ... 553
Vector reduce multiply bytes ... 555
Vector dual multiply signed by unsigned bytes ... 557
Vector multiply even halfwords ... 559
Vector multiply halfwords ... 561
Vector multiply halfwords with round and pack ... 563
Vector multiply halfwords, signed by unsigned ... 565
Vector reduce multiply halfwords ... 567
Vector multiply bytes ... 569
Vector polynomial multiply halfwords ... 571
11.12.6 XTYPE/PERM ... 573
CABAC decode bin ... 574
CABAC encode bin ... 575
Saturate ... 576
Swizzle bytes ... 578
Vector align ... 579
Vector round and pack ... 581
Vector saturate and pack ... 583
Vector saturate without pack ... 586
Vector shuffle ... 588

Vector splat bytes ... 590
Vector splat halfwords ... 591
Vector splice ... 592
Vector sign extend ... 593
Vector truncate ... 595
Vector zero extend ... 598
11.12.7 XTYPE/PRED ... 599
Bounds check ... 600
Compare byte ... 601
Compare half ... 603
Compare doublewords ... 605
Compare bit mask ... 606
Mask generate from predicate ... 607
Check for TLB match ... 608
Predicate transfer ... 609
Test bit ... 610
Vector compare halfwords ... 611
Vector compare bytes for any match ... 613
Vector compare bytes ... 614
Vector compare words ... 616
Viterbi pack even and odd predicate bits ... 618
Vector mux ... 619
11.12.8 XTYPE/SHIFT ... 620
Shift by immediate ... 621
Shift by immediate and accumulate ... 623
Shift by immediate and add ... 626
Shift by immediate and logical ... 627
Shift right by immediate with rounding ... 631
Shift left by immediate with saturation ... 633
Shift by register ... 634
Shift by register and accumulate ... 637
Shift by register and logical ... 640
Shift by register with saturation ... 644
Vector shift halfwords by immediate ... 646
Vector arithmetic shift halfwords with round ... 648
Vector arithmetic shift halfwords with saturate and pack ... 650
Vector shift halfwords by register ... 652
Vector shift words by immediate ... 654
Vector shift words by register ... 656
Vector shift words with truncate and pack ... 658

Instruction Index ... 660
Intrinsics Index ... 674

Figures

Figure 1-1 Hexagon V6x processor architecture 19
Figure 1-2 Vector instruction example 23
Figure 1-3 Instruction classes and combinations 26
Figure 1-4 Register field symbols 30
Figure 2-1 General registers 34
Figure 2-2 Control registers 36
Figure 3-1 Packet grouping combinations 53
Figure 4-1 Vector byte operation 60
Figure 4-2 Vector halfword operation 60
Figure 4-3 Vector word operation 60
Figure 4-4 64-bit shift and add/sub/logical 69
Figure 4-5 Vector halfword shift right 72
Figure 5-1 Hexagon processor byte order 83
Figure 5-2 L2FETCH instruction 101
Figure 6-1 Vector byte compare 113
Figure 6-2 Vector halfword compare 113
Figure 6-3 Vector mux instruction 114
Figure 8-1 Stack structure 141
Figure 10-1 Instruction packet encoding 170

Tables

Table 1-1 Register symbols ... 29
Table 1-2 Register bit field symbols ... 30
Table 1-3 Instruction operands ... 31
Table 1-4 Data symbols ... 32
Table 2-1 General register aliases ... 35
Table 2-2 General register pairs ... 35
Table 2-3 Aliased control registers ... 37
Table 2-4 Control register pairs ... 38
Table 2-5 Loop registers ... 39
Table 2-6 User status register ... 40
Table 2-7 Modifier registers (indirect auto-increment addressing) ... 42
Table 2-8 Modifier registers (circular addressing) ... 42
Table 2-9 Modifier registers (bit-reversed addressing) ... 43
Table 2-10 Predicate registers ... 43
Table 2-11 Circular start registers ... 44
Table 2-12 User general pointer register ... 44
Table 2-13 Global pointer register ... 44
Table 2-14 Cycle count registers ... 45
Table 2-15 Frame limit register ... 45
Table 2-16 Frame key register ... 46
Table 2-17 Packet count registers ... 46
Table 2-18 Qtimer registers ... 47
Table 3-1 Instruction symbols ... 49
Table 3-2 Instruction classes ... 50
Table 4-1 Single-precision multiply options ... 67
Table 4-2 Double precision multiply options ... 67
Table 4-3 Control register transfer instructions ... 73
Table 5-1 Memory alignment restrictions ... 84
Table 5-2 Load instructions ... 84
Table 5-3 Store instructions ... 85
Table 5-4 Mem-ops ... 86
Table 5-5 Addressing modes ... 87
Table 5-6 Offset ranges (Global pointer relative) ... 89
Table 5-7 Offset ranges (Indirect with offset) ... 89
Table 5-8 Increment ranges (Indirect with auto-inc immediate) ... 90
Table 5-9 Increment ranges (Circular with auto-inc immediate) ... 92
Table 5-10 Increment ranges (Circular with auto-inc register) ... 93
Table 5-11 Addressing modes (Conditional load/store) ... 95
Table 5-12 Conditional offset ranges (Indirect with offset) ... 96
Table 5-13 Hexagon processor cache size ... 97
Table 5-14 Cache instructions (User-level) ... 98

Table 5-15 Memory ordering instructions ... 103
Table 5-16 Atomic instructions ... 104
Table 6-1 Scalar predicate-generating instructions ... 107
Table 6-2 Vector mux instruction ... 114
Table 6-3 Predicate register instructions ... 116
Table 7-1 Loop instructions ... 120
Table 7-2 Software pipelined loop ... 124
Table 7-3 Software pipelined loop (using spNloop0) ... 125
Table 7-4 Software branch instructions ... 126
Table 7-5 Jump instructions ... 127
Table 7-6 Call instructions ... 127
Table 7-7 Return instructions ... 128
Table 7-8 Speculative jump instructions ... 130
Table 7-9 Compare jump instructions ... 132
Table 7-10 New-value compare jump instructions ... 133
Table 7-11 Register transfer jump instructions ... 134
Table 7-12 Dual jump instructions ... 134
Table 7-13 Jump hint instruction ... 135
Table 7-14 Pause instruction ... 136
Table 7-15 V60/61 exceptions ... 137
Table 8-1 Stack registers ... 143
Table 8-2 Stack instructions ... 144
Table 9-1 V60 processor event symbols ... 146
Table 9-2 V61 processor event symbols ... 154
Table 10-1 Instruction fields ... 163
Table 10-2 Sub-instructions ... 164
Table 10-3 Sub-instruction registers ... 166
Table 10-4 Duplex instruction ... 167
Table 10-5 Duplex ICLASS field ... 167
Table 10-6 Instruction class encoding ... 169
Table 10-7 Loop packet encoding ... 171
Table 10-8 Scaled immediate encoding (indirect offsets) ... 172
Table 10-9 Constant extender encoding ... 173
Table 10-10 Constant extender instructions ... 174
Table 10-11 Instruction mapping ... 177
Table 11-1 Instruction syntax symbols ... 180
Table 11-2 Instruction operand symbols ... 180
Table 11-3 Instruction behavior symbols ... 181

1 Introduction

1.1 Overview

The Qualcomm Hexagon™ processor is a general-purpose digital signal processor designed for high performance and low power across a wide variety of multimedia and modem applications. V6x is the sixth version of the Hexagon processor architecture.

This chapter provides an introduction to the following topics:

- Hexagon processor features
- Processor functional units
- Instruction set
- Processor versions

The chapter concludes with an overview of the manual organization and a description of the terminology and notations used in the manual. It also explains how to contact Qualcomm with any comments or suggestions.

1.2 Features

- **Memory**

  Program code and data are stored in a unified 32-bit address space. The load/store architecture supports a complete set of addressing modes for both compiler code generation and DSP application programming.

- **Registers**

  Thirty two 32-bit general purpose registers can be accessed as single registers or as 64-bit register pairs. The general registers hold all data including scalar, pointer, and packed vector data.

- **Data types**

  Instructions can perform a wide variety of operations on fixed-point or floating-point data. The fixed-point operations support scalar and vector data in a variety of sizes. The floating-point operations support single-precision data.

- **Parallel execution**

  Instructions can be grouped into very long instruction word (VLIW) packets for parallel execution, with each packet containing from one to four instructions. Vector instructions operate on single instruction multiple data (SIMD) vectors.

- **Program flow**

  Nestable zero-overhead hardware loops are supported. Conditional/unconditional jumps and subroutine calls support both PC-relative and register indirect addressing. Two program flow instructions can be grouped into one packet.

- **Instruction pipeline**

  Pipeline hazards are resolved by the hardware: instruction scheduling is not constrained by pipeline restrictions.

- **Code compression**

  Compound instructions merge certain common operation sequences (add-accumulate, shift-add, etc.) into a single instruction. *Duplex* encodings express two parallel instructions in a single 32-bit word.

- **Cache memory**

  Memory accesses can be cached or uncached. Separate L1 instruction and data caches exist for program code and data. A unified L2 cache can be partly or wholly configured as tightly-coupled memory (TCM).

- **Virtual memory**

  Memory is addressed virtually, with virtual-to-physical memory mapping handled by a resident OS. Virtual memory supports the implementation of memory management and memory protection in a hardware-independent manner.

1.3 Functional units

Figure 1-1 shows the major functional units of the Hexagon V6x processor architecture:

- Memory and registers
- Instruction sequencer
- Execution units
- Load/store units

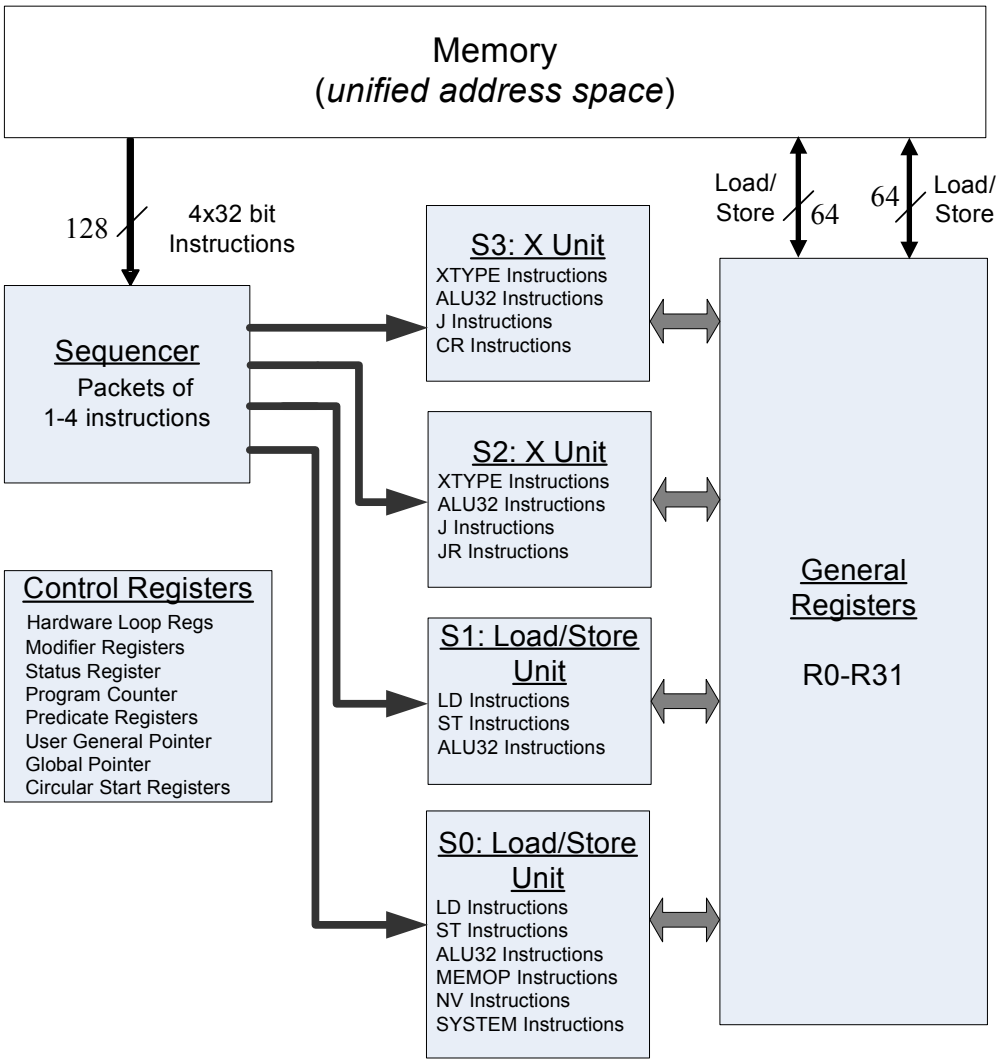


Figure 1-1 Hexagon V6x processor architecture

1.3.1 Memory

The Hexagon processor features a unified byte-addressable memory. This memory has a single 32-bit virtual address space which holds both instructions and data. It operates in little-endian mode.

1.3.2 Registers

The Hexagon processor has two sets of registers: general registers and control registers.

The general registers include thirty-two 32-bit registers (named `R0` through `R31`) which can be accessed either as single registers or as aligned 64-bit register pairs. The general registers are used to contain all pointer, scalar, vector, and accumulator data.

The control registers include special-purpose registers such as program counter, status register, loop registers, etc.

1.3.3 Sequencer

The instruction sequencer processes packets of one to four instructions in each cycle. If a packet contains more than one instruction, the instructions are executed in parallel.

The instruction combinations allowed in a packet are limited to the instruction types that can be executed in parallel in the four execution units (as shown in Figure 1-1).

1.3.4 Execution units

The dual execution units are identical: each includes a 64-bit shifter and a vector multiply/accumulate unit with four 16x16 multipliers to support both scalar and vector instructions.

These units also perform 32- and 64-bit ALU instructions, and jump and loop instructions.

NOTE Each execution unit supports floating-point instructions.

1.3.5 Load/store units

The two load/store units can operate on signed or unsigned bytes, halfwords (16-bit), words (32-bit), or double words (64-bit).

To increase the number of instruction combinations allowed in packets, the load units also support 32-bit ALU instructions.

1.4 Instruction set

In order for the Hexagon processor to achieve large amounts of work per cycle, the instruction set was designed with the following properties:

- Static grouping (VLIW) architecture
- Static fusing of simple dependent instructions
- Extensive compound instructions
- A large set of SIMD and application-specific instructions

To support efficient compilation, the instruction set is designed to be orthogonal with respect to registers, addressing modes, and load/store access size.

1.4.1 Addressing modes

The Hexagon processor supports the following memory addressing modes:

- 32-bit absolute
- 32-bit absolute-set
- Absolute with register offset
- Global pointer relative
- Indirect
- Indirect with offset
- Indirect with register offset
- Indirect with auto-increment (immediate or register)
- Circular with auto-increment (immediate or register)
- Bit-reversed with auto-increment register

For example:

```
R2 = memw(##myvariable)
R2 = memw(R3=##myvariable)
R2 = memw(R4<<#3+##myvariable)
R2 = memw(GP+#200)
R2 = memw(R1)
R2 = memw(R3+#100)
R2 = memw(R3+R4<<#2)
R2 = memw(R3++#4)
R2 = memw(R0++M1)
R0 = memw(R2++#8:circ(M0))
R0 = memw(R2++I:circ(M0))
R2 = memw(R0++M1:brev)
```

Auto-increment with register addressing uses one of the two dedicated address-modify registers `M0` and `M1` (which are part of the control registers).

NOTE Atomic memory operations (load locked/store conditional) are supported to implement multi-thread synchronization.

1.4.2 Scalar operations

The Hexagon processor includes the following scalar operations on fixed-point data:

- Multiplication of 16-bit, 32-bit, and complex data
- Addition and subtraction of 16-, 32-, and 64-bit data (with and without saturation)
- Logical operations on 32- and 64-bit data (And, Or, XOR, Not)
- Shifts on 32- and 64-bit data (arithmetic and logical)
- Min/max, negation, absolute value, parity, norm, swizzle
- Compares of 8-, 16-, 32-, and 64-bit data
- Sign and zero extension (8- and 16- to 32-bit, 32- to 64-bit)
- Bit manipulation
- Predicate operations

1.4.3 Vector operations

The Hexagon processor includes the following vector operations on fixed-point data:

- Multiplication (halfwords, word by half, vector reduce, dual multiply)
- Addition and subtraction of word and halfword data
- Shifts on word and halfword data (arithmetic and logical)
- Min/max, average, negative average, absolute difference, absolute value
- Compares of word, halfword, and byte data
- Reduce, sum of absolute differences on unsigned bytes
- Special-purpose data arrangement (such as pack, splat, shuffle, align, saturate, splice, truncate, complex conjugate, complex rotate, zero extend)

NOTE Certain vector operations support automatic scaling, saturation, and rounding.

For example, the following instruction performs a vector operation:

```
R1:0 += vrmpyh(R3:2,R5:4)
```

It is defined to perform the following operations in one cycle:

```
R1:0 += ((R2.L * R4.L) +
    (R2.H * R4.H) +
    (R3.L * R5.L) +
    (R3.H * R5.H)
         )
```

Figure 1-2 shows a schematic of this instruction type.

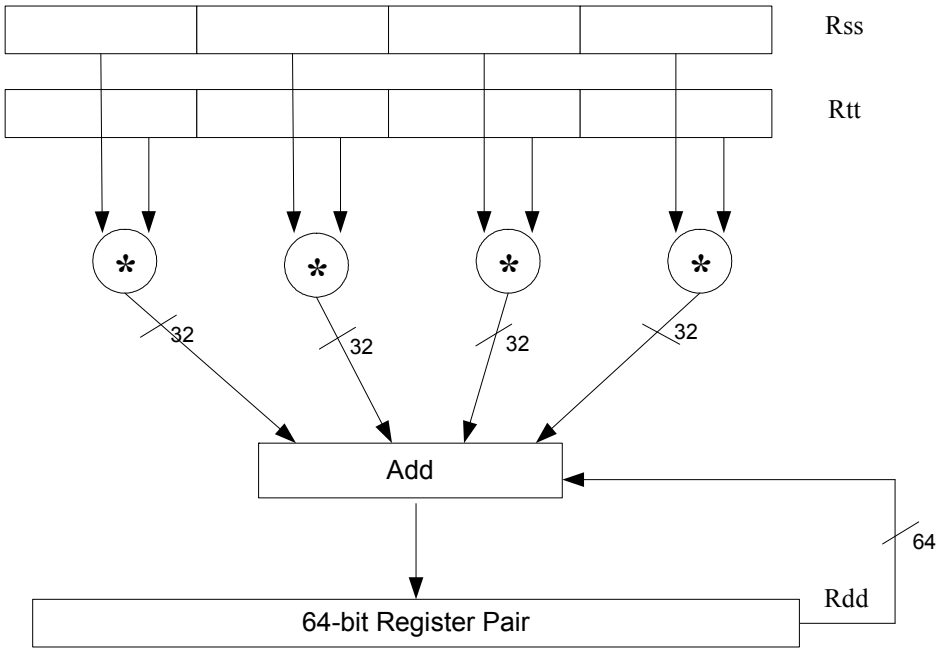


Figure 1-2 Vector instruction example

1.4.4 Floating-point operations

The Hexagon processor includes the following operations on floating-point data:

- Addition and subtraction
- Multiplication (with optional scaling)
- Min/max/compare
- Reciprocal/square root approximation
- Format conversion

1.4.5 Program flow

The Hexagon processor supports zero-overhead hardware loops. For example:

```
  loop0(start,#3)      // loop 3 times
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

The loop instructions support nestable loops, with few restrictions on their use.

Software branches use a predicated branch mechanism. Explicit compare instructions generate a predicate bit, which is then tested by conditional branch instructions. For example:

```
P1 = cmp.eq(R2, R3)
if (P1) jump end
```

Jumps and subroutine calls can be conditional or unconditional, and support both PC-relative and register indirect addressing modes. For example:

```
jump end
jumpr R1
call function
callr R2
```

The subroutine call instructions store the return address in register `R31`. Subroutine returns are performed using a jump indirect instruction through this register. For example:

```
jumpr R31     // subroutine return
```

1.4.6 Instruction packets

Sequences of instructions can be explicitly grouped into packets for parallel execution. For example:

```
{
    R8 = memh(R3++#2)
    R12 = memw(R1++#4)
    R = mpy(R10,R6):<<1:sat
    R7 = add(R9,#2)
}
```

Brace characters are used to delimit the start and end of an instruction packet.

Packets can be from one to four instructions long. Packets of varying length can be freely mixed in a program.

Packets have various restrictions on the allowable instruction combinations. The primary restriction is determined by the instruction class of the instructions in a packet.

1.4.7 Dot-new instructions

In many cases, a predicate or general register can be both generated and used in the same instruction packet. This feature is expressed in assembly language by appending the suffix "`.new`" to the specified register. For example:

```
{
P0 = cmp.eq(R2,#4)
if (P0.new) R3 = memw(R4)
if (!P0.new) R5 = #5
}

{
R2 = memh(R4+#8)
memw(R5) = R2.new
}
```

1.4.8 Compound instructions

Certain common operation pairs (add-accumulate, shift-add, deallocframe-return, etc.) can be performed by compound instructions. Using compound instructions reduces code size and improves code performance.

1.4.9 Duplex instructions

A subset of the most common instructions (load, store, branch, ALU) can be packed together in pairs into single 32-bit instructions known as *duplex* instructions. Duplex instructions reduce code size.

1.4.10 Instruction classes

The instructions are assigned to specific instruction classes. Classes are important for two reasons:

- Only certain combinations of instructions can be written in parallel (as shown in Figure 1-1), and the allowable combinations are specified by instruction class.
- Instruction classes logically correspond with instruction types, so they serve as mnemonics for looking up specific instructions.

Figure 1-3 presents an overview of the instruction classes and how they can be grouped together.

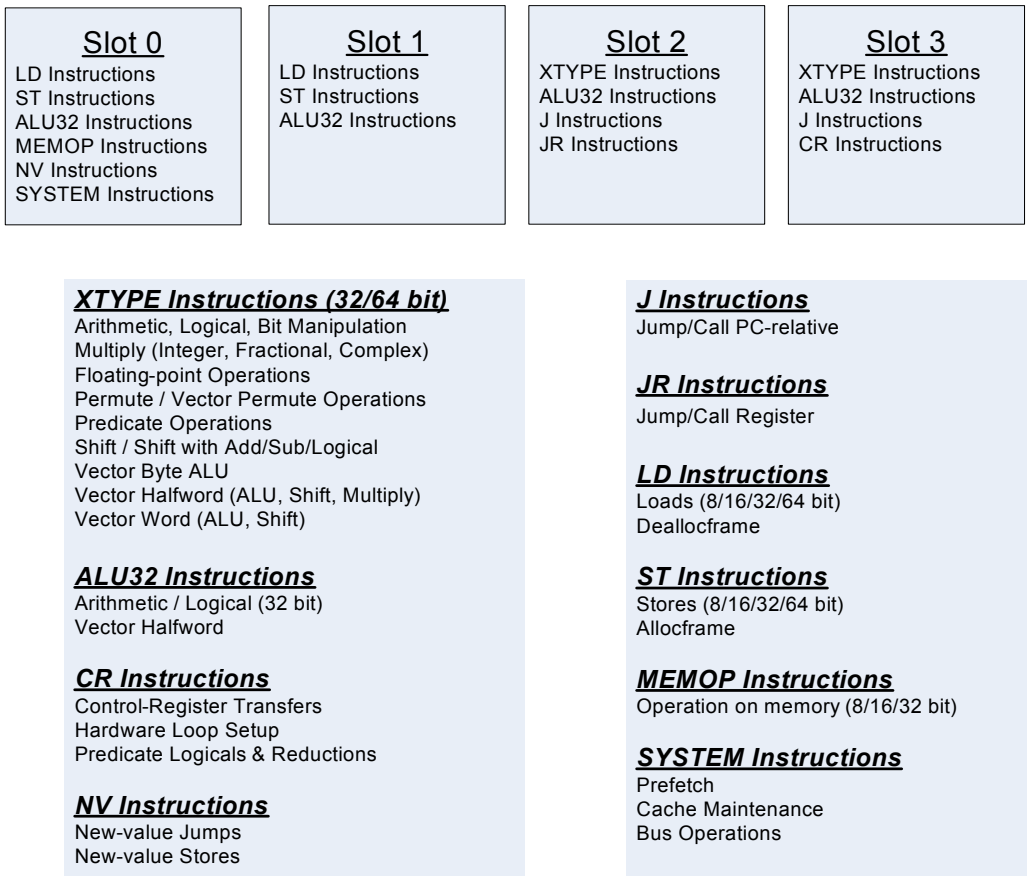


Figure 1-3 Instruction classes and combinations

1.4.11 Instruction intrinsics

To support efficient coding of the time-critical sections of a program (without resorting to assembly language), the C compilers support intrinsics which are used to directly express Hexagon processor instructions from within C code. For example:

```
int main()
{
    long long v1 = 0xFFFF0000FFFF0000;
    long long v2 = 0x0000FFFF0000FFFF;
    long long result;

    // find the minimum for each half-word in 64-bit vector
    result = Q6_P_vminh_PP(v1,v2);
}
```

Intrinsics are defined for most of the Hexagon processor instructions.

1.5 Processor versions

This manual describes version V60 and V61 of the Hexagon processor:

The V60 Hexagon processor is the sixth version of the Hexagon processor architecture. While remaining functionally equivalent to V56 at the user level, V60 adopts a new micro-architecture which provides higher performance for modem and audio applications.

V60 extends the V56 architecture with the following features:

- Dual-cluster micro-architecture
- Simultaneous multi-threading (SMT)

The V61 Hexagon processor extends the V60 architecture with the following user-level changes:

- New instructions (Vector absolute difference bytes, Vector add compare and select maximum halfwords, Vector minimum bytes (conditional), Vector splat bytes (64-bit), Vector truncate (64-bit))
- Stack security (bounds check, stack smashing protection)
- Larger L2 cache (up to 1MB)
- Packet counting
- Qtimer access
- `syncht` and `barrier` instructions now interruptible
- `dccleana`, `dccleaninva`, and `dcinva` have different permissions

For more information on V56 see the *Hexagon V5x Programmer's Reference Manual*.

1.6 Using the manual

This manual describes the Hexagon processor architecture and instruction set.

- Chapter 1, *Introduction*, presents an overview of the Hexagon processor and the manual.
- Chapter 2, *Registers*, describes the processor registers.
- Chapter 3, *Instructions*, describes instructions and instruction packets.
- Chapter 4, *Data Processing*, presents an overview of the data-processing operations.
- Chapter 5, *Memory*, describes the memory address space and addressing modes.
- Chapter 6, *Conditional Execution*, describes the conditional execution model.
- Chapter 7, *Program Flow*, describes instructions and events which affect the program flow, including loops, jumps, and subroutine calls.
- Chapter 8, *Software Stack*, describes the software stack and stack operations.
- Chapter 9, *PMU Events*, describes events generated by the Hexagon Performance Monitor Unit (PMU).
- Chapter 10, *Instruction Encoding*, describes the encoding of instruction packets and operands.
- Chapter 11, *Instruction Set*, describes the processor instruction set.

1.7 Notation

This section presents the notational conventions used in this document to describe Hexagon processor instructions:

- Instruction syntax
- Register operands
- Numeric operands

NOTE The notation described here does not appear in actual assembly language instructions. It is used only to specify the instruction syntax and behavior.

1.7.1 Instruction syntax

The following notation is used to describe the syntax of instructions:

- `Monospaced font is used for instructions`
- Square brackets enclose optional items (e.g., [`:sat`], means that saturation is optional)
- Braces are used to indicate a choice of items (e.g., {`Rs,#s16`}, means that either Rs or a signed 16-bit immediate can be used)

1.7.2 Register operands

The following notation is used to describe register operands in the syntax and behavior of instructions:

R*ds*[.*elst*]

The *ds* field indicates the register operand type and bit size (as defined in Table 1-1).

Table 1-1 Register symbols

Symbol	Operand Type	Size (in Bits)
d	Destination	32
dd		64
s	First source	32
ss		64
t	Second source	32
tt		64
u	Third source	32
uu		64
x	Source *and* destination	32
xx		64

Examples of *ds* field (describing instruction syntax):

```
Rd = neg(Rs)            // Rd -> 32-bit dest, Rs 32-bit source
Rd = xor(Rs,Rt)         // Rt -> 32-bit second source
Rx = insert(Rs,Rtt)     // Rx -> both source and dest
```

Examples of *ds* field (describing instruction behavior):

```
Rdd = Rss + Rtt         // Rdd, Rss, Rtt -> 64-bit registers
```

The optional *elst* field (short for "element size and type") is used to specify parts of a register when the register is used as a vector. It can specify the following values:

- A signed or unsigned byte, halfword, or word within the register (as defined in Figure 1-4)
- A bit-field within the register (as defined in Table 1-2)

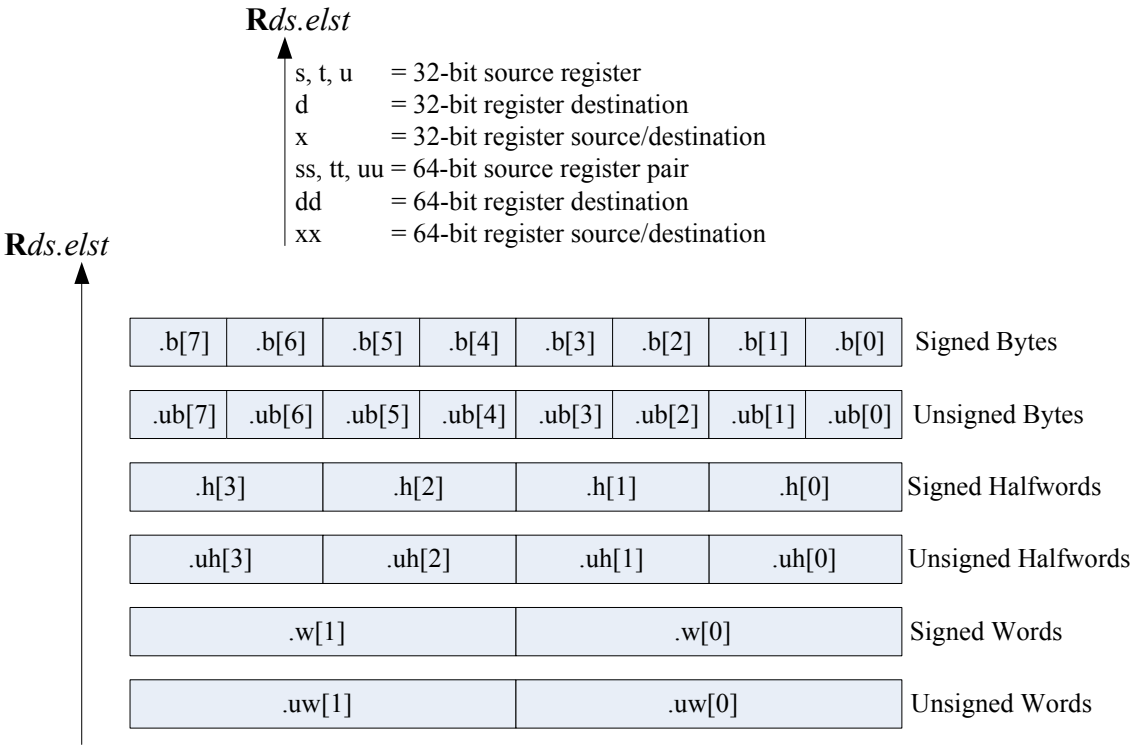


Figure 1-4 Register field symbols

Table 1-2 Register bit field symbols

Symbol	Meaning
.sN	Bits [N-1:0] are treated as a N-bit signed number. For example, R0.s16 means that the least significant 16-bits of R0 are treated as a 16-bit signed number.
.uN	Bits [N-1:0] are treated as a N-bit unsigned number.
.H	The most-significant 16 bits of a 32-bit register.
.L	The least-significant 16 bits of a 32-bit register.

Examples of *elst* field:

```
EA = Rt.h[1]               // .h[1] -> bit field 31:16 in Rt
Pd = (Rss.u64 > Rtt.u64)   // .u64  -> unsigned 64-bit value
Rd = mpyu(Rs.L,Rt.H)       // .L/.H -> low/high 16-bit fields
```

NOTE The control and predicate registers use the same notation as the general registers, but are written as `Cx` and `Px` (respectively) instead of `Rx`.

1.7.3 Numeric operands

Table 1-3 describes the notation used to describe numeric operands in the syntax and behavior of instructions:

Table 1-3 Instruction operands

Symbol	Meaning	Min	Max
#uN	Unsigned N-bit immediate value	0	2^N-1
#sN	Signed N-bit immediate value	-2^{N-1}	2^{N-1}-1
#mN	Signed N-bit immediate value	$-(2^{N-1}-1)$	2^{N-1}-1
#uN:S	Unsigned N-bit immediate value representing integral multiples of 2^S in specified range	0	$(2^N-1) \times 2^S$
#sN:S	Signed N-bit immediate value representing integral multiples of 2^S in specified range	$(-2^{N-1}) \times 2^S$	$(2^{N-1}-1) \times 2^S$
#rN:S	Same as #sN:S, but value is offset from PC of current packet	$(-2^{N-1}) \times 2^S$	$(2^{N-1}-1) \times 2^S$
##	Same as #, but associated value (u,s,m,r) is 32 bits	–	–
usat_N	Saturate value to unsigned N-bit number	0	2^N-1
sat_N	Saturate value to signed N-bit number	-2^{N-1}	2^{N-1}-1

`#uN`, `#sN`, and `#mN` are used to specify immediate operands in instructions. The `#` symbol appears in the actual instruction to indicate the immediate operand.

`#rN` is used to specify loop and branch destinations in instructions. In this case the `#` symbol does *not* appear in the actual instruction; instead, the entire `#rN` symbol (including its `:S` suffix) is expressed as a loop or branch symbol whose numeric value is determined by the assembler and linker. For example:

```
call my_proc            // instruction example
```

The `:S` suffix indicates that the `S` least-significant bits in a value are implied zero bits and therefore not encoded in the instruction. The implied zero bits are called *scale bits*.

For example, `#s4:2` denotes a signed immediate operand represented by four bits encoded in the instruction, and two scale bits. The possible values for this operand are -32, -28, -24, -20, -16, -12, -8, -4, 0, 4, 8, 12, 16, 20, 24, and 28.

`##` is used to specify a 32-bit immediate operand in an instruction (including a loop or branch destination). The `##` symbol appears in the actual instruction to indicate the operand.

Examples of operand symbols:

```
Rd = add(Rs,#s16)      // #s16    -> signed 16-bit imm value
Rd = memw(Rs++#s4:2)   // #s4:2   -> scaled signed 4-bit imm value
call #r22:2            // #r22:2 -> scaled 22-bit PC-rel addr value
Rd = ##u32             // ##u32   -> unsigned 32-bit imm value
```

NOTE When an instruction contains more than one immediate operand, the operand symbols are specified in upper and lower case (e.g., `#uN` and `#UN`) to indicate where they appear in the instruction encodings

1.8 Terminology

Table 1-4 lists the symbols used in Hexagon processor instruction names to specify the supported data types.

Table 1-4 Data symbols

Size	Symbol	Type
8-bit	B	Byte
8-bit	UB	Unsigned Byte
16-bit	H	Half word
16-bit	UH	Unsigned Half word
32-bit	W	Word
32-bit	UW	Unsigned Word
64-bit	D	Double word

1.9 Feedback

If you have any comments or suggestions on how to improve this manual, please send them to:

https://support.cdmatech.com

2 Registers

2.1 Overview

This chapter describes the Hexagon processor registers:

- General registers
- Control registers

General registers are used for all general-purpose computation including address generation and scalar and vector arithmetic.

Control registers support special-purpose processor features such as hardware loops and predicates.

2.2 General registers

The Hexagon processor has thirty-two 32-bit general-purpose registers (named `R0` through `R31`). These registers are used to store operands in virtually all the instructions:

- Memory addresses for load/store instructions
- Data operands for arithmetic/logic instructions
- Vector operands for vector instructions

For example:

```
R1 = memh(R0)              // Load from address R0
R4 = add(R2,R3)            // Add
R28 = vaddh(R11,R10)       // Vector add halfword
```

Figure 2-1 shows the general registers.

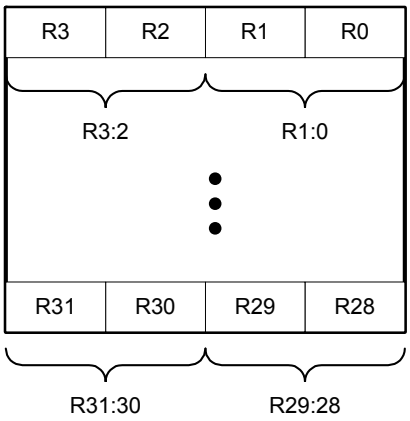


Figure 2-1 General registers

Aliased registers

Three of the general registers – `R29` through `R31` – are used to support subroutines (Section 7.4.2) and the software stack (Chapter 8). These registers are modified implicitly by the subroutine and stack instructions. They have symbol aliases which are used to indicate when these registers are being accessed as subroutine and stack registers.

For example:

```
SP = add(SP, #-8)    // SP is alias of R29
allocframe           // Modifies SP (R29) and FP (R30)
call init            // Modifies LR (R31)
```

Table 2-1 defines the aliased general registers.

Table 2-1 General register aliases

Register	Alias	Name	Description
R29	SP	Stack pointer	Points to topmost element of stack in memory.
R30	FP	Frame pointer	Points to current procedure frame on stack. Used by external debuggers to examine the stack and determine call sequence, parameters, local variables, etc.
R31	LR	Link register	Stores return address of a subroutine call.

Register pairs

The general registers can be specified as register pairs which represent a single 64-bit register. For example:

```
R1:0 = memd(R3)                    // Load doubleword
R7:6 = valignb(R9:8,R7:6, #2)      // Vector align
```

NOTE The first register in a register pair must always be odd-numbered, and the second must be the next lower register.

Table 2-2 lists the general register pairs.

Table 2-2 General register pairs

Register	Register Pair
R0	R1:0
R1	
R2	R3:2
R3	
R4	R5:4
R5	
R6	R7:6
R7	
...	
R24	R25:24
R25	
R26	R27:26
R27	
R28	R29:28
R29 (SP)	
R30 (FP)	R31:30 (LR:FP)
R31 (LR)	

2.3 Control registers

The Hexagon processor includes a set of 32-bit control registers which provide access to processor features such as the program counter, hardware loops, and vector predicates.

Unlike general registers, control registers can be used as instruction operands only in the following cases:

- Instructions that require a specific control register as an operand
- Register transfer instructions

For example:

```
R2 = memw(R0++M1)    // Auto-increment addressing mode (M1)
R9 = PC              // Get program counter (PC)
LC1 = R3             // Set hardware loop count (LC1)
```

NOTE When a control register is used in a register transfer, the other operand must be a general register.

Figure 2-2 shows the control registers.

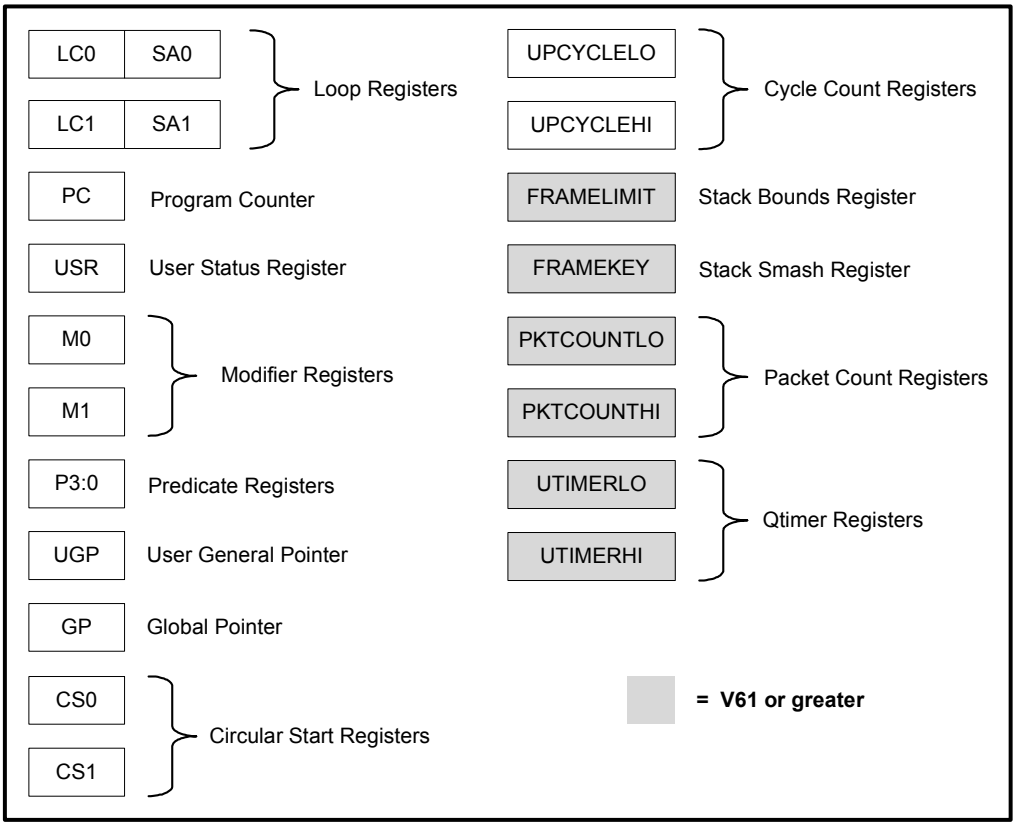


Figure 2-2 Control registers

Aliased registers

The control registers have numeric aliases (`C0` through `C31`).

Table 2-3 lists the aliased control registers.

Table 2-3 Aliased control registers

Register	Alias	Name
SA0	C0	Loop start address register 0
LC0	C1	Loop count register 0
SA1	C2	Loop start address register 1
LC1	C3	Loop count register 1
P3:0	C4	Predicate registers 3:0
reserved	C5	–
M0	C6	Modifier register 0
M1	C7	Modifier register 1
USR	C8	User status register
PC	C9	Program counter
UGP	C10	User general pointer
GP	C11	Global pointer
CS0	C12	Circular start register 0
CS1	C13	Circular start register 1
UPCYCLELO	C14	Cycle count register (low)
UPCYCLEHI	C15	Cycle count register (high)
UPCYCLE	C15:14	Cycle count register
FRAMELIMIT	C16	Frame limit register
FRAMEKEY	C17	Frame key register
PKTCOUNTLO	C18	Packet count register (low)
PKTCOUNTHI	C19	Packet count register (high)
PKTCOUNT	C19:18	Packet count register
reserved	C20-29	–
UTIMERLO	C30	Qtimer register (low)
UTIMERHI	C31	Qtimer register (high)
UTIMER	C31:30	Qtimer register

NOTE The control register numbers (0-31) are used to specify the control registers in instruction encodings (Chapter 10).

The registers shown in gray are supported only in processor versions V61 or greater.

Register pairs

The control registers can be specified as register pairs which represent a single 64-bit register. Control registers specified as pairs must use their numeric aliases. For example:

```
C1:0 = R5:4     // C1:0 specifies the LC0/SA0 register pair
```

NOTE The first register in a control register pair must always be odd-numbered, and the second must be the next lower register.

Table 2-4 lists the control register pairs.

Table 2-4 Control register pairs

Register	Register Pair
C0	C1:0
C1	
C2	C3:2
C3	
C4	C5:4
C5	
C6	C7:6
C7	
...	
C30	C31:30
C31	

2.3.1 Program counter

The Program Counter (PC) register points to the next instruction packet to execute (Section 3.4). It is modified implicitly by instruction execution, but can be read directly. For example:

```
R7 = PC      // Get program counter
```

NOTE The PC register is read-only: writing to it has no effect.

2.3.2 Loop registers

The Hexagon processor includes two sets of loop registers to support nested hardware loops (Section 7.3). Each hardware loop is implemented with a pair of registers containing the loop count and loop start address. The loop registers are modified implicitly by the `loop` instruction, but can also be accessed directly. For example:

```
loop0(start, R4)  // Modifies LC0 and SA0 (LC0=R4, SA0=&start)
LC1 = R22         // Set loop1 count
R9 = SA1          // Get loop1 start address
```

Table 2-5 defines the loop registers.

Table 2-5 Loop registers

Register	Name	Description
LC0, LC1	Loop count	Number of loop iterations to execute.
SA0, SA1	Loop start address	Address of first instruction in loop.

2.3.3 User status register

The user status register (USR) stores processor status and control bits that are accessible by user programs. The status bits contain the status results of certain instructions, while the control bits contain user-settable processor modes for hardware prefetching. For example:

```
R9:8 = vaddw(R9:8, R3:2):sat     // Vector add words
R6 = USR                         // Get saturation status
```

USR stores the following status and control values:

- Cache prefetch enable (Section 5.10.6)
- Cache prefetch status (Section 5.10.6)
- Floating point modes (Section 4.4.4)
- Floating point status (Section 4.4.4)
- Hardware loop configuration (Section 7.3)
- Sticky saturation overflow (Section 4.3.2)

NOTE A user control register transfer to USR cannot be grouped in an instruction packet with a floating point instruction (Section 4.4.4).

Whenever a transfer to USR changes the Enable trap bits [29:25], an `isync` instruction (Section 5.11) must be executed before the new exception programming can take effect.

Table 2-6 defines the user status register.

Table 2-6 User status register

Name	R/W	Bits	Field	Description
USR		32		User Status Register
	R	31	PFA	L2 Prefetch Active. 1: l2fetch instruction in progress 0: l2fetch finished (or inactive) Set when non-blocking l2fetch instruction is prefetching requested data. Remains set until l2fetch prefetch operation is completed (or not active).
	R	30	reserved	Return 0 if read. Reserved for future expansion. To remain compatible with future processor versions, software should always write this field with the same value read from the field.
	R/W	29	FPINEE	Enable trap on IEEE Inexact.
	R/W	28	FPUNFE	Enable trap on IEEE Underflow.
	R/W	27	FPOVFE	Enable trap on IEEE Overflow.
	R/W	26	FPDBZE	Enable trap on IEEE Divide-By-Zero.
	R/W	25	FPINVE	Enable trap on IEEE Invalid.
	R	24	reserved	Reserved
	R/W	23:22	FPRND	Rounding Mode for Floating-Point Instructions. 00: Round to nearest, ties to even (default) 01: Toward zero 10: Downward (toward negative infinity) 11: Upward (toward positive infinity)
	R	21:20	reserved	Return 0 if read. Reserved for future expansion. To remain compatible with future processor versions, software should always write this field with the same value read from the field.
	R	19:18	reserved	Reserved
	R	17	reserved	Return 0 if read. Reserved for future expansion. To remain compatible with future processor versions, software should always write this field with the same value read from the field.
	R/W	16:15	HFI	L1 Instruction Prefetch. 00: Disable 01: Enable (1 line) 10: Enable (2 lines)

Table 2-6 User status register (Continued)

Name	R/W	Bits	Field	Description
	R/W	14:13	HFD	L1 Data Cache Prefetch. Four levels are defined from disabled to Aggressive. It is implementation-defined how these levels should be interpreted. 00: Disable 01: Conservative 10: Moderate 11: Aggressive
	R/W	12	PCMME	Enable packet counting in Monitor mode.
	R/W	11	PCGME	Enable packet counting in Guest mode.
	R/W	10	PCUME	Enable packet counting in User mode.
	R/W	9:8	LPCFGE	Hardware Loop Configuration. Number of loop iterations (0-3) remaining before pipeline predicate should be set.
	R	7:6	reserved	Return 0 if read. Reserved for future expansion. To remain compatible with future processor versions, software should always write this field with the same value read from the field.
	R/W	5	FPINPF	Floating-point IEEE Inexact Sticky Flag.
	R/W	4	FPUNFF	Floating-point IEEE Underflow Sticky Flag.
	R/W	3	FPOVFF	Floating-point IEEE Overflow Sticky Flag.
	R/W	2	FPDBZF	Floating-point IEEE Divide-By-Zero Sticky Flag.
	R/W	1	FPINVF	Floating-point IEEE Invalid Sticky Flag.
	R/W	0	OVF	Sticky Saturation Overflow. 1: Saturation occurred 0: No saturation Set when saturation occurs while executing instruction that specifies optional saturation. Remains set until explicitly cleared by a USR = Rs instruction.

2.3.4 Modifier registers

The modifier registers (M0-M1) are used in the following addressing modes:

- Indirect auto-increment register addressing
- Circular addressing
- Bit-reversed addressing

Indirect auto-increment

In indirect auto-increment register addressing (Section 5.8.9) the modifier registers store a signed 32-bit value which specifies the increment (or decrement) value. For example:

```
M1 = R0                 // Set modifier register
R3 = memw(R2++M1)       // Load word
```

Table 2-7 defines the modifier registers as used in auto-increment register addressing.

Table 2-7 Modifier registers (indirect auto-increment addressing)

Register	Name	Description
M0, M1	Increment	Signed auto-increment value.

Circular

In circular addressing (Section 5.8.10) the modifier registers store the circular buffer length and related "K" and "I" values. For example:

```
M0 = R7                         // Set modifier register
R0 = memb(R2++#4:circ(M0))      // Load from circ buffer pointed
                                //  to by R2 with size/K vals in M0

R0 = memb(R7++I:circ(M1))       // Load from circ buffer pointed
                                //  to by R7 with size/K/I vals in M1
```

Table 2-8 defines the modifier registers as used in circular addressing.

Table 2-8 Modifier registers (circular addressing)

Name	R/W	Bits	Field	Description
M0, M1		32		Circular buffer specifier.
	R/W	31:28	I[10:7]	I value (MSB - see Section 5.8.11)
	R/W	27:24	K	K value (Section 5.8.10)
	R/W	23:17	I[6:0]	I value (LSB)
	R/W	16:0	Length	Circular buffer length

Bit-reversed

In bit-reversed addressing (Section 5.8.12) the modifier registers store a signed 32-bit value which specifies the increment (or decrement) value. For example:

```
M1 = R7                  // Set modifier register
R2 = memub(R0++M1:brev)  // The address is (R0.H | bitrev(R0.L))
                         // The orginal R0 (not reversed) is added
                         // to M1 and written back to R0
```

Table 2-9 defines the modifier registers as used in bit-reversed addressing.

Table 2-9 Modifier registers (bit-reversed addressing)

Register	Name	Description
M0, M1	Increment	Signed auto-increment value.

2.3.5 Predicate registers

The predicate registers (P0-P3) store the status results of the scalar and vector compare instructions (Chapter 6). For example:

```
P1 = cmp.eq(R2, R3)       // Scalar compare
if (P1) jump end          // Jump to address (conditional)
R8 = P1                   // Get compare status (P1 only)
P3:0 = R4                 // Set compare status (P0-P3)
```

The four predicate registers can be specified as a register quadruple (P3:0) which represents a single 32-bit register.

NOTE Unlike the other control registers, the predicate registers are only 8 bits wide because vector compares return a maximum of 8 status results.

Table 2-10 defines the predicate registers.

Table 2-10 Predicate registers

Register	Bits	Description
P0, P1, P2, P3	8	Compare status results.
P3:0	32	Compare status results.
	31:24	P3 register
	23:16	P2 register
	15:8	P1 register
	7:0	P0 register

2.3.6 Circular start registers

The circular start registers (CS0 - CS1) store the start address of a circular buffer in circular addressing (Section 5.8.10). For example:

```
CS0 = R5                      // Set circ start register
M0 = R7                       // Set modifier register
R0 = memb(R2++#4:circ(M0))    // Load from circ buffer pointed
                              //  to by CS0 with size/K vals in M0
```

Table 2-11 defines the circular start registers.

Table 2-11 Circular start registers

Register	Name	Description
CS0, CS1	Circular Start	Circular buffer start address.

2.3.7 User general pointer register

The user general pointer (UGP) register is a general-purpose control register. For example:

```
R9 = UGP      // Get UGP
UGP = R3      // Set UGP
```

NOTE UGP is typically used to store the address of thread local storage.

Table 2-12 defines the user general pointer register.

Table 2-12 User general pointer register

Register	Name	Description
UGP	User General Pointer	General-purpose control register.

2.3.8 Global pointer

The Global Pointer (GP) is used in GP-relative addressing. For example:

```
GP = R7                 // Set GP
R2 = memw(GP+#200)      // GP-relative load
```

Table 2-13 defines the global pointer register.

Table 2-13 Global pointer register

Name	R/W	Bits	Field	Description
GP		32		Global Pointer Register
	R/W	31:7	GDP	Global Data Pointer (Section 5.8.4).
	R	6:0	reserved	Return 0 if read. Reserved for future expansion. To remain forward-compatible with future processor versions, software should always write this field with the same value read from the field.

2.3.9 Cycle count registers

The cycle count registers (UPCYCLELO - UPCYCLEHI) store a 64-bit value containing the current number of processor cycles executed since the Hexagon processor was last reset. For example:

```
R5 = UPCYCLEHI      // Get cycle count (high)
R4 = UPCYCLELO      // Get cycle count (low)
R5:4 = UPCYCLE      // Get cycle count
```

NOTE The RTOS must grant permission to access these registers. Without this permission, reading these registers from user code always returns zero.

Table 2-14 defines the cycle count registers.

Table 2-14 Cycle count registers

Register	Name	Description
UPCYCLELO	Cycle count (low)	Processor cycle count (low 32 bits)
UPCYCLEHI	Cycle count (high)	Processor cycle count (high 32 bits)
UPCYCLE	Cycle count	Processor cycle count (64 bits)

2.3.10 Frame limit register

The frame limit register (FRAMELIMIT) stores the low address of the memory area reserved for the software stack (Section 8.4.1). For example:

```
R9 = FRAMELIMIT       // Get frame limit register
FRAMELIMIT = R3       // Set frame limit register
```

Table 2-15 defines the frame limit register.

Table 2-15 Frame limit register

Register	Name	Description
FRAMELIMIT	Frame Limit	Low address of software stack area.

2.3.11 Frame key register

The frame key register (FRAMEKEY) stores the key value that is used to XOR-scramble return addresses when they are stored on the software stack (Section 8.4.2). For example:

```
R2 = FRAMEKEY      // Get frame key register
FRAMEKEY = R1      // Set frame key register
```

Table 2-16 defines the frame key register.

Table 2-16 Frame key register

Register	Name	Description
FRAMEKEY	Frame Key	Key used to scramble return addresses stored on software stack.

2.3.12 Packet count registers

The packet count registers (PKTCOUNTLO - PKTCOUNTHI) store a 64-bit value containing the current number of instruction packets executed since a PKTCOUNT register was last written to. For example:

```
R9 = PKTCOUNTHI     // Get packet count (high)
R8 = PKTCOUNTLO     // Get packet count (low)
R9:8 = PKTCOUNT     // Get packet count
```

Packet counting can be configured to operate only in specific sets of processor modes (e.g., user mode only, or guest and monitor modes only). The configuration for each mode is controlled by bits [12:10] in the user status register (Section 2.3.3).

Packets with exceptions are not counted as committed packets.

NOTE Each hardware thread has its own set of packet count registers.

The RTOS must grant permission to access these registers. Without this permission, reading these registers from user code always returns zero.

When a value is written to a PKTCOUNT register, the 64-bit packet count value is incremented before the value is stored in the register.

Table 2-17 defines the packet count registers.

Table 2-17 Packet count registers

Register	Name	Description
PKTCOUNTLO	Packet count (low)	Processor packet count (low 32 bits)
PKTCOUNTHI	Packet count (high)	Processor packet count (high 32 bits)
PKTCOUNT	Cycle count	Processor packet count (64 bits)

2.3.13 Qtimer registers

The Qtimer registers (UTIMERLO - UTIMERHI) provide access to the Qtimer global reference count value. They enable Hexagon software to read the 64-bit time value without having to perform an expensive AHB load. For example:

```
R5 = UTIMERHI      // Get Qtimer reference count (high)
R4 = UTIMERLO      // Get Qtimer reference count (low)
R5:4 = UTIMER      // Get Qtimer reference count
```

These registers are read-only – they are automatically updated by hardware to always contain the current Qtimer value.

NOTE The RTOS must grant permission to access these registers. Without this permission, reading these registers from user code always returns zero.

Table 2-18 defines the Qtimer registers.

Table 2-18 Qtimer registers

Register	Name	Description
UTIMERLO	Qtimer (low)	Qtimer global reference count (low 32 bits)
UTIMERHI	Qtimer (high)	Qtimer global reference count (high 32 bits)
UTIMER	Qtimer	Qtimer global reference count (64 bits)

3 Instructions

3.1 Overview

This chapter covers the following topics:

- Instruction syntax
- Instruction classes
- Instruction packets
- Instruction intrinsics
- Compound instructions
- Duplex instructions

The instruction encoding is described in Chapter 10.

For detailed descriptions of the Hexagon processor instructions see Chapter 11.

3.2 Instruction syntax

Most Hexagon processor instructions have the following syntax:

```
dest = instr_name(source1,source2,...)[:option1][:option2]...
```

The item specified on the left-hand side (LHS) of the equation is assigned the value specified by the right-hand side (RHS). For example:

```
R2 = add(R3,R1)   // Add R3 and R1, assign result to R2
```

Table 3-1 lists symbols commonly used in Hexagon processor instructions.

Table 3-1 Instruction symbols

Symbol	Example	Meaning
=	`R2 = R3`	Assignment of RHS to LHS
#	`R1 = #1`	Immediate value
0x	`0xBABE`	Hexadecimal number prefix
mem*XX*	`R2 = memub(R3)`	Memory access *XX* specifies access size and type
;	`R2 = R3; R4 = R5;`	Instruction delimiter, or end of instruction
{ … }	`{R2 = R3; R5 = R6}`	Instruction packet delimiter
(…)	`R2 = memw(R0 + #100)`	Source list delimiter
:endloop*X*	`:endloop0`	Loop end *X* specifies loop instruction (0 or 1)
:t	`if (P0.new) jump:t target`	Direction hint (jump taken)
:nt	`if (!P1.new) jump:nt target`	Direction hint (jump not taken)
:sat	`R2 = add(R1,R2):sat`	Saturate result
:rnd	`R2 = mpy(R1.H,R2.H):rnd`	Round result
:carry	`R5:4=add(R1:0,R3:2,P1):carry`	Predicate used as carry input and output
:<<16	`R2 = add(R1.L,R2.L):<<16`	Shift result left by halfword

3.3 Instruction classes

The Hexagon processor instructions are assigned to specific *instruction classes*. Classes determine what combinations of instructions can be written in parallel (Section 3.4).

Instruction classes logically correspond with instruction types. For instance, the ALU32 class contains ALU instructions which operate on 32-bit operands.

Table 3-2 lists the instruction classes and subclasses.

Table 3-2 Instruction classes

Class	Subclass	Description	Section
XTYPE	–	Various operations	Section 11.12
	ALU	64-bit ALU operations	Section 11.12.1
	Bit	Bit operations	Section 11.12.2
	Complex	Complex math (using real and imaginary numbers)	Section 11.12.3
	Floating point	Floating point operations	Section 11.12.4
	Multiply	Multiply operations	Section 11.12.5
	Permute	Vector permute and format conversion (pack, splat, swizzle)	Section 11.12.6
	Predicate	Predicate operations	Section 11.12.7
	Shift	Shift operations (with optional ALU operations)	Section 11.12.8
ALU32	–	32-bit ALU operations	Section 11.3
	ALU	Arithmetic and logical	Section 11.3.1
	Permute	Permute	Section 11.3.2
	Predicate	Predicate operations	Section 11.3.3
CR	–	Control register access, loops	Section 11.4
JR	–	Jumps (register indirect addressing mode)	Section 11.5
J	–	Jumps (PC-relative addressing mode)	Section 11.6
LD	–	Memory load operations	Section 11.7
MEMOP	–	Memory operations	Section 11.8
NV	–	New-value operations	Section 11.9
	Jump	New-value jumps	Section 11.9.1
	Store	New-value stores	Section 11.9.2
ST	–	Memory store operations; alloc stack frame	Section 11.10
SYSTEM	–	Operating system access	Section 11.11
	USER	Application-level access	Section 11.11.1

3.4 Instruction packets

Instructions can be grouped together to form packets of independent instructions which are executed together in parallel. The packets can contain 1, 2, 3, or 4 instructions.

Instruction packets must be explicitly specified in software. They are expressed in assembly language by enclosing groups of instructions in curly braces. For example:

```
{ R0 = R1; R2 = R3 }
```

Various rules and restrictions exist on what types of instructions can be grouped together, and in what order they can appear in the packet. In particular, packet formation is subject to the following constraints:

- *Resource constraints* determine how many instructions of a specific type can appear in a packet. The Hexagon processor has a fixed number of execution units: each instruction is executed on a particular type of unit, and each unit can process at most one instruction at a time. Thus, for example, because the Hexagon processor contains only two load units, an instruction packet with three load instructions is invalid. The resource constraints are described in Section 3.4.3.
- *Grouping constraints* are a small set of rules that apply above and beyond the resource constraints. These rules are described in Section 3.4.4.
- *Dependency constraints* ensure that no write-after-write hazards exist in a packet. These rules are described in Section 3.4.5.
- *Ordering constraints* dictate the ordering of instructions within a packet. These rules are described in Section 3.4.6.
- *Alignment constraints* dictate the placement of packets in memory. These rules are described in Section 3.4.7.

NOTE Individual instructions (which are not explicitly grouped in packets) are executed by the Hexagon processor as packets containing a single instruction.

3.4.1 Packet execution semantics

Packets are defined to have *parallel execution semantics*. Specifically, the execution behavior of a packet is defined as follows:

- First, all instructions in the packet read their source registers in parallel.
- Next, all instructions in the packet execute.
- Finally, all instructions in the packet write their destination registers in parallel.

For example, consider the following packet:

```
{ R2 = R3; R3 = R2; }
```

In the first phase, registers R3 and R2 are read from the register file. Then, after execution, R2 is written with the old value of R3 and R3 is written with the old value of R2. In effect, the result of this packet is that the values of R2 and R3 are swapped.

NOTE Dual stores (Section 5.5), dual jumps (Section 7.8), new-value stores (Section 5.6), new-value compare jumps (Section 7.6.1), and dot-new predicates (Section 6.2.4) have non-parallel execution semantics.

3.4.2 Sequencing semantics

Packets of any length can be freely mixed in code. A packet is considered an atomic unit: in essence, a single large "instruction". From the program perspective a packet either executes to completion or not at all; it never executes only partially. For example, if a packet causes a memory exception, the exception point is established before the packet.

A packet containing multiple load/store instructions may require service from the external system. For instance, consider the case of a packet which performs two load operations that both miss in the cache. The packet requires the data to be supplied by the memory system:

- From the memory system perspective the two resulting load requests are processed serially.
- From the program perspective, however, both load operations must complete before the packet can complete.

Thus, the packet is atomic from the program perspective.

Packets have a single PC address which is the address of the start of the packet. Branches cannot be performed into the middle of a packet.

Architecturally, packets execute to completion – including updating all registers and memory – before the next packet begins. As a result, application programs are not exposed to any pipeline artifacts.

3.4.3 Resource constraints

A packet cannot use more hardware resources than are physically available on the processor. For instance, because the Hexagon processor has only two load units, a packet with three load instructions is invalid. The behavior of such a packet is undefined. The assembler automatically rejects packets that oversubscribe the hardware resources.

The processor supports up to four parallel instructions. The instructions are executed in four parallel pipelines which are referred to as *slots*.The four slots are named Slot 0, Slot 1, Slot 2, and Slot 3. (For more information see Section 1.3.)

NOTE `endloopN` instructions (Section 7.3.2) do not use any slots.

Each instruction belongs to a specific *instruction class* (Section 3.3). For example, jumps belong to instruction class J, while loads belong to instruction class LD. An instruction's class determines which slot it can execute in.

Figure 3-1 shows which instruction classes can be assigned to each of the four slots.

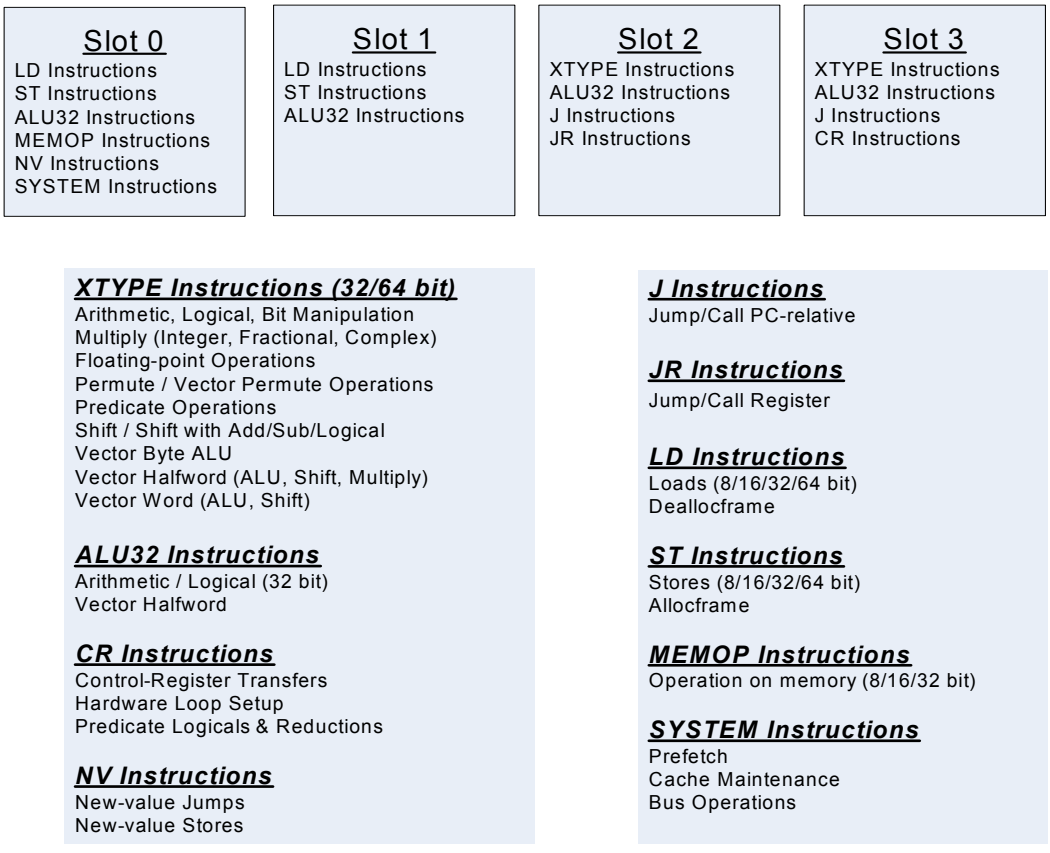


Figure 3-1 Packet grouping combinations

3.4.4 Grouping constraints

A small number of restrictions determines what constitutes a valid packet. The assembler ensures that all packets follow valid grouping rules. If a packet is executed which violates a grouping rule, the behavior is undefined. The following rules must be followed:

- Dot-new conditional instructions (Section 6.2.4) must be grouped in a packet with an instruction that generates dot-new predicates.
- ST-class instructions can be placed in Slot 1. However, in this case Slot 0 must contain a second ST-class instruction (Section 5.5).
- J-class instructions can be placed in Slots 2 or 3. However, only certain combinations of program flow instructions (J or JR) can be grouped together in a packet (Section 7.8). Otherwise, at most one program flow instruction is allowed in a packet.
- JR-class instructions can be placed in Slot 2. However, when encoded in a duplex `jumpr R31` can be placed in Slot 0 (Section 10.4).
- Restrictions exist which limit the instructions that can appear in a packet at the setup or end of a hardware loop (Section 7.3.4).
- A user control register transfer to the control register `USR` cannot be grouped with a floating point instruction (Section 2.3.3).
- The SYSTEM-class instructions include prefetch, cache operations, bus operations, load locked, and store conditional instructions (Section 5.10). These instructions have the following grouping rules:
  - `brkpt`, `trap`, `pause`, `icinva`, `isync`, and `syncht` are *solo instructions*. They must not be grouped with other instructions in a packet.
  - `memw_locked`, `memd_locked`, `l2fetch`, and `trace` must execute on Slot 0. They must be grouped only with ALU32 or (non-FP) XTYPE instructions.
  - `dccleana`, `dcinva`, `dccleaninva`, and `dczeroa` must execute on Slot 0. Slot 1 must be empty or an ALU32 instruction.

3.4.5 Dependency constraints

Instructions in a packet cannot write to the same destination register. The assembler automatically flags such packets as invalid. If the processor executes a packet with two writes to the same general register, an error exception is raised.

If the processor executes a packet which performs multiple writes to the same predicate or control register, the behavior is undefined. Three special cases exist for this rule:

- Conditional writes are allowed to target the same destination register only if at most one of the writes is actually performed (Section 6.2.5).
- The overflow flag in the status register has defined behavior when multiple instructions write to it (Section 2.3.3). Note that instructions that write to the entire user status register (for example, `USR=R2`) are not allowed to be grouped in a packet with any instruction that writes to a bit in the user status register.
- Multiple compare instructions are allowed to target the same predicate register in order to perform a logical AND of the results (Section 6.2.3).

3.4.6 Ordering constraints

In assembly code, instructions can appear in a packet in any order (with the exception of dual jumps – Section 7.8). The assembler automatically encodes instructions in the packet in the proper order.

In the binary encoding of a packet, the instructions must be ordered from Slot 3 down to Slot 0. If the packet contains less than four instructions, any unused slot is skipped – a NOP is unnecessary as the hardware handles the proper spacing of the instructions.

In memory, instructions in a packet must appear in strictly decreasing slot order. Additionally, if an instruction can go in a higher-numbered slot, and that slot is empty, then it must be moved into the higher-numbered slot.

For example, if a packet contains three instructions and Slot 1 is not used, the instructions should be encoded in the packet as follows:

- Slot 3 instruction at lowest address
- Slot 2 instruction follows Slot 3 instruction
- Slot 0 instructions at the last (highest) address

If a packet contains a single load or store instruction, that instruction must go in Slot 0, which is the highest address. As an example, a packet containing both LD and ALU32 instructions must be ordered so the LD is in Slot 0 and the ALU32 in another slot.

3.4.7 Alignment constraints

Packets have the following constraints on their placement or alignment in memory:

- Packets must be word-aligned (32-bit). If the processor executes an improperly aligned packet, it will raise an error exception (Section 7.11).
- Packets should not wrap the 4GB address space. If address wraparound occurs, the processor behavior is undefined.

No other core-based restrictions exist for code placement or alignment.

If the processor branches to a packet which crosses a 16-byte address boundary, the resulting instruction fetch will stall for one cycle. Packets that are jump targets or loop body entries can be explicitly aligned to ensure this does not occur (Section 9.5.2).

3.5 Instruction intrinsics

To support efficient coding of the time-critical sections of a program (without resorting to assembly language), the C compilers support intrinsics which are used to directly express Hexagon processor instructions from within C code.

The following example shows how an instruction intrinsic is used to express the XTYPE instruction "`Rdd = vminh(Rtt,Rss)`":

```
#include <hexagon_protos.h>

int main()
{
    long long v1 = 0xFFFF0000FFFF0000LL;
    long long v2 = 0x0000FFFF0000FFFFLL;
    long long result;

    // find the minimum for each half-word in 64-bit vector
    result = Q6_P_vminh_PP(v1,v2);
}
```

Intrinsics are provided for instructions in the following classes:

- ALU32
- XTYPE
- CR (predicate operations only)
- SYSTEM (`dcfetch` only)

For more information on intrinsics see Chapter 11.

3.6 Compound instructions

The Hexagon processor supports *compound instructions*, which encode pairs of commonly-used operations in a single instruction. For example, each of the following is a single compound instruction:

```
dealloc_return                  // deallocate frame and return
R2 &= and(R1, R0)               // and and and
R7 = add(R4, sub(#15, R3))      // subtract and add
R3 = sub(#20, asl(R3, #16))     // shift and subtract
R5 = add(R2, mpyi(#8, R4))      // multiply and add
{                               // compare and jump
   P0 = cmp.eq (R2, R5)
   if (P0.new) jump:nt target
}
{                               // register transfer and jump
      R2 = #15
      jump target
}
```

Using compound instructions reduces code size and improves code performance.

NOTE Compound instructions (with the exception of X-and-jump, as shown above) have distinct assembly syntax from the instructions they are composed of.

3.7 Duplex instructions

To reduce code size the Hexagon processor supports *duplex instructions*, which encode pairs of commonly-used instructions in a 32-bit instruction container.

Unlike compound instructions (Section 3.6), duplex instructions do not have distinctive syntax – in assembly code they appear identical to the instructions they are composed of. The assembler is responsible for recognizing when a pair of instructions can be encoded as a single duplex rather than a pair of regular instruction words.

In order to fit two instructions into a single 32-bit word, duplexes are limited to a subset of the most common instructions (load, store, branch, ALU), and the most common register operands.

For more information on duplexes see Section 10.3 and Section 10.4.

4 Data Processing

4.1 Overview

The Hexagon processor provides a rich set of operations for processing scalar and vector data.

This chapter presents an overview of the operations provided by the following Hexagon processor instruction classes:

- XTYPE – General-purpose data operations
- ALU32 – Arithmetic/logical operations on 32-bit data

NOTE For detailed descriptions of these instruction classes see Chapter 11.

4.2 Data types

The Hexagon processor provides operations for processing the following data types:

- Fixed-point data
- Floating-point data
- Complex data
- Vector data

4.2.1 Fixed-point data

The Hexagon processor provides operations to process 8-, 16-, 32-, or 64-bit fixed-point data. The data can be either integer or fractional, and in signed or unsigned format.

4.2.2 Floating-point data

The Hexagon processor provides operations to process 32-bit floating-point numbers. The numbers are stored in IEEE single-precision floating-point format.

Per the IEEE standard, certain floating-point values are defined to represent positive or negative infinity, as well as "Not-a-Number" (NaN), which represents values that have no mathematical meaning.

Floating-point numbers can be held in a general register.

4.2.3 Complex data

The Hexagon processor provides operations to process 32- or 64-bit complex data.

Complex numbers include a signed real portion and a signed imaginary portion. Given two complex numbers (a+bi) and (c+di), the complex multiply operations computes both the real portion (ac-bd) and the imaginary portion (ad+bc) in a single instruction.

Complex numbers can be packed in a general register or register pair. When packed, the imaginary portion occupies the most-significant portion of the register or register pair.

4.2.4 Vector data

The Hexagon processor provides operations to process 64-bit vector data.

Vector data types pack multiple data items – bytes, halfwords, or words – into 64-bit registers. Vector data operations are common in video and image processing.

Eight 8-bit bytes can be packed into a 64-bit register.

Figure 4-1 shows an example of a vector byte operation.

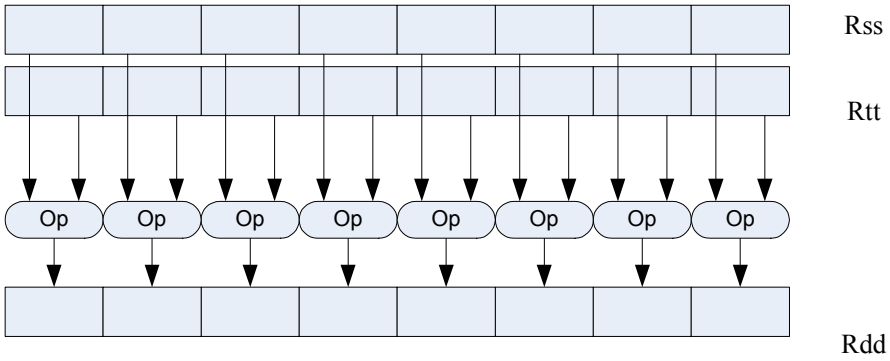


Figure 4-1 Vector byte operation

Four 16-bit halfword values can be packed in a single 64-bit register pair.

Figure 4-2 shows an example of a vector halfword operation.

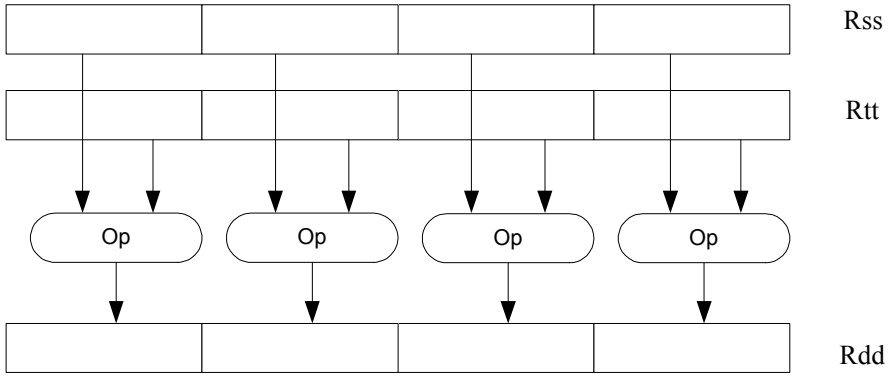


Figure 4-2 Vector halfword operation

Two 32-bit word values can be packed in a single 64-bit register pair.

Figure 4-3 shows an example of a vector word operation.

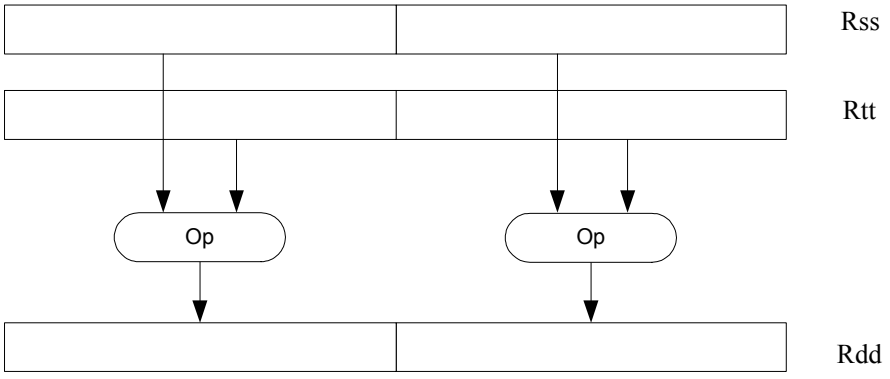


Figure 4-3 Vector word operation

4.3 Instruction options

Some instructions support optional scaling, saturation, and rounding. There are no mode bits controlling these options – instead, they are explicitly specified as part of the instruction name. The options are described in this section.

4.3.1 Fractional scaling

In fractional data format, data is treated as fixed-point fractional values whose range is determined by the word length and radix point position.

Fractional scaling is specified in an instruction by adding the `:<<1` specifier. For example:

```
R3:2 = cmpy(R0,R1):<<1:sat
```

When two fractional numbers are multiplied, the product must be scaled to restore the original fractional data format. The Hexagon processor allows fractional scaling of the product to be specified in the instruction for shifts of 0 and 1. A shift of 1 should be done for Q1.15 numbers, while a shift of 0 should be done for integer multiplication.

4.3.2 Saturation

Certain instructions are available in saturating form. If a saturating arithmetic instruction has a result which is smaller than the minimum value, then the result is set to the minimum value. Similarly, if the operation has a result which is greater than the maximum value, then the result is set to the maximum value.

Saturation is specified in an instruction by adding the `:sat` specifier. For example:

```
R2 = abs(R1):sat
```

The OVF bit in the user status register (Section 2.3.3) is set whenever a saturating operation saturates to the maximum or minimum value. It remains set until explicitly cleared by a control register transfer to USR. For vector-type saturating operations, if any of the individual elements of the vector saturate, then OVF is set.

4.3.3 Arithmetic rounding

Certain signed multiply instructions support optional arithmetic rounding (also known as biased rounding). The arithmetic rounding operation takes a double precision fractional value and adds 0x8000 to the low 16-bits (least significant 16-bit halfword).

Rounding is specified in an instruction by adding the `:rnd` specifier. For example:

```
R2 = mpy(R1.h,R2.h):rnd
```

NOTE Arithmetic rounding can accumulate numerical errors, especially when the number to be rounded is exactly 0.5. This happens most frequently when dividing by 2 or averaging.

4.3.4 Convergent rounding

To address the problem of error accumulation in arithmetic rounding (Section 4.3.3), the Hexagon processor includes four instructions that support positive and negative averaging with a convergent rounding option.

These instructions work as follows:

1. Compute (A+B) or (A-B) for AVG and NAVG respectively.
2. Based on the two least-significant bits of the result, add a rounding constant as follows:
   - If the two LSBs are 00, add 0
   - If the two LSBs are 01, add 0
   - If the two LSBs are 10, add 0
   - If the two LSBs are 11, add 1
3. Shift the result right by one bit.

4.3.5 Scaling for divide and square-root

On the Hexagon processor, floating point divide and square-root operations are implemented in software using library functions. To enable the efficient implementation of these operations, the processor supports special variants of the multiply-accumulate instruction. These are named *scale FMA*.

Scale FMA supports optional scaling of the product generated by the floating-point fused multiply-add instruction.

Scaling is specified in the instruction by adding the `:scale` specifier and a predicate register operand. For example:

```
R3 += sfmpy(R0,R1,P2):scale
```

For single precision, the scaling factor is two raised to the power specified by the contents of the predicate register (which is treated as an 8-bit two's complement value). For double precision, the predicate register value is doubled before being used as a power of two.

NOTE Scale FMA instructions should not be used outside of divide and square-root library routines. No guarantee is provided that future versions of the Hexagon processor will implement these instructions using the same semantics. Future versions assume only that compatibility for scale FMA is limited to the needs of divide and square-root library routines.

4.4 XTYPE operations

The XTYPE instruction class includes most of the data-processing operations performed by the Hexagon processor. These operations are categorized by their operation type:

- ALU
- Bit manipulation
- Complex
- Floating point
- Multiply
- Permute
- Predicate
- Shift

4.4.1 ALU

ALU operations modify 8-, 16-, 32-, and 64-bit data. These operations include:

- Add and subtract with and without saturation
- Add and subtract with accumulate
- Absolute value
- Logical operations
- Min, max, negate instructions
- Register transfers of 64-bit data
- Word to doubleword sign extension
- Comparisons

For more information see Section 11.3.1 and Section 11.12.1.

4.4.2 Bit manipulation

Bit manipulation operations modify bit fields in a register or register pair. These operations include:

- Bit field insert
- Bit field signed and unsigned extract
- Count leading and trailing bits
- Compare bit masks
- Set / Clear / Toggle bit
- Test bit operation
- Interleave/deinterleave bits
- Bit reverse
- Split bitfield
- Masked parity and Linear Feedback shift
- Table index formation

For more information see Section 11.12.2.

4.4.3 Complex

Complex operations manipulate complex numbers. These operations include:

- Complex add and subtract
- Complex multiply with optional round and pack
- Vector complex multiply
- Vector complex conjugate
- Vector complex rotate
- Vector reduce complex multiply real or imaginary

For more information see Section 11.12.3.

4.4.4 Floating point

Floating-point operations manipulate single-precision floating point numbers. These operations include:

- Addition and subtraction
- Multiplication (with optional scaling)
- Min/max/compare
- Format conversion

The Hexagon floating-point operations are defined to support the IEEE floating-point standard. However, certain IEEE-required operations – such as divide and square root – are not supported directly. Instead, special instructions are defined to support the implementation of the required operations as library routines. These instructions include:

- A special version of the fused multiply-add instruction (designed specifically for use in library routines)
- Reciprocal/square root approximations (which compute the approximate initial values used in reciprocal and reciprocal-square-root routines)
- Extreme value assistance (which adjusts input values if they cannot produce correct results using convergence algorithms)

For more information see Section 11.12.4.

NOTE The special floating-point instructions are not intended for use directly in user code – they should be used only in the floating point library.

Format conversion

The floating-point conversion instructions `sfmake` and `dfmake` convert an unsigned 10-bit immediate value into the corresponding floating-point value.

The immediate value must be encoded so bits [5:0] contain the significand, and bits [9:6] the exponent. The exponent value is added to the initial exponent value (`bias - 6`).

For example, to generate the single-precision floating point value 2.0, bits [5:0] must be set to 0, and bits [9:6] set to 7. Performing `sfmake` on this immediate value yields the floating point value `0x40000000`, which is 2.0.

NOTE The conversion instructions are designed to handle common floating point values, including most integers and many basic fractions (1/2, 3/4, etc.).

Rounding

The Hexagon user status register (Section 2.3.3) includes the FPRND field, which is used to specify the IEEE-defined floating-point rounding mode.

Exceptions

The Hexagon user status register (Section 2.3.3) includes five status fields which work as sticky flags for the five IEEE-defined exception conditions: inexact, overflow, underflow, divide by zero, and invalid. A sticky flag is set when the corresponding exception occurs, and remains set until explicitly cleared.

The user status register also includes five mode fields which are used to specify whether an operating-system trap should be performed if one of the floating-point exceptions occur. For every instruction packet containing a floating-point operation, if a floating-point sticky flag and the corresponding trap-enable bit are both set, then a floating-point trap will be generated. After the packet commits, the Hexagon processor then automatically traps to the operating system.

NOTE Non-floating-point instructions never generate a floating-point trap, regardless of the state of the sticky flag and trap-enable bits.

4.4.5 Multiply

Multiply operations support fixed-point multiplication, including both single- and double-precision multiplication, and polynomial multiplication.

Single precision

In single-precision arithmetic a 16-bit value is multiplied by another 16-bit value. These operands can come from the high portion or low portion of any register. Depending on the instruction, the result of the 16 × 16 operation can optionally be accumulated, saturated, rounded, or shifted left by 0-1 bits.

The instruction set supports operations on signed × signed, unsigned × unsigned, and signed × unsigned data.

Table 4-1 summarizes the options available for 16 × 16 single precision multiplications. The symbols used in the table are as follows:

- SS – Perform signed × signed multiply
- UU – Perform unsigned × unsigned multiply
- SU – Perform signed × unsigned multiply
- A+ – Result added to accumulator
- A- – Result subtracted from accumulator
- 0 – Result not added to accumulator

Table 4-1 Single-precision multiply options

Multiply	Result	Sign	Accumulate	Sat	Rnd	Scale
16 × 16	32	SS	A+, A-	Yes	No	0-1
16 × 16	32	SS	0	Yes	Yes	0-1
16 × 16	64	SS	A+, A-	No	No	0-1
16 × 16	64	SS	0	No	Yes	0-1
16 × 16	32	UU	A+, A-, 0	No	No	0-1
16 × 16	64	UU	A+, A-, 0	No	No	0-1
16 × 16	32	SU	A+, 0	Yes	No	0-1

Double precision

Double precision instructions are available for both 32 × 32 and 32 × 16 multiplication:

- For 32 × 32 multiplication the result can be either 64 or 32 bits. The 32-bit result can be either the high or low portion of the 64-bit product.
- For 32 × 16 multiplication the result is always taken as the upper 32 bits.

The operands can be either signed or unsigned.

Table 4-2 summarizes the options available in double precision multiply.

Table 4-2 Double precision multiply options

Multiply	Result	Sign	Accumulate	Sat	Rnd	Scale
32 × 32	64	SS, UU	A+, A-, 0	No	No	0
32 × 32	32 (upper)	SS, UU	0	No	Yes	0
32 × 32	32 (low)	SS, UU	A+, 0	No	No	0
32 × 16	32 (upper)	SS, UU	A+, 0	Yes	Yes	0-1
32 × 32	32 (upper)	SU	0	No	No	0

Polynomial

Polynomial multiply instructions are available for both words and vector halfwords.

These instructions are useful for many algorithms including scramble code generation, cryptographic algorithms, convolutional, and Reed Solomon code.

For more information on multiply operations see Section 11.12.5.

4.4.6 Permute

Permute operations perform various operations on vector data, including arithmetic, format conversion, and rearrangement of vector elements. Many types of conversions are supported:

- Swizzle bytes
- Vector shuffle
- Vector align
- Vector saturate and pack
- Vector splat bytes
- Vector splice
- Vector sign extend halfwords
- Vector zero extend bytes
- Vector zero extend halfwords
- Scalar saturate to byte, halfword, word
- Vector pack high and low halfwords
- Vector round and pack
- Vector splat halfwords

For more information see Section 11.3.2 and Section 11.12.6.

4.4.7 Predicate

Predicate operations modify predicate source data. The categories of instructions available include:

- Vector mask generation
- Predicate transfers
- Viterbi packing

For more information see Section 11.3.3 and Section 11.12.7.

4.4.8 Shift

Scalar shift operations perform a variety of 32 and 64-bit shifts followed by an optional add/sub or logical operation. Figure 4-4 shows the general operation.

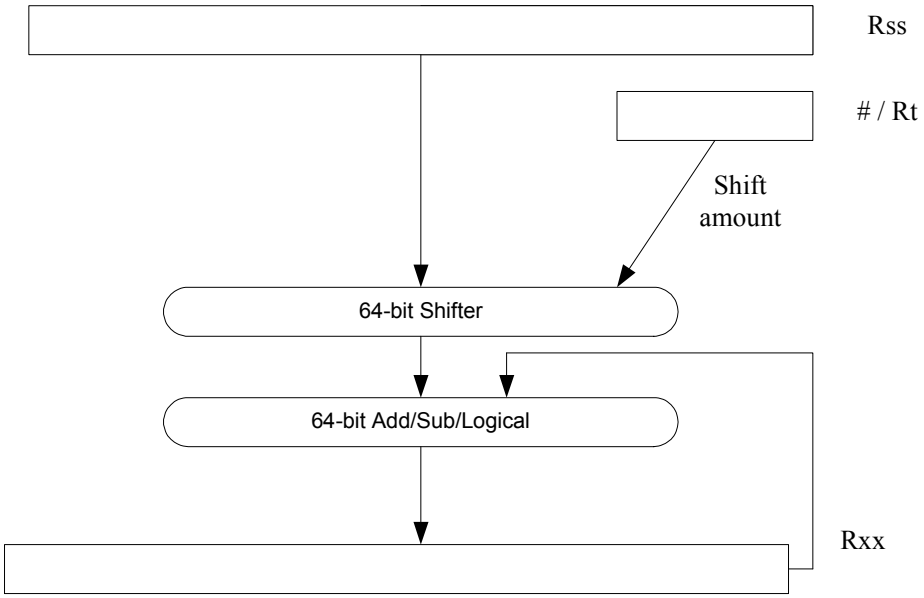


Figure 4-4 64-bit shift and add/sub/logical

Four shift types are supported:

- ASR – Arithmetic shift right
- ASL – Arithmetic shift left
- LSR – Logical shift right
- LSL – Logical shift left

In register-based shifts, the Rt register is a signed two's-complement number. If this value is positive, then the instruction opcode tells the direction of shift (right or left). If this value is negative, then the shift direction indicated by the opcode is reversed.

When arithmetic right shifts are performed, the sign bit is shifted in, whereas logical right shifts shift in zeros. Left shifts always shift in zeros.

Some shifts are available with saturation and rounding options.

For more information see Section 11.12.8.

4.5 ALU32 operations

The ALU32 instruction class includes general arithmetic/logical operations on 32-bit data:

- Add, subtract, negate without saturation on 32-bit data
- Logical operations such as And, Or, Xor, And with immediate, and Or with immediate
- Scalar 32-bit compares
- Combine halfwords, combine words, combine with immediates, shift halfwords, and Mux
- Conditional add, combine, logical, subtract, and transfer.
- NOP
- Sign and zero-extend bytes and halfwords
- Transfer immediates and registers
- Vector add, subtract, and average halfwords

For more information see Section 11.3.

NOTE ALU32 instructions can be executed on any slot (Section 3.4.3).

Chapter 6 describes the conditional execution and compare instructions.

4.6 Vector operations

Vector operations support arithmetic operations on vectors of bytes, halfwords, and words.

The vector operations belong to the XTYPE instruction class (except for vector add, subtract, and average halfwords, which are ALU32).

Vector byte operations

The vector byte operations process packed vectors of signed or unsigned bytes. They include the following operations:

- Vector add and subtract signed or unsigned bytes
- Vector min and max signed or unsigned bytes
- Vector compare signed or unsigned bytes
- Vector average unsigned bytes
- Vector reduce add unsigned bytes
- Vector sum of absolute differences unsigned bytes

Vector halfword operations

The vector halfword operations process packed 16-bit halfwords. They include the following operations:

- Vector add and subtract halfwords
- Vector average halfwords
- Vector compare halfwords
- Vector min and max halfwords
- Vector shift halfwords
- Vector dual multiply
- Vector dual multiply with round and pack
- Vector multiply even halfwords with optional round and pack
- Vector multiply halfwords
- Vector reduce multiply halfwords

For example, Figure 4-5 shows the operation of the vector arithmetic shift right halfword (`vasrh`) instruction. In this instruction, each 16-bit half-word is shifted right by the same amount which is specified in a register or with an immediate value. Because the shift is arithmetic, the bits shifted in are copies of the sign bit.

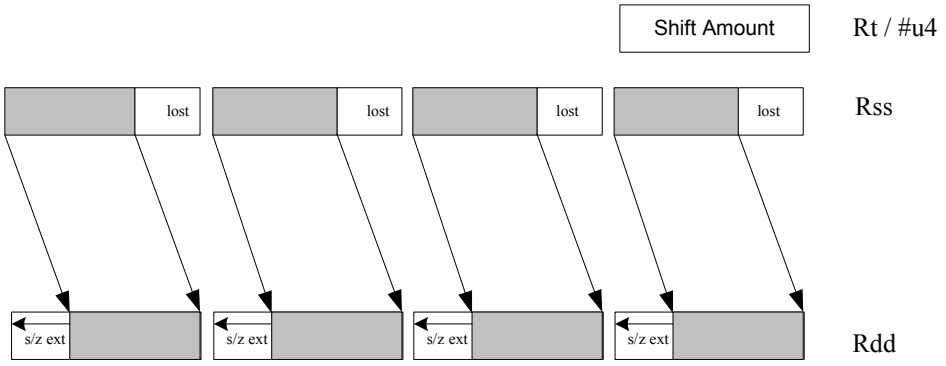


Figure 4-5 Vector halfword shift right

Vector word operations

The vector word operations process packed vectors of two words. They include the following operations:

- Vector add and subtract words
- Vector average words
- Vector compare words
- Vector min and max words
- Vector shift words with optional truncate and pack

For more information on vector operations see Section 11.3.1 and Section 11.12.1.

4.7 CR operations

The CR instruction class includes operations that access the control registers (Section 2.3).

Table 4-3 lists the instructions that access the control registers.

Table 4-3 Control register transfer instructions

Syntax	Operation
`Rd = Cs` `Cd = Rs`	Move control register to / from a general register. NOTE - PC is not a valid destination register.
`Rdd = Css` `Cdd = Rss`	Move control register pair to / from a general register pair. NOTE - PC is not a valid destination register.

NOTE In register-pair transfers, control registers must be specified using their numeric alias names – see Section 2.3 for details.

For more information see Section 11.4.

4.8 Compound operations

The instruction set includes a number of instructions which perform multiple logical or arithmetic operations in a single instruction. They include the following operations:

- And/Or with inverted input
- Compound logical register
- Compound logical predicate
- Compound add-subtract with immediates
- Compound shift-operation with immediates (arithmetic or logical)
- Multiply-add with immediates

For more information see Section 11.12.1.

4.9 Special operations

The instruction set includes a number of special-purpose instructions to support specific applications:

- H.264 CABAC processing
- IP internet checksum
- Software-defined radio

4.9.1 H.264 CABAC processing

H.264/AVC is adopted in a diverse range of multimedia applications:

- HD-DVDs
- HDTV broadcasting
- Internet video streaming

Context Adaptive Binary Arithmetic Coding (CABAC) is one of the two alternative entropy coding methods specified in the H.264 main profile. CABAC offers superior coding efficiency at the expense of greater computational complexity. The Hexagon processor includes a dedicated instruction (`decbin`) to support CABAC decoding.

Binary arithmetic coding is based on the principle of recursive interval subdivision, and its state is characterized by two quantities:

- The current interval range
- The current offset in the current code interval

The offset is read from the encoded bit stream. When decoding a bin, the interval range is subdivided in two intervals based on the estimation of the probability $_{pLPS}$ of LPS: one interval with width of *rLPS = range x pLPS*, and another with width of *rMPS = range x pMPS = range -rLPS*, where LPS stands for Least Probable Symbol, and MPS for Most Probable Symbol.

Depending on which subinterval the offset falls into, the decoder decides whether the bin is decoded as MPS or LPS, after which the two quantities are iteratively updated, as shown in Figure 4-1.

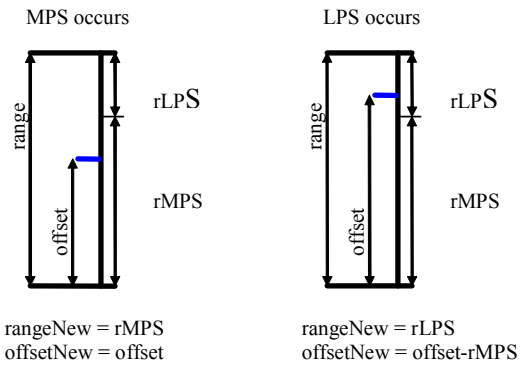


Figure 4-1 Arithmetic decoding for one bin

4.9.1.1 CABAC implementation

In H.264 *range* is a 9-bit quantity, and *offset* is 9-bits in regular mode and 10-bits in bypass mode during the whole decoding process. The calculation of rLPS is approximated by a 64×4 table of 256 bytes, where the range and the context state (selected for the bin to be decoded) are used to address the lookup table. To maintain the precision of the whole decoding process, the new range must be renormalized to ensure that the most significant bit is always 1, and that the offset is synchronously refilled from the bit stream.

To simplify the renormalization/refilling process, the decoding scheme shown in Figure 4-2 was created to significantly reduce the frequency of renormalization and refilling bits from the bit-stream, while also being suitable for DSP implementation.

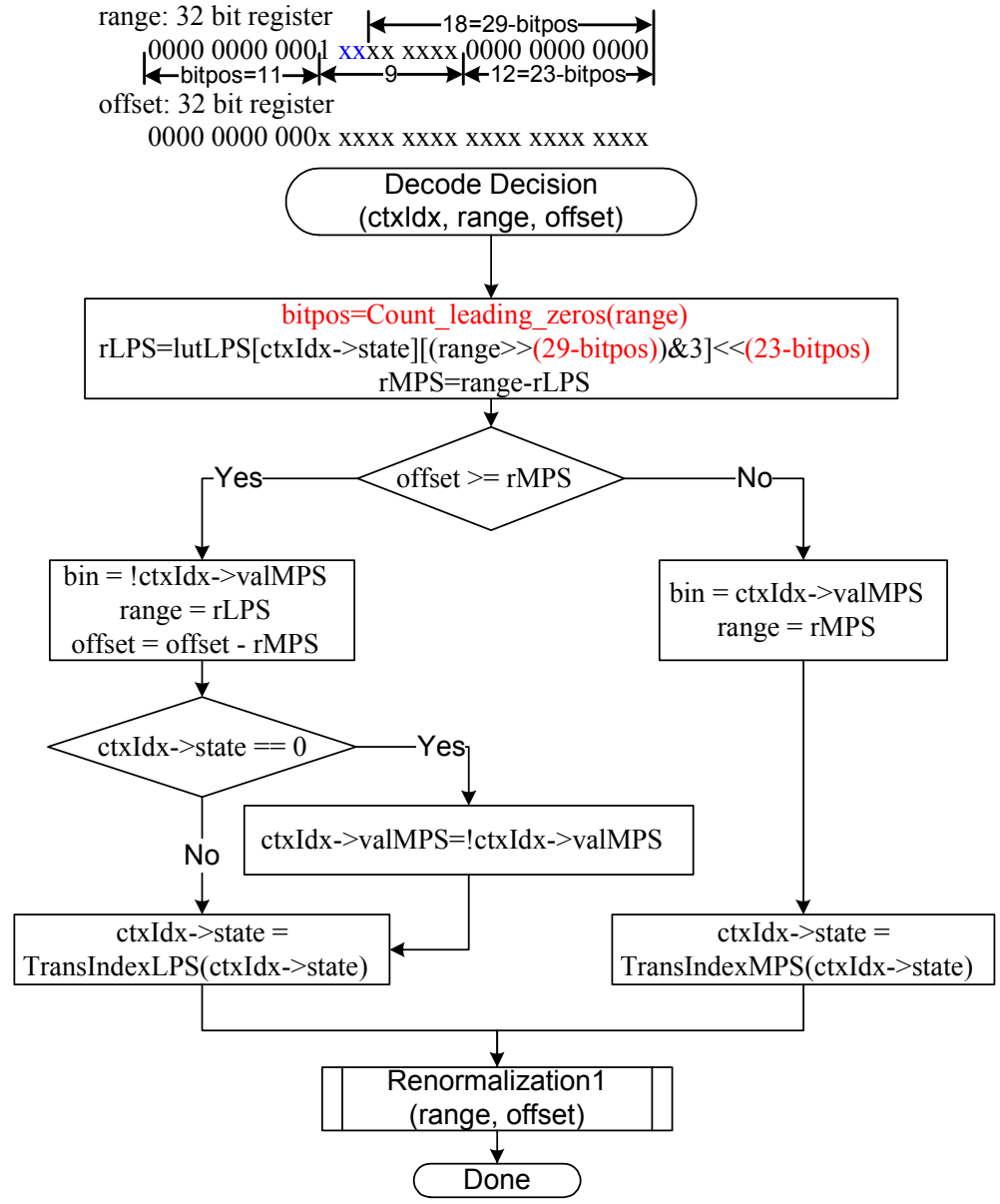


Figure 4-2 CABAC decoding engine for regular bin

By using the `decbin` instruction, the Hexagon processor is able to decode one regular bin in 2 cycles (not counting the bin refilling process).

For more information on the `decbin` instruction see Section 11.12.6.

For example:

```
Rdd = decbin(Rss,Rtt)

INPUT: Rss and Rtt register pairs as:
Rtt.w1[5:0] = state
Rtt.w1[8] = valMPS
Rtt.w0[4:0] =  bitpos
Rss.w0 = range
Rss.w1 = offset

OUTPUT: Rdd register pair is packed as
Rdd.w0[5:0] = state
Rdd.w0[8] = valMPS
Rdd.w0[31:23] = range
Rdd.w0[22:16] = '0'
Rdd.w1 = offset (normalized)

OUTPUT: P0
P0 = (bin)
```

4.9.1.2 Code example

```
H264CabacGetBinNC:
/*****************************************************************
* Non-conventional call:
* Input: R1:0 = offset : range  ,  R2 = dep,   R3 = ctxIdx,
*  R4 = (*ctxIdx), R5 = bitpos
*
* Return:
*       R1: 0 - offset : range
*       P0 - (bin)
*****************************************************************/

// Cycle #1
{ R1:0= decbin(R1:0,R5:4)   // decoding one bin
  R6 = asl(R22,R5)          // where R22 = 0x100
}

// Cycle #2
{ memb(R3) = R0             // save context to *ctxIdx
   R1:0 = vlsrw(R1:0,R5)    // re-align range and offset
   P1 = cmp.gtu(R6,R1)      // need refill? i.e., P1= (range<0x100)
   IF (!P1.new) jumpr:t LR  // return
}
RENORM_REFILL:
...
```

4.9.2 IP internet checksum

The key features of the internet checksum[1] include:

- The checksum can be summed in any order
- Carries may be accumulated using an accumulator larger than size being added, and added back in at any time

Using standard data-processing instructions, the internet checksum can be computed at 8 bytes per cycle in the main loop, by loading words and accumulating into doublewords. After the loop, the upper word is added to the lower word; then the upper halfword is added to the lower halfword, and any carries are added back in.

The Hexagon processor supports a dedicated instruction (`vradduh`) which enables the internet checksum to be computed at a rate of 16 bytes per cycle.

The `vradduh` instruction accepts the halfwords of the two input vectors, adds them all together, and places the result in a 32-bit destination register. This operation can be used for both computing the sum of 16 bytes of input while preserving the carries, and also accumulating carries at the end of computation.

For more information on the `vradduh` instruction see Section 11.12.1.

NOTE This operation utilizes the maximum load bandwidth available in the Hexagon processor.

[1] See RFC 1071 (http://www.faqs.org/rfcs/rfc1071.html)

4.9.2.1 Code example

```
.text
.global fast_ip_check
// Assumes data is 8-byte aligned
// Assumes data is padded at least 16 bytes afterwords with 0's.
// input R0 points to data
// input R1 is length of data
// returns IP checksum in R0

fast_ip_check:
{
   R1 = lsr(R1,#4)          // 16-byte chunks, rounded down, +1
   R9:8 = combine(#0,#0)
   R3:2 = combine(#0,#0)
}
{
   loop0(1f,R1)
   R7:6 = memd(R0+#8)
   R5:4 = memd(R0++#16)
}
.falign
1:
{
   R7:6 = memd(R0+#8)
   R5:4 = memd(R0++#16)
   R2 = vradduh(R5:4,R7:6)    // accumulate 8 halfwords
   R8 = vradduh(R3:2,R9:8)    // accumulate carries
}:endloop0
// drain pipeline
{
   R2 = vradduh(R5:4,R7:6)
   R8 = vradduh(R3:2,R9:8)
   R5:4 = combine(#0,#0)
}
{
   R8 = vradduh(R3:2,R9:8)
   R1 = #0
}
// may have some carries to add back in
{
   R0 = vradduh(R5:4,R9:8)
}
// possible for one more to pop out
{
   R0 = vradduh(R5:4,R1:0)
}
{
   R0 = not(R0)
   jumpr LR
}
```

4.9.3 Software-defined radio

The Hexagon processor includes six special-purpose instructions which support the implementation of software-defined radio. The instructions greatly accelerate the following algorithms:

- Rake despreading
- Scramble code generation
- Polynomial field processing

4.9.3.1 Rake despreading

A fundamental operation in despreading is the PN multiply operation. In this operation the received complex chips are compared against a pseudo-random sequence of QAM constellation points and accumulated.

Figure 4-3 shows the `vrcrotate` instruction which is used to perform this operation. The products are summed to form a soft 32-bit complex symbol. The instruction has both accumulating and non-accumulating versions.

For more information on the `vrcrotate` instruction see Section 11.12.3.

NOTE Using this instruction the Hexagon processor can process 5.3 chips per cycle, and a 12-finger WCDMA user requires only 15 MHz.

xx += vrcrotate(Rss,Rt,#0)

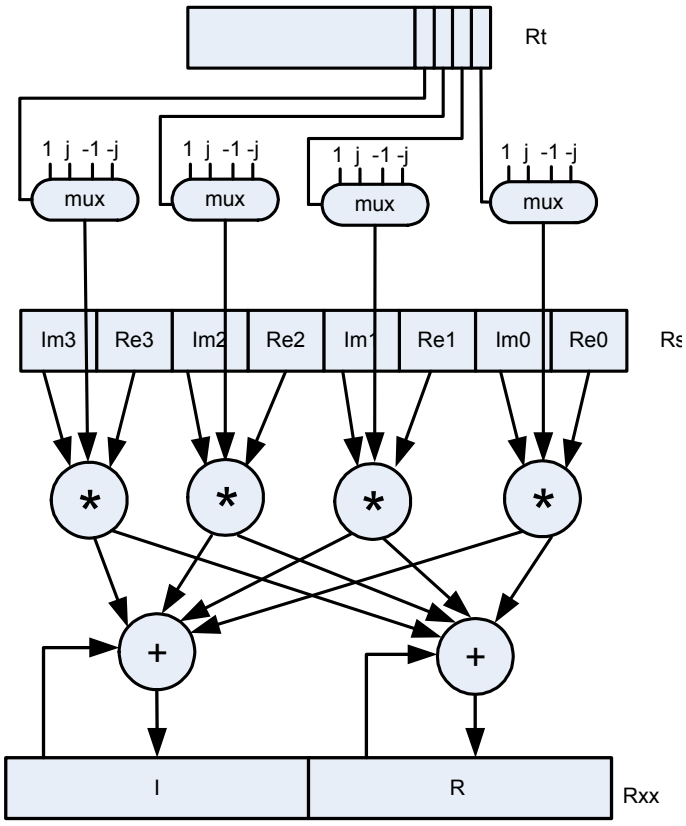


Figure 4-3 Vector reduce complex rotate

4.9.3.2 Polynomial operations

The polynomial multiply instructions support the following operations:

- Scramble code generation (at a rate of 8 symbols per cycle for WCDMA)
- Cryptographic algorithms (such as Elliptic Curve)
- CRC checks (at a rate of 21bits per cycle)
- Convolutional encoding
- Reed Solomon codes

The four versions of this instruction support 32 x 32 and vector 16 x 16 multiplication both with and without accumulation, as shown in Figure 4-4.

For more information on the `pmpy` instructions see Section 11.12.5.

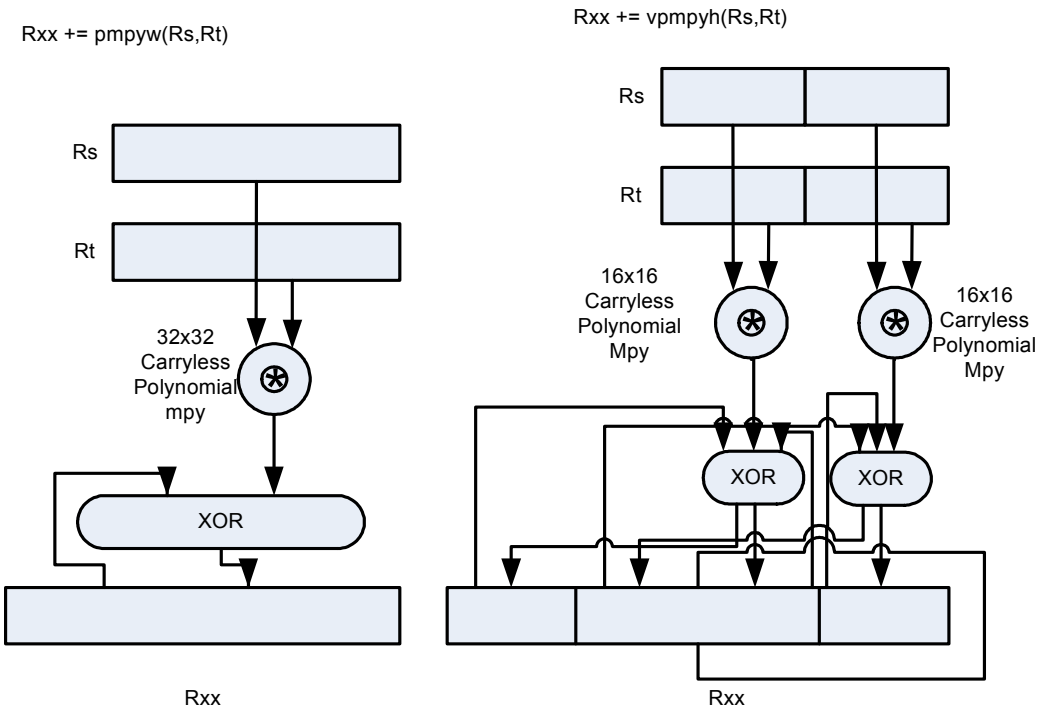


Figure 4-4 Polynomial multiply

5 Memory

5.1 Overview

The Hexagon processor features a load/store architecture, where numeric and logical instructions operate on registers. Explicit load instructions move operands from memory to registers, while store instructions move operands from registers to memory. A small number of instructions (known as *mem-ops*) perform numeric and logical operations directly on memory.

The address space is unified: all accesses target the same linear address space, which contains both instructions and data.

5.2 Memory model

This section describes the memory model for the Hexagon processor.

5.2.1 Address space

The Hexagon processor has a 32-bit byte-addressable memory address space. The entire 4G linear address space is addressable by the user application. A virtual-to-physical address translation mechanism is provided.

5.2.2 Byte order

The Hexagon processor is a little-endian machine: the lowest address byte in memory is held in the least significant byte of a register, as shown in Figure 5-1.

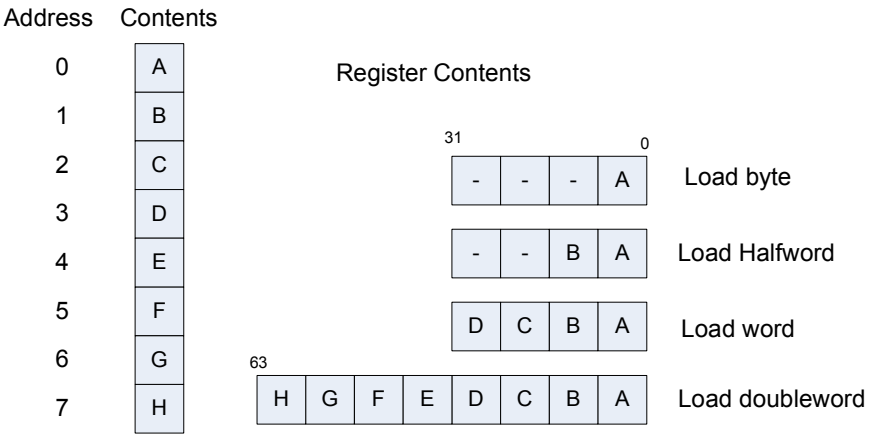


Figure 5-1 Hexagon processor byte order

5.2.3 Alignment

Even though the Hexagon processor memory is byte-addressable, instructions and data must be aligned in memory on specific address boundaries:

- Instructions and instruction packets must be 32-bit aligned
- Data must be aligned to its native access size.

Any unaligned memory access will cause a memory-alignment exception.

The permute instructions (Section 4.4.6) can be used in applications that need to reference unaligned vector data. The loads and stores still must be memory-aligned; however, the permute instructions enable the data to be easily rearranged in registers.

Table 5-1 summarizes the alignment restrictions.

Table 5-1 Memory alignment restrictions

Data Type	Size (bits)	Exception When
Byte Unsigned byte	8	Never
Halfword Unsigned halfword	16	LSB[0] != 0 [1]
Word Unsigned word	32	LSB[1:0] != 00
Doubleword	64	LSB[2:0] != 000
Instruction Instruction packet	32	LSB[1:0] != 00

[1] LSB = Least significant bits of address

5.3 Memory loads

Memory can be loaded in byte, halfword, word, or doubleword sizes. The data types supported are signed or unsigned. The syntax used is `memXX`, where `XX` denotes the data type.

Table 5-2 summarizes the supported load instructions.

Table 5-2 Load instructions

Syntax	Source Size (bits)	Destination Size (bits)	Data Placement	Comment
`Rd = memub(Rs)`	8	32	Low 8 bits	Zero-extend 8 to 32 bits
`Rd = memb(Rs)`	8	32	Low 8 bits	Sign-extend 8 to 32 bits
`Rd = memuh(Rs)`	16	32	Low 16 bits	Zero-extend 16 to 32 bits
`Rd = memh(Rs)`	16	32	Low 16 bits	Sign-extend 16 to 32 bits
`Rd = memubh(Rs)`	16	32	Bytes 0 and 2	Bytes 1 and 3 zeroed [1]
`Rd = membh(Rs)`	16	32	Bytes 0 and 2	Bytes 1 and 3 sign-extended
`Rd = memw(Rs)`	32	32	All 32 bits	Load word
`Rdd = memubh(Rs)`	32	64	Bytes 0,2,4,6	Bytes 1,3,5,7 zeroed
`Rdd = membh(Rs)`	32	64	Bytes 0,2,4,6	Bytes 1,3,5,7 sign-extended
`Rdd = memd(Rs)`	64	64	All 64 bits	Load doubleword
`Ryy = memh_fifo(Rs)`	16	64	High 16 bits	Shift vector and load halfword
`deallocframe`	64	64	All 64 bits	See Chapter 8
`dealloc_return`	64	64	All 64 bits	See Chapter 8

[1] The `memubh` and `membh` instructions load contiguous bytes from memory (either 2 or 4 bytes) and unpack these bytes into a vector of halfwords. The instructions are useful when bytes are used as input into halfword vector operations, which is common in video and image processing..

NOTE The memory load instructions belong to instruction class LD, and can execute only in Slots 0 or 1.

5.4 Memory stores

Memory can be stored in byte, halfword, word, or doubleword sizes. The syntax used is `memX`, where `X` denotes the data type.

Table 5-3 summarizes the supported store instructions.

Table 5-3 Store instructions

Syntax	Source Size (bits)	Destination Size (bits)	Comment
`memb(Rs) = Rt`	32	8	Store byte (bits 7:0)
`memb(Rs) = #s8`	8	8	Store byte
`memh(Rs) = Rt`	32	16	Store lower half (bits 15:0)
`memh(Rs) = Rt.H`	32	16	Store upper half (bits 31:16)
`memh(Rs) = #s8`	8	16	Sign-extend 8 to 16 bits
`memw(Rs) = Rt`	32	32	Store word
`memw(Rs) = #s8`	8	32	Sign-extend 8 to 32 bits
`memd(Rs) = Rtt`	64	64	Store doubleword
`allocframe(#u11)`	64	64	See Chapter 8

NOTE The memory store instructions belong to instruction class ST, and can execute only in slot 0 or – when part of a dual store (Section 5.5) – slot 1.

5.5 Dual stores

Two memory store instructions can appear in the same instruction packet. The resulting operation is considered a *dual store*. For example:

```
{
memw(R5) = R2       // dual store
memh(R6) = R3
}
```

Unlike most packetized operations, dual stores are not executed in parallel (Section 3.4.1). Instead, the store instruction in Slot 1 effectively executes first, followed by the store instruction in Slot 0.

NOTE The store instructions in a dual store must belong to instruction class ST (Section 5.4), and can execute only in Slots 0 and 1.

5.6 New-value stores

A memory store instruction can store a register that is assigned a new value in the same instruction packet (Section 3.4). This feature is expressed in assembly language by appending the suffix ".new" to the source register. For example:

```
{
R2 = memh(R4+#8)        // load halfword
memw(R5) = R2.new       // store newly-loaded value
}
```

New-value store instructions have the following restrictions:

- If an instruction uses auto-increment or absolute-set addressing mode (Section 5.8), its address register cannot be used as the new-value register.
- If an instruction produces a 64-bit result, its result registers cannot be used as the new-value register.
- If the instruction that sets a new-value register is conditional (Section 6.2.2), it must always be executed.

NOTE The new-value store instructions belong to instruction class NV, and can execute only in Slot 0.

5.7 Mem-ops

Mem-ops perform basic arithmetic, logical, and bit operations directly on memory operands, without the need for a separate load or store. Mem-ops can be performed on byte, halfword, or word sizes. Table 5-4 lists the mem-ops.

Table 5-4 Mem-ops

Syntax	Operation
`memXX(Rs+#u6) [+-|&] = Rt`	Arithmetic/logical on memory
`memXX(Rs+#u6) [+-] = #u5`	Arithmetic on memory
`memXX(Rs+#u6) = clrbit(#u5)`	Clear bit in memory
`memXX(Rs+#u6) = setbit(#u5)`	Set bit in memory

NOTE The mem-op instructions belong to instruction class MEMOP, and can execute only in Slot 0.

5.8 Addressing modes

Table 5-5 summarizes the supported addressing modes.

Table 5-5 Addressing modes

Mode	Syntax	Operation [1]
Absolute	`memXX(##address)`	`EA = address`
Absolute-set	`memXX(Re=##address)`	`EA = address` `Re = address`
Absolute with register offset	`memXX(Ru<<#u2+##U32)`	`EA = imm + (Ru << #u2)`
Global-pointer-relative	`memXX(GP+#immediate)` `memXX(#immediate)`	`EA = GP + immediate`
Indirect	`memXX(Rs)`	`EA = Rs`
Indirect with offset	`memXX(Rs+#s11)`	`EA = Rs + imm`
Indirect with register offset	`memXX(Rs+Ru<<#u2)`	`EA = Rs + (Ru << #u2)`
Indirect with auto-increment immediate	`memXX(Rx++#s4)`	`EA = Rx;` `Rx += (imm)`
Indirect with auto-increment register	`memXX(Rx++Mu)`	`EA = Rx;` `Rx += Mu`
Circular with auto-increment immediate	`memXX(Rx++#s4:circ(Mu))`	`EA = Rx;` `Rx = circ_add(Rx,imm,Mu)`
Circular with auto-increment register	`memXX(Rx++I:circ(Mu))`	`EA = Rx;` `Rx = circ_add(Rx,I,Mu)`
Bit-reversed with auto-increment register	`memXX(Rx++Mu:brev)`	`EA = Rx.H + bit_reverse(Rx.L)` `Rx += Mu`

[1] EA (Effective Address) is equivalent to VA (Virtual Address).

5.8.1 Absolute

The absolute addressing mode uses a 32-bit constant value as the effective memory address. For example:

```
R2 = memw(##100000)    // load R2 with word from addr 100000
memw(##200000) = R4    // store R4 to word at addr 200000
```

5.8.2 Absolute-set

The absolute-set addressing mode assigns a 32-bit constant value to the specified general register, then uses the assigned value as the effective memory address. For example:

```
R2 = memw(R1=##400000)    // load R2 with word from addr 400000
                          //  and load R1 with value 400000
memw(R3=##600000) = R4    // store R4 to word at addr 600000
                          //  and load R3 with value 600000
```

5.8.3 Absolute with register offset

The absolute with register offset addressing mode performs an arithmetic left shift of a 32-bit general register value by the amount specified in a 2-bit unsigned immediate value, and then adds the shifted result to an unsigned 32-bit constant value to create the 32-bit effective memory address. For example:

```
R2 = memh(R3 << #3 + ##100000) // load R2 with signed halfword
                               // from addr [100000 + (R3 << 3)]
```

The 32-bit constant value is the base address, and the shifted result is the byte offset.

NOTE This addressing mode is useful for loading an element from a global table, where the immediate value is the name of the table, and the register holds the index of the element.

5.8.4 Global pointer relative

The global pointer relative addressing mode adds an unsigned offset value to the Hexagon processor global data pointer `GP` to create the 32-bit effective memory address. This addressing mode is used to access global and static data in C.

Global pointer relative addresses can be expressed two ways in assembly language:

- By explicitly adding an unsigned offset value to register `GP`
- By specifying only an immediate value as the instruction operand

For example:

```
R2 = memh(GP+#100)      // load R2 with signed halfword
                        // from [GP + 100 bytes]

R3 = memh(#2000)        // load R3 with signed halfword
                        // from [GP + #2000 - _SDA_BASE]
```

Specifying only an immediate value causes the assembler and linker to automatically subtract the value of the special symbol `_SDA_BASE_` from the immediate value, and use the result as the effective offset from `GP`.

The global data pointer is programmed in the GDP field of register `GP` (Section 2.3.8). This field contains an unsigned 26-bit value which specifies the most significant 26 bits of the 32-bit global data pointer. (The least significant 6 bits of the pointer are defined to always be zero.)

The memory area referenced by the global data pointer is known as the *global data area*. It can be up to 512 KB in length, and – because of the way the global data pointer is defined – must be aligned to a 64-byte boundary in virtual memory.

When expressed in assembly language, the offset values used in global pointer relative addressing always specify byte offsets from the global data pointer. Note that the offsets must be integral multiples of the size of the instruction data type.

Table 5-6 lists the offset ranges for global pointer relative addressing.

Table 5-6 Offset ranges (Global pointer relative)

Data Type	Offset Range	Offset Must Be Multiple Of
doubleword	0 ... 524280	8
word	0 ... 262140	4
halfword	0 ... 131070	2
byte	0 ... 65535	1

NOTE When using global pointer relative addressing, the immediate operand should be a symbol in the `.sdata` or `.sbss` section to ensure that the offset is valid.

5.8.5 Indirect

The indirect addressing mode uses a 32-bit value stored in a general register as the effective memory address. For example:

```
R2 = memub(R1)   // load R2 with unsigned byte from addr R1
```

5.8.6 Indirect with offset

The indirect with offset addressing mode adds a signed offset value to a general register value to create the 32-bit effective memory address. For example:

```
R2 = memh(R3 + #100)   // load R2 with signed halfword
                       // from [R3 + 100 bytes]
```

When expressed in assembly language, the offset values always specify byte offsets from the general register value. Note that the offsets must be integral multiples of the size of the instruction data type.

Table 5-7 lists the offset ranges for indirect with offset addressing.

Table 5-7 Offset ranges (Indirect with offset)

Data Type	Offset Range	Offset Must Be Multiple Of
doubleword	-8192 ... 8184	8
word	-4096 ... 4092	4
halfword	-2048 ... 2046	2
byte	-1024 ... 1023	1

NOTE The offset range is smaller for conditional instructions (Section 5.9).

5.8.7 Indirect with register offset

The indirect with register offset addressing mode adds a 32-bit general register value to the result created by performing an arithmetic left shift of a second 32-bit general register value by the amount specified in a 2-bit unsigned immediate value, forming the 32-bit effective memory address. For example:

```
R2 = memh(R3+R4<<#1)    // load R2 with signed halfword
                        // from [R3 + (R4 << 1)]
```

The register values always specify byte addresses.

5.8.8 Indirect with auto-increment immediate

The indirect with auto-increment immediate addressing mode uses a 32-bit value stored in a general register to specify the effective memory address. However, after the address is accessed, a signed value (known as the *increment*) is added to the register so it specifies a different memory address (which will be accessed in a subsequent instruction). For example:

```
R2 = memw(R3++#4)    // R3 contains the effective address
                     // R3 is then incremented by 4
```

When expressed in assembly language, the increment values always specify byte offsets from the general register value. Note that the offsets must be integral multiples of the size of the instruction data type.

Table 5-8 lists the increment ranges for indirect with auto-increment immediate addressing.

Table 5-8 Increment ranges (Indirect with auto-inc immediate)

Data Type	Increment Range	Increment Must Be Multiple Of
doubleword	-64 ... 56	8
word	-32 ... 28	4
halfword	-16 ... 14	2
byte	-8 ... 7	1

5.8.9 Indirect with auto-increment register

The indirect with auto-increment register addressing mode is functionally equivalent to indirect with auto-increment immediate, but uses a modifier register `Mx` (Section 2.3.4) instead of an immediate value to hold the increment. For example:

```
R2 = memw(R0++M1)    // The effective addr is the value of R0.
                     // Next, M1 is added to R0 and the result
                     // is stored in R0.
```

When auto-incrementing with a modifier register, the increment is a signed 32-bit value which is added to the general register. This offers two advantages over auto-increment immediate:

- A larger increment range
- Variable increments (since the modifier register can be programmed at runtime)

The increment value always specifies a byte offset from the general register value.

NOTE The signed 32-bit increment range is identical for all instruction data types (doubleword, word, halfword, byte).

5.8.10 Circular with auto-increment immediate

The circular with auto-increment immediate addressing mode is a variant of indirect with auto-increment addressing – it accesses data buffers in a modulo wrap-around fashion. Circular addressing is commonly used in data stream processing.

Circular addressing is expressed in assembly language with the address modifier "`:circ(Mx)`", where `Mx` specifies a modifier register which is programmed to specify the circular buffer (Section 2.3.4). For example:

```
R0 = memb(R2++#4:circ(M0))    // load from R2 in circ buf specified
                              //  by M0
memw(R2++#8:circ(M1)) = R0    // store to R2 in circ buf specified
                              //  by M1
```

Circular addressing is set up by programming the following elements:

- The Length field of the `Mx` register is set to the length (in bytes) of the circular buffer to be accessed. A circular buffer can be from 4 to (128K-1) bytes long.
- The K field of the `Mx` register is always set to 0.
- The circular start register `CSx` that corresponds to `Mx` (`CS0` for `M0`, `CS1` for `M1`) is set to the start address of the circular buffer.

In circular addressing, after memory is accessed at the address specified in the general register, the general register is incremented by the immediate increment value and then modulo'ed by the circular buffer length to implement wrap-around access of the buffer.

When expressed in assembly language, the increment values always specify byte offsets from the general register value. Note that the offsets must be integral multiples of the size of the instruction data type.

Table 5-9 lists the increment ranges for circular with auto-increment immediate addressing.

Table 5-9 Increment ranges (Circular with auto-inc immediate)

Data Type	Increment Range	Increment Must Be Multiple Of
doubleword	-64 ... 56	8
word	-32 ... 28	4
halfword	-16 ... 14	2
byte	-8 ... 7	1

When programming a circular buffer the following rules apply:

- The start address must be aligned to the native access size of the buffer elements.
- ABS(Increment) < Length. The absolute value of the increment must be less than the buffer length.
- Access size < (Length-1). The memory access size (1 for byte, 2 for halfword, 4 for word, 8 for doubleword) must be less than (Length-1).
- Buffers must not wrap around in the 32-bit address space.

NOTE If any of these rules are not followed the execution result is undefined.

For example, a 150-byte circular buffer can be set up and accessed as follows:

```
R4.H = #0                      // K = 0
R4.L = #150                    // length = 150
M0 = R4
R2 = ##cbuf                    // start addr = cbuf
CS0 = R2
R0 = memb(R2++#4:circ(M0))     // Load byte from circ buf
                               // specified by M0/CS0
                               // inc R2 by 4 after load
                               // wrap R2 around if >= 150
```

The following C function precisely describes the behavior of the circular add function:

```
unsigned int
fcircadd(unsigned int pointer, int offset,
 unsigned int M_reg, unsigned int CS_reg)
{
unsigned int length;
int new_pointer, start_addr, end_addr;

length = (M_reg&0x01ffff);  // lower 17-bits gives buffer size
new_pointer = pointer+offset;
start_addr = CS_reg;
end_addr = CS_reg + lenth;
if (new_pointer >= end_addr) {
   new_pointer -= length;
} else if (new_pointer < start_addr) {
   new_pointer += length;
}
return (new_pointer);
}
```

5.8.11 Circular with auto-increment register

The circular with auto-increment register addressing mode is functionally equivalent to circular with auto-increment immediate, but uses a register instead of an immediate value to hold the increment.

Register increments are specified in circular addressing instructions by using the symbol `I` as the increment (instead of an immediate value). For example:

```
R0 = memw(R2++I:circ(M1))     // load byte with incr of I*4 from
                              // circ buf specified by M1/CS1
```

When auto-incrementing with a register, the increment is a signed 11-bit value which is added to the general register. This offers two advantages over circular addressing with immediate increments:

- Larger increment ranges
- Variable increments (since the increment register can be programmed at runtime)

The circular register increment value is programmed in the `I` field of the modifier register `Mx` (Section 2.3.4) as part of setting up the circular data access. This register field holds the signed 11-bit increment value.

Increment values are expressed in units of the buffer element data type, and are automatically scaled at runtime to the proper data access size.

Table 5-10 lists the increment ranges for circular with auto-increment register addressing.

Table 5-10 Increment ranges (Circular with auto-inc register)

Data Type	Increment Range	Increment Must Be Multiple Of
doubleword	-8192 ... 8184	8
word	-4096 ... 4092	4
halfword	-2048 ... 2046	2
byte	-1024 ... 1023	1

When programming a circular buffer (with either a register or immediate increment), all the rules that apply to circular addressing must be followed – for details see Section 5.8.10.

NOTE If any of these rules are not followed the execution result is undefined.

5.8.12 Bit-reversed with auto-increment register

The bit-reversed with auto-increment register addressing mode is a variant of indirect with auto-increment addressing – it accesses data buffers using an address value which is the bit-wise reversal of the value stored in the general register. Bit-reversed addressing is used in fast Fourier transforms (FFT) and Viterbi encoding.

The bit-wise reversal of a 32-bit address value is defined as follows:

- The lower 16 bits are transformed by exchanging bit 0 with bit 15, bit 1 with bit 14, and so on.
- The upper 16 bits remain unchanged.

Bit-reversed addressing is expressed in assembly language with the address modifier "`:brev`". For example:

```
R2 = memub(R0++M1:brev)  // The address is (R0.H | bitrev(R0.L))
                         // The orginal R0 (not reversed) is added
                         // to M1 and written back to R0
```

The initial values for the address and increment must be set in bit-reversed form, with the hardware bit-reversing the bit-reversed address value to form the effective address.

The buffer length for a bit-reversed buffer must be an integral power of 2, with a maximum length of 64K bytes.

To support bit-reversed addressing, buffers must be properly aligned in memory. A bit-reversed buffer is properly aligned when its starting byte address is aligned to a power of 2 greater than or equal to the buffer size (in bytes). For example:

```
int bitrev_buf[256] __attribute__((aligned(1024)));
```

The bit-reversed buffer declared above is aligned to 1024 bytes because the buffer size is 1024 bytes (256 integer words × 4 bytes), and 1024 is an integral power of 2.

The buffer location pointer for a bit-reversed buffer must be initialized so the least-significant 16 bits of the address value are bit-reversed.

The increment value must be initialized to the following value:

```
bitreverse(buffer_size_in_bytes / 2)
```

...where `bitreverse` is defined as bit-reversing the least-significant 16 bits while leaving the remaining bits unchanged.

NOTE To simplify the initialization of the bit-reversed pointer, bit-reversed buffers can be aligned to a 64K byte boundary. This has the advantage of allowing the bit-reversed pointer to be initialized to the base address of the bit-reversed buffer, with no bit-reversing required for the least-significant 16 bits of the pointer value (which are all set to 0 by the 64K alignment).

Since buffers allocated on the stack only have an alignment of 8 bytes or less, in most cases bit-reversed buffers should not be declared on the stack.

After a bit-reversed memory access is completed, the general register is incremented by the register increment value. Note that the value in the general register is never affected by the bit-reversal that is performed as part of the memory access.

NOTE The Hexagon processor supports only register increments for bit-reversed addressing – it does not support immediate increments.

5.9 Conditional load/stores

Some load and store instructions can be executed conditionally based on predicate values which were set in a previous instruction. The compiler generates conditional loads and stores to increase instruction-level parallelism.

Conditional loads and stores are expressed in assembly language with the instruction prefix "`if (`*`pred_expr`*`)`", where *`pred_expr`* specifies a predicate register expression (Section 6.2). For example:

```
if (P0) R0 = memw(R2)              // conditional load
if (!P2) memh(R3 + #100) = R1      // conditional store
if (P1.new) R3 = memw(R3++#4)      // conditional load
```

Not all addressing modes are supported in conditional loads and stores. Table 5-11 shows which modes are supported.

Table 5-11 Addressing modes (Conditional load/store)

Addressing Mode	Conditional
Absolute	Yes
Absolute-set	No
Absolute with register offset	No
Global pointer relative	No
Indirect	Yes
Indirect with offset	Yes
Indirect with register offset	Yes
Indirect with auto-increment immediate	Yes
Indirect with auto-increment register	No
Circular with auto-increment immediate	No
Circular with auto-increment register	No
Bit-reversed with auto-increment register	No

When a conditional load or store instruction uses indirect-with-offset addressing mode, note that the offset range is smaller than the range normally defined for indirect-with-offset addressing (Section 5.8.6).

Table 5-12 lists the conditional and normal offset ranges for indirect-with-offset addressing.

Table 5-12 Conditional offset ranges (Indirect with offset)

Data Type	Offset Range (Conditional)	Offset Range (Normal)	Offset Must Be Multiple Of
doubleword	0 ... 504	-8192 ... 8184	8
word	0 ... 252	-4096 ... 4092	4
halfword	0 ... 126	-2048 ... 2046	2
byte	0 ... 63	-1024 ... 1023	1

NOTE For more information on conditional execution see Chapter 6.

5.10 Cache memory

The Hexagon processor has a cache-based memory architecture:

- A level 1 *instruction cache* holds recently-fetched instructions.
- A level 1 *data cache* holds recently-accessed data memory.

Load/store operations that access memory through the level 1 caches are referred to as *cached accesses.*

Load/stores that bypass the level 1 caches are referred to as *uncached accesses*.

Specific memory areas can be configured so they perform cached or uncached accesses. This configuration is performed by the Hexagon processor's memory management unit (MMU). The operating system is responsible for programming the MMU.

Two types of caching are supported (as cache modes):

- W*rite-through caching* keep the cache data consistent with external memory by always writing to the memory any data that is stored in the cache.
- W*rite-back caching* allows data to be stored in the cache without being immediately written to external memory. Cached data that is inconsistent with external memory is referred to as *dirty.*

The Hexagon processor includes dedicated cache maintenance instructions which can be used to push dirty data out to external memory.

Table 5-13 lists the cache sizes for the V6x processor versions.

Table 5-13 Hexagon processor cache size

Processor version	L1 instruction cache	L1 data cache	L2 cache
V60	16KB	32KB	256KB
V61	32KB	32KB	768KB [1]

[1] 512K L2$ + 256K TCM

5.10.1 Uncached memory

In some cases load/store operations need to bypass the cache memories and be serviced externally (for example, when accessing memory-mapped I/O, registers, and peripheral devices, or other system defined entities). The operating system is responsible for configuring the MMU to generate uncached memory accesses.

Uncached memory is categorized into two distinct types:

- *Device-type* is for accessing memory that has side-effects (such as a memory-mapped FIFO peripheral). The hardware ensures that interrupts do not cancel a pending device access. The hardware does not re-order device accesses. Peripheral control registers should be marked as device-type.
- *Uncached-type* is for memory-like memory. No side effects are associated with an access. The hardware can load from uncached memory multiple times. The hardware can re-order uncached accesses.

For instruction accesses, device-type memory is functionally identical to uncached-type memory. For data accesses, they are different.

Code can be executed directly from the L2 cache, bypassing the L1 cache.

5.10.2 Tightly coupled memory

The Hexagon processor supports tightly-coupled instruction memory at Level 1, which is defined as memory with similar access properties to the instruction cache.

Tightly-coupled memory is also supported at level 2, which is defined as backing store to the primary caches.

For more information see Chapter 9.

5.10.3 Cache maintenance operations

The Hexagon processor includes dedicated cache maintenance instructions which can be used to invalidate cache data or push dirty data out to external memory.

The cache maintenance instructions operate on specific memory addresses. If the instruction causes an address error (due to a privilege violation), the processor raises an exception.

NOTE The exception to this rule is `dcfetch`, which never causes a processor exception.

Whenever maintenance operations are performed on the instruction cache, the `isync` instruction (Section 5.11) must be executed immediately afterwards. This instruction ensures that the maintenance operations will be observed by subsequent instructions.

Table 5-14 lists the cache maintenance instructions.

Table 5-14 Cache instructions (User-level)

Syntax	Permitted In Packet	Operation
`icinva(Rs)`	Solo [1]	Instruction cache invalidate. Look up instruction cache at address Rs. If address is in cache, invalidate it.
`dccleaninva(Rs)`	Slot 1 empty or ALU32 only	Data cache clean and invalidate. Look up data cache at address Rs. If address is in cache and has dirty data, flush that data out to memory. The cache line is then invalidated, whether or not dirty data was written.
`dccleana(Rs)`	Slot 1 empty or ALU32 only	Data cache clean. Look up data cache at address Rs. If address is in cache and has dirty data, flush that data out to memory.
`dcinva(Rs)`	Slot 1 empty or ALU32 only	Equivalent to `dccleaninva(Rs)`.
`dcfetch(Rs)`	Normal [2]	Data cache prefetch. Prefetch data at address Rs into data cache. NOTE - This instruction will not cause an exception.
`l2fetch(Rs,Rt)`	ALU32 or XTYPE only	L2 cache prefetch. Prefetch data from memory specified by Rs and Rt into L2 cache.

[1] *Solo* means that the instruction must not be grouped with other instructions in a packet.
[2] *Normal* means that the normal instruction-grouping constraints apply.

5.10.4 L2 cache operations

The cache maintenance operations (Section 5.10.3) operate on both the L1 and L2 caches.

The data cache coherency operations (including clean, invalidate, and clean and invalidate) affect both the L1 and L2 caches, and ensure that the memory hierarchy remains coherent.

However, the instruction cache invalidate operation affects only the L1 cache. Therefore, invalidating instructions that may be in the L1 or L2 caches requires a two-step procedure:

1. Use `icinva` to invalidate instructions from the L1 cache.
2. Use `dcinva` separately to invalidate instructions from the L2 cache.

5.10.5 Cache line zero

The Hexagon processor includes the instruction `dczeroa`. This instruction clears the specified 32-byte block in memory (by storing zero to all 32 bytes).

The detailed instruction behavior is as follows:

- If the specified cache line exists in the L1 or L2 cache, it is cleared and marked dirty.
- If the line does not exist, it is allocated for any cache operating in write-back mode (Section 5.10).
- If the cache is configured as write-through, the cleared cache line is written to the next level of memory.
- If the cache is configured as write-back, no memory write occurs. Instead, the cleared cache line is marked as dirty.

This instruction is useful in optimizing write-only data. It allows for the use of write-back pages – which are the most power and performance efficient – without the need to initially fetch the line to be written. This removes unnecessary read bandwidth and latency.

NOTE The memory address passed to `dczeroa` must be 32-byte aligned. If it is unaligned, the processor will raise an unaligned error exception.

`dczeroa` has the same exception behavior as write-back stores.

A packet with `dczeroa` must have Slot 1 either empty or containing an ALU32 instruction.

5.10.6 Cache prefetch

The Hexagon processor supports the following types of cache prefetching:

- Hardware-based instruction cache prefetching
- Software-based data cache prefetching
- Software-based L2FETCH
- Hardware-based data cache prefetching

Hardware-based instruction cache prefetching

L1 and L2 instruction cache prefetching can be enabled or disabled on a per-thread basis – this is done by setting the HFI field in the user status register (Section 2.3.3).

Software-based data cache prefetching

The Hexagon processor includes the instruction `dcfetch`. This instruction queries the L1 data cache based on the address specified in the instruction:

- If the address is present in the cache, no action is taken.
- If the cache line for the address is missing, the processor attempts to fill the cache line from the next level of memory. Note that the thread does not stall, but rather continues executing while the cache line fill occurs in the background.
- If the address is invalid, no exception is generated and the `dcfetch` instruction is treated as a NOP.

Software-based L2FETCH

More powerful L2 prefetching – of data or instructions – is provided by the `l2fetch` instruction, which specifies an area of memory that is prefetched by the Hexagon processor's hardware prefetch engine. `l2fetch` specifies two registers (Rs and Rt) as operands. Rs contains the 32-bit virtual start address of the memory area to be prefetched. Rt contains three bit fields which further specify the memory area:

- Rt[15:8] – `Width`, which specifies the width (in bytes) of a block of memory to fetch.
- Rt[7:0] – `Height`, which specifies the number of `Width`-sized blocks to fetch.
- Rt[31:16] – `Stride`, which specifies an unsigned byte offset that is used to increment the pointer after each `Width`-sized block is fetched.

The `l2fetch` instruction is non-blocking: it initiates a prefetch operation which is performed in the background by the prefetch engine while the thread continues to execute Hexagon processor instructions.

The prefetch engine requests all lines in the specified memory area. If the line(s) of interest are already resident in the L2 cache, the prefetch engine performs no action. If the lines are not in the L2 cache, the prefetch engine attempts to fetch them.

The prefetch engine makes a best effort to prefetch the requested data, and attempts to perform prefetching at a lower priority than demand fetches. This prevents the prefetch engine from adding bus traffic when the system is under a heavy load.

If a program executes an `l2fetch` instruction while the prefetch operation from a previous `l2fetch` is still active, the prefetch engine halts the current prefetch operation.

NOTE Executing `l2fetch` with any bit field operand programmed to zero will cancel all prefetch activity.

The status of the current prefetch operation is maintained in the PFA field of the user status register (Section 2.3.3). This field can be used to determine whether or not a prefetch operation has completed.

With respect to MMU permissions and error checking, the `l2fetch` instruction behaves similarly to a load instruction. If the virtual address causes a processor exception, the exception will be taken. (Note that this differs from the `dcfetch` instruction, which is treated as a NOP in the presence of a translation/protection error.)

NOTE Prefetches are dropped when the generated prefetch address resides on a different page than the start address. The programmer must use sufficiently large pages to ensure this does not occur.

Figure 5-2 shows two examples of using the `l2fetch` instruction. The first shows a 'box' prefetch, where a 2-D range of memory is defined within a larger frame. The second example shows a prefetch for a large linear memory area of size (`Lines` * 128).

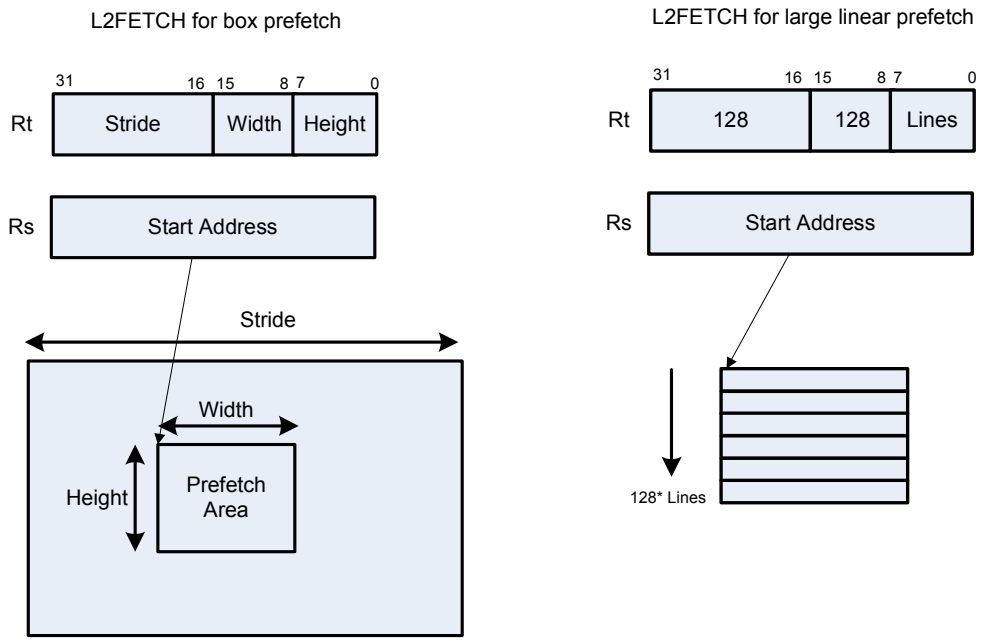


Figure 5-2 L2FETCH instruction

Hardware-based data cache prefetching

L1 data cache prefetching can be enabled or disabled on a per-thread basis – this is done by setting the HFD field in the user status register (Section 2.3.3).

When data cache prefetching is enabled, the Hexagon processor observes patterns of data cache misses, and attempts to predict future misses based on any recurring patterns of misses where the addresses are separated by a constant stride. If such patterns are found, the processor attempts to automatically prefetch future cache lines.

Data cache prefetching can be user-enabled at four levels of aggressiveness:

- HFD = 00: No prefetching
- HFD = 01: Prefetch up to 4 lines for misses originating from a load, with a post-update addressing mode that occurs within a hardware loop
- HFD = 10: Prefetch up to 4 lines for misses originating from loads that occur within a hardware loop
- HFD = 11: Prefetch up to 8 lines for misses originating from loads

5.11 Memory ordering

Some devices may require synchronization of stores and loads when they are accessed. In this case a set of processor instructions enable programmer control of the synchronization and ordering of memory accesses.

Table 5-15 lists the memory-ordering instructions.

Table 5-15 Memory ordering instructions

Syntax	Operation
`isync`	Instruction synchronize. This instruction should be executed after any instruction cache maintenance operation.
`syncht`	Synchronize transactions. Perform "heavyweight" synchronization. Ensure that all previous program transactions (e.g., memw_locked, cached and uncached load/store) have completed before execution resumes past this instruction. NOTE - In V61 syncht ensures that outstanding memory operations from all threads are complete before the syncht instruction is committed.
`barrier`	Set memory barrier. Ensure proper ordering between the program accesses performed before the instruction and those performed after the instruction. All accesses before the barrier will be globally observable before any access occurring after the barrier can be observed. NOTE - In V61 barrier ensures that all outstanding memory operations from the thread executing the barrier are complete before the instruction is committed.

Data memory accesses and program memory accesses are treated separately and held in separate caches. Software should ensure coherency between data and program code if necessary.

For example, with generated or self-modified code, the modified code will be placed in the data cache and may be inconsistent with program cache. The software must explicitly force modified data cache lines to memory (either by using a write-through policy, or through explicit cache clean instructions). A `barrier` instruction should then be used to ensure completion of the stores. Finally, relevant instruction cache contents should be invalidated so the new instructions can be re-fetched.

Here is the recommended code sequence to change and then execute an instruction:

```
ICINVA(R1)          // clear code from instruction cache
ISYNC               // ensure that ICINVA is finished
MEMW(R1)=R0         // write the new instruction
DCCLEANINVA(R1)     // force data out of data cache
SYNCHT              // ensure that it's in memory
JUMPR R1            // can now execute code at R1
```

NOTE The memory-ordering instructions must not be grouped with other instructions in a packet, otherwise the behavior is undefined.

This code sequence differs from the one used in previous processor versions.

5.12 Atomic operations

The Hexagon processor includes an LL/SC (Load Locked / Store Conditional) mechanism to provide the atomic read-modify-write operation that is necessary to implement synchronization primitives such as semaphores and mutexes.

These primitives are used to synchronize the execution of different software programs running concurrently on the Hexagon processor. They can also be used to provide atomic memory support between the Hexagon processor and external blocks.

Table 5-16 describes the atomic instructions.

Table 5-16 Atomic instructions

Syntax	Description
`Rd = memw_locked(Rs)`	Load locked word. Reserve lock on word at address Rs.
`memw_locked(Rs,Pd) = Rt`	Store conditional word. If no other atomic operation has been performed at the address (i.e., atomicity is ensured), perform the store to the word at address Rs and return TRUE in Pd; otherwise return FALSE. TRUE indicates that the LL and SC operations have been performed atomically.
`Rdd = memd_locked(Rs)`	Load locked doubleword. Reserve lock on doubleword at address Rs.
`memd_locked(Rs,Pd) = Rtt`	Store conditional doubleword. If no other atomic operation has been performed at the address (i.e., atomicity is ensured), perform the store to the doubleword at address Rs and return TRUE in Pd; otherwise return FALSE. TRUE indicates that the LL and SC operations have been performed atomically.

Here is the recommended code sequence to acquire a mutex:

```
// assume mutex address is held in R0
// assume R1,R3,P0,P1 are scratch

lockMutex:
R3 = #1
lock_test_spin:
R1 = memw_locked(R0)          // do normal test to wait
P1 = cmp.eq(R1,#0)            // for lock to be available
if (!P1) jump lock_test_spin
memw_locked(R0,P0) = r3       // do store conditional (SC)
if (!P0) jump lock_test_spin  // was LL and SC done atomically?
```

Here is the recommended code sequence to release a mutex:

```
// assume mutex address is held in R0
// assume R1 is scratch

R1 = #0
memw(R0) = R1
```

Atomic `memX_locked` operations are supported for external accesses that use the AXI bus and support atomic operations. To perform load-locked operations with external memory, the operating system must define the memory page as uncacheable, otherwise the processor behavior is undefined.

If a load locked operation is performed on an address that does not support atomic operations, the behavior is undefined.

For atomic operations on cacheable memory, the page attributes must be set to cacheable and write-back, otherwise the behavior is undefined. Cacheable memory must be used when threads need to synchronize with each other.

NOTE External `memX_locked` operations are not supported on the AHB bus. If they are performed on the AHB bus, the behavior is undefined.

6 Conditional Execution

6.1 Overview

The Hexagon processor uses a conditional execution model based on compare instructions that set predicate bits in one of four 8-bit predicate registers (`P0-P3`). These predicate bits can be used to conditionally execute certain instructions.

Conditional scalar operations examine only the least-significant bit in a predicate register, while conditional vector operations examine multiple bits in the register.

Branch instructions are the main consumers of the predicate registers.

6.2 Scalar predicates

Scalar predicates are 8-bit values which are used in conditional instructions to represent truth values:

- 0xFF represents true
- 0x00 represents false

The Hexagon processor provides the four 8-bit predicate registers P0-P3 to hold scalar predicates (Section 2.3.5). These registers are assigned values by the predicate-generating instructions, and examined by the predicate-consuming instructions.

6.2.1 Generating scalar predicates

The following instructions generate scalar predicates:

- Compare byte, halfword, word, doubleword
- Compare single- and double-precision floating point
- Classify floating-point value
- Compare bitmask
- Bounds check
- TLB match
- Store conditional

Table 6-1 lists the scalar predicate-generating instructions.

Table 6-1 Scalar predicate-generating instructions

Syntax	Operation

`Pd = cmpb.eq(Rs,{Rt,#u8})`
`Pd = cmph.eq(Rs,{Rt,#s8})`
`Pd = [!]cmp.eq(Rs,{Rt,#s10})`
`Pd = cmp.eq(Rss,Rtt)`
`Pd = sfcmp.eq(Rs,Rt)`
`Pd = dfcmp.eq(Rss,Rtt)`
	Equal (signed).
	Compare register Rs to Rt or a signed immediate for equality. Assign Pd the resulting truth value.

`Pd = cmpb.gt(Rs,{Rt,#s8}`
`Pd = cmph.gt(Rs,{Rt,#s8})`
`Pd = [!]cmp.gt(Rs,{Rt,#s10})`
`Pd = cmp.gt(Rss,Rtt)`
`Pd = sfcmp.gt(Rs,Rt)`
`Pd = dfcmp.gt(Rss,Rtt)`
	Greater than (signed).
	Compare register Rs to Rt or a signed immediate for signed greater than. Assign Pd the resulting truth value.

`Pd = cmpb.gtu(Rs,{Rt,#u7})`
`Pd = cmph.gtu(Rs,{Rt,#u7})`
`Pd = [!]cmp.gtu(Rs,{Rt,#u9})`
`Pd = cmp.gtu(Rss,Rtt)`
	Greater than (unsigned).
	Compare register Rs to Rt or an unsigned immediate for unsigned greater than. Assign Pd the resulting truth value.

Table 6-1 Scalar predicate-generating instructions (Continued)

`Pd = cmp.ge(Rs,#s8)`
`Pd = sfcmp.ge(Rs,Rt)`
`Pd = dfcmp.ge(Rss,Rtt)`
	Greater than or equal (signed).
	Compare register Rs to Rt or a signed immediate for signed greater than or equal. Assign Pd the resulting truth value.

`Pd = cmp.geu(Rs,#u8)`	Greater than or equal (unsigned). Compare register Rs to an unsigned immediate for unsigned greater than or equal. Assign Pd the resulting truth value.
`Pd = cmp.lt(Rs,Rt)`	Less than (signed). Compare register Rs to Rt for signed less than. Assign Pd the resulting truth value.
`Pd = cmp.ltu(Rs,Rt)`	Less than (unsigned). Compare register Rs to Rt for unsigned less than. Assign Pd the resulting truth value.
`Pd = sfcmp.uo(Rs,Rt)` `Pd = dfcmp.uo(Rss,Rtt)`	Unordered (signed). Determine if register Rs or Rt is set to the value NaN. Assign Pd the resulting truth value.
`Pd=sfclass(Rs,#u5)` `Pd=dfclass(Rss,#u5)`	Classify value (signed). Determine if register Rs is set to any of the specified classes. Assign Pd the resulting truth value.
`Pd = [!]tstbit(Rs,{Rt,#u5})`	Test if bit set. Rt or an unsigned immediate specifies a bit position. Test if the bit in Rs that is specified by the bit position is set. Assign Pd the resulting truth value.
`Pd = [!]bitsclr(Rs,{Rt,#u6})`	Test if bits clear. Rt or an unsigned immediate specifies a bitmask. Test if the bits in Rs that are specified by the bitmask are all clear. Assign Pd the resulting truth value.
`Pd = [!]bitsset(Rs,Rt)`	Test if bits set. Rt specifies a bitmask. Test if the bits in Rs that are specified by the bitmask are all set. Assign Pd the resulting truth value.
`memw_locked(Rs,Pd) = Rt` `memd_locked(Rs,Pd) = Rtt`	Store conditional. If no other atomic operation has been performed at the address (i.e., atomicity is ensured), perform the store to the word at address Rs. Assign Pd the resulting truth value.
`Pd = boundscheck(Rs,Rtt)`	Bounds check. Determine if Rs falls in the numeric range defined by Rtt. Assign Pd the resulting truth value.
`Pd = tlbmatch(Rss,Rt)`	Determine if TLB entry in Rss matches the ASID:PPN specified in Rt. Assign Pd the resulting truth value.

NOTE One of the compare instructions (`cmp.eq`) includes a variant which stores a binary predicate value (`0` or `1`) in a general register not a predicate register.

6.2.2 Consuming scalar predicates

Certain instructions can be conditionally executed based on the value of a scalar predicate (or alternatively specify a scalar predicate as an input to their operation).

The conditional instructions that consume scalar predicates examine only the least-significant bit of the predicate value. In the simplest case, this bit value directly determines whether the instruction is executed:

- 1 indicates that the instruction is executed
- 0 indicates that the instruction is not executed

If a conditional instruction includes the operator `!` in its predicate expression, then the logical negation of the bit value determines whether the instruction is executed.

Conditional instructions are expressed in assembly language with the instruction prefix "`if (`*`pred_expr`*`)`", where *`pred_expr`* specifies the predicate expression. For example:

```
if (P0) jump target          // jump if P0 is true
if (!P2) R2 = R5             // assign register if !P2 is true
if (P1) R0 = sub(R2,R3)      // conditionally subtract if P1
if (P2) R0 = memw(R2)        // conditionally load word if P2
```

The following instructions can be used as conditional instructions:

- Jumps and calls (Section 7.4)
- Many load and store instructions (Section 5.9)
- Logical instructions (including AND/OR/XOR)
- Shift halfword
- 32-bit add/subtract by register or short immediate
- Sign and zero extend
- 32-bit register transfer and 64-bit combine word
- Register transfer immediate
- Deallocate frame and return

When a conditional load or store is executed and the predicate expression is false, the instruction is cancelled (including any exceptions that might occur). For example, if a conditional load uses an address with a memory permission violation, and the predicate expression is false, then the load does not execute and the exception is not raised.

The `mux` instruction accepts a predicate as one of its basic operands:

```
Rd = mux(Ps,Rs,Rt)
```

`mux` selects either Rs or Rt based on the least significant bit in Ps. If the least-significant bit in Ps is a 1, then Rd is set to Rs, otherwise it is set to Rt.

6.2.3 Auto-AND predicates

If multiple compare instructions in a packet write to the same predicate register, the result is the logical AND of the individual compare results. For example:

```
{
P0 = cmp(A)                          // if A && B then jump
P0 = cmp(B)
if (P0.new) jump:T taken_path
}
```

To perform the corresponding OR operation, the following instructions can be used to compute the negation of an existing compare (using De Morgan's law):

- `Pd = !cmp.{eq,gt}(Rs, {#s10,Rt} )`
- `Pd = !cmp.gtu(Rs, {#u9,Rt} )`
- `Pd = !tstbit(Rs, {#u5,Rt} )`
- `Pd = !bitsclr(Rs, {#u6,Rt} )`
- `Pd = !bitsset(Rs,Rt)`

Auto-AND predicates have the following restrictions:

- If a packet contains `endloopN`, it cannot perform an auto-AND with predicate register P3.
- If a packet contains a register transfer from a general register to a predicate register, then no other instruction in the packet can write to the same predicate register. (As a result, a register transfer to `P3:0` or `C5:4` cannot be grouped with any other predicate-writing instruction.)
- The instructions `spNloop0`, `decbin`, `tlbmatch`, `memw_locked`, `memd_locked`, `add:carry`, `sub:carry`, `sfcmp`, and `dfcmp` cannot be grouped with another instruction that sets the same predicate register.

NOTE A register transfer from a predicate register to a predicate register has the same auto-AND behavior as a compare instruction.

6.2.4 Dot-new predicates

The Hexagon processor can generate and use a scalar predicate in the same instruction packet (Section 3.4). This feature is expressed in assembly language by appending the suffix ".new" to the specified predicate register. For example:

```
if (P0.new) R3 = memw(R4)
```

To see how dot-new predicates are used, consider the following C statement and the corresponding assembly code that is generated from it by the compiler:

C statement

```
if (R2 == 4)
R3 = *R4;
else
R5 = 5;
```

Assembly code

```
{
P0 = cmp.eq(R2,#4)
if (P0.new) R3 = memw(R4)
if (!P0.new) R5 = #5
}
```

In the assembly code a scalar predicate is generated and then consumed twice within the same instruction packet.

The following conditions apply to using dot-new predicates:

- The predicate must be generated by an instruction in the same packet. The assembler normally enforces this restriction, but if the processor executes a packet that violates this restriction, the execution result is undefined.
- A single packet can contain both the dot-new and normal forms of predicates. The normal form examines the old value in the predicate register, rather than the newly-generated value. For example:

```
{
  P0 = cmp.eq(R2,#4)
  if (P0.new) R3 = memw(R4)  // use newly-generated P0 value
  if (P0) R5 = #5            // use previous P0 value
}
```

6.2.5 Dependency constraints

Two instructions in an instruction packet should not write to the same destination register (Section 3.4.5). An exception to this rule is if the two instructions are conditional, and only one of them ever has the predicate expression value `true` when the packet is executed.

For example, the following packet is valid as long as `P2` and `P3` never both evaluate to `true` when the packet is executed:

```
{
   if (P2) R3 = #4       // P2, P3, or both must be false
   if (P3) R3 = #7
}
```

Because predicate values change at runtime, the programmer is responsible for ensuring that such packets are always valid during program execution. If they are invalid, the processor takes the following actions:

- When writing to general registers, an error exception is raised.
- When writing to predicate or control registers, the result is undefined.

6.3 Vector predicates

The predicate registers are also used for conditional vector operations. Unlike scalar predicates, vector predicates contain multiple truth values which are generated by vector predicate-generating operations.

For example, a vector compare instruction compares each element of a vector and assigns the compare results to a predicate register. Each bit in the predicate vector contains a truth value indicating the outcome of a separate compare performed by the vector instruction.

The vector `mux` instruction uses a vector predicate to selectively merge elements from two separate vectors into a single destination vector. This operation is useful for enabling the vectorization of loops with control flow (i.e., branches).

The vector instructions that use predicates are described in the following sections.

6.3.1 Vector compare

A vector compare instruction inputs two 64-bit vectors, performs separate compares for each pair of vector elements, and generates a predicate value which contains a bit vector of truth values.

Figure 6-1 shows an example of a vector byte compare.

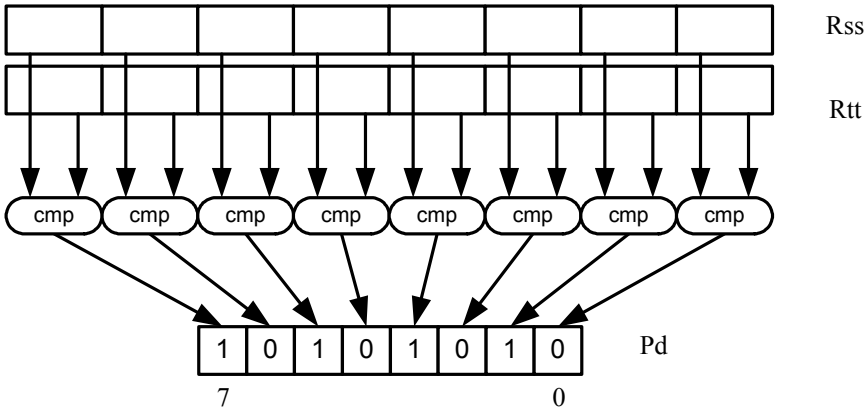


Figure 6-1 Vector byte compare

In Figure 6-1 two 64-bit vectors of bytes (contained in Rss and Rtt) are being compared. The result is assigned as a vector predicate to the destination register Pd.

In the example vector predicate shown in Figure 6-1, note that every other compare result in the predicate is true (i.e., 1).

Figure 6-2 shows how a vector halfword compare generates a vector predicate.

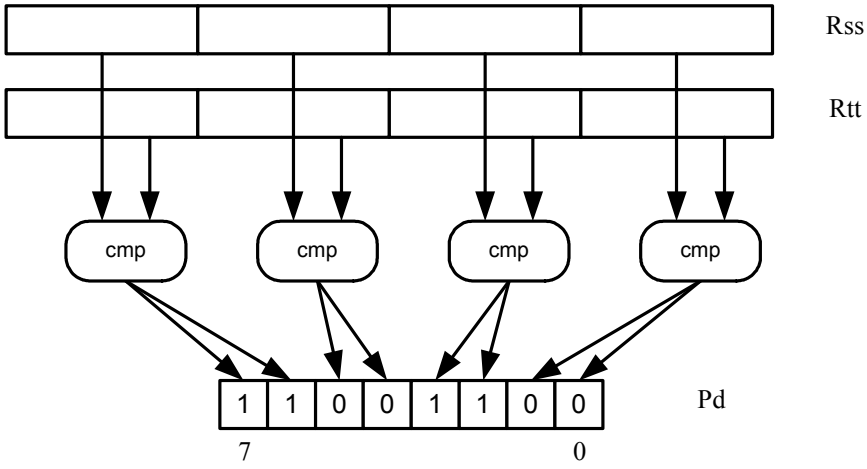


Figure 6-2 Vector halfword compare

In Figure 6-2 two 64-bit vectors of halfwords are being compared. The result is assigned as a vector predicate to the destination register Pd.

Because a vector halfword compare yields only four truth values, each truth value is encoded as two bits in the generated vector predicate.

6.3.2 Vector mux instruction

A vector mux instruction is used to conditionally select the elements from two vectors. The instruction takes as input two source vectors and a predicate register. For each byte in the vector, the corresponding bit in the predicate register is used to choose from one of the two input vectors. The combined result is written to the destination register.

Figure 6-3 shows the operation of the vector mux instruction.

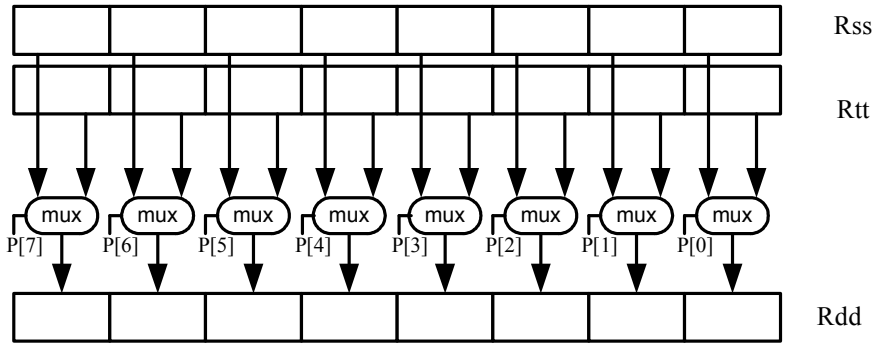


Figure 6-3 Vector mux instruction

Table 6-2 defines the vector mux instruction.

Table 6-2 Vector mux instruction

Syntax	Operation
`Rdd = vmux(Ps,Rss,Rtt)`	Select bytes from Rss and Rtt

Changing the order of the source operands in a mux instruction enables both senses of the result to be formed. For example:

```
R1:0 = vmux(P0,R3:2,R5:4)     // choose bytes from R3:2 if true
R1:0 = vmux(P0,R5:4,R3:2)     // choose bytes from R3:2 if false
```

NOTE By replicating the predicate bits generated by word or halfword compares, the vector mux instruction can be used to select words or halfwords.

6.3.3 Using vector conditionals

Vector conditional support is used to vectorize loops with conditional statements.

Consider the following C statement:

```
for (i=0; i<8; i++) {
if (A[i]) {
   B[i] = C[i];
}
}
```

Assuming arrays of bytes, this code can be vectorized as follows:

```
R1:0 = memd(R_A)            // R1:0 holds A[7]-A[0]
R3 = #0                    // clear R3:2
R2 = #0
P0 = vcmpb.eq(R1:0,R3:2)    // compare bytes in A to zero
R5:4 = memd(R_B)            // R5:4 holds B[7]-B[0]
R7:6 = memd(R_C)            // R7:6 holds C[7]-C[0]
R3:2 = vmux(P0,R7:6,R5:4)   // if (A[i]) B[i]=C[i]
memd(R_B) = R3:2            // store B[7]-B[0]
```

6.4 Predicate operations

The Hexagon processor provides a set of operations for manipulating and moving predicate registers.

Table 6-3 lists the predicate register instructions.

Table 6-3 Predicate register instructions

Syntax	Operation
`Pd = Ps`	Transfer predicate Ps to Pd
`Pd = Rs`	Transfer register Rs to predicate Pd
`Rd = Ps`	Transfer predicate Ps to register Rd
`Pd = and(Ps,[!]Pt)`	Set Pd to bitwise AND of Ps and [NOT] Pt
`Pd = or(Ps,[!]Pt)`	Set Pd to bitwise OR of Ps and [NOT] Pt
`Pd = and(Ps, and(Pt,[!]Pu)`	Set Pd to AND of Ps and (AND of Pt and [NOT] Pu)
`Pd = and(Ps, or(Pt,[!]Pu)`	Set Pd to AND of Ps and (OR of Pt and [NOT] Pu)
`Pd = or(Ps, and(Pt,[!]Pu)`	Set Pd to OR of Ps and (AND of Pt and [NOT] Pu)
`Pd = or(Ps, or(Pt,[!]Pu)`	Set Pd to OR of Ps and (OR of Pt and [NOT] Pu)
`Pd = not(Ps)`	Set Pd to bitwise inversion of Ps
`Pd = xor(Ps,Pt)`	Set Pd to bitwise exclusive OR of Ps and Pt
`Pd = any8(Ps)`	Set Pd to 0xFF if any bit in Ps is 1, 0x00 otherwise
`Pd = all8(Ps)`	Set Pd to 0x00 if any bit in Ps is 0, 0xFF otherwise

NOTE These instructions belong to instruction class CR.

Predicate registers can be transferred to and from the general registers either individually or as register quadruples (Section 2.3.5).

7 Program Flow

7.1 Overview

The Hexagon processor supports the following program flow facilities:

- Conditional instructions
- Hardware loops
- Software branches
- Pauses
- Exceptions

Software branches include jumps, calls, and returns. Several types of jumps are supported:

- Speculative jumps
- Compare jumps
- Register transfer jumps
- Dual jumps

7.2 Conditional instructions

Many Hexagon processor instructions can be conditionally executed. For example:

```
if (P0) R0 = memw(R2)     // conditionally load word if P0
if (!P1) jump label       // conditionally jump if not P1
```

The following instructions can be specified as conditional:

- Jumps and calls
- Many load and store instructions
- Logical instructions (including AND/OR/XOR)
- Shift halfword
- 32-bit add/subtract by register or short immediate
- Sign and zero extend
- 32-bit register transfer and 64-bit combine word
- Register transfer immediate
- Deallocate frame and return

For more information see Section 5.9 and Chapter 6.

7.3 Hardware loops

The Hexagon processor includes *hardware loop* instructions which can perform loop branches with zero overhead. For example:

```
  loop0(start,#3)           // loop 3 times
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

Two sets of hardware loop instructions are provided – `loop0` and `loop1` – to enable hardware loops to be nested one level deep. For example:

```
// Sum the rows of a 100x200 matrix.

  loop1(outer_start,#100)
outer_start:
    R0 = #0
    loop0(inner_start,#200)
inner_start:
      R3 = memw(R1++#4)
```

```
    { R0 = add(R0,R3) }:endloop0
{ memw(R2++#4) = R0 }:endloop1
```

The hardware loop instructions are used as follows:

- For non-nested loops, `loop0` is used.
- For nested loops, `loop0` is used for the inner loop, and `loop1` for the outer loop.

NOTE If a program needs to create loops nested more than one level deep, the two innermost loops can be implemented as hardware loops, with the remaining outer loops implemented as software branches.

Each hardware loop is associated with a pair of dedicated loop registers:

- The *loop start address* register `SAn` is set to the address of the first instruction in the loop (which is typically expressed in assembly language as a label).
- The *loop count* register `LCn` is set to a 32-bit unsigned value which specifies the number of loop iterations to perform. When the PC reaches the end of the loop, `LCn` is examined to determine whether the loop should repeat or exit.

The hardware loop setup instruction sets both of these registers at once – typically there is no need to set them individually. However, because the loop registers completely specify the hardware loop state, they can be saved and restored (either automatically by a processor interrupt or manually by the programmer), enabling a suspended hardware loop to be resumed normally once its loop registers are reloaded with the saved values.

The Hexagon processor provides two sets of loop registers for the two hardware loops:

- `SA0` and `LC0` are used by `loop0`
- `SA1` and `LC1` are used by `loop1`

Table 7-1 lists the hardware loop instructions.

Table 7-1 Loop instructions

Syntax	Description
loopN(start, Rs)	Hardware loop with register loop count. Set registers SAn and LCn for hardware loop N: ■ SAn is assigned the specified start address of the loop. ■ LCn is assigned the value of general register Rs. NOTE - The loop start operand is encoded as a PC-relative immediate value.
loopN(start, #count)	Hardware loop with immediate loop count. Set registers SAn and LCn for hardware loop N: ■ SAn is assigned the specified start address of the loop. ■ LCn is assigned the specified immediate value (0-1023). NOTE - The loop start operand is encoded as a PC-relative immediate value.
:endloopN	Hardware loop end instruction. Performs the following operation: `if (LCn > 1) {PC = SAn; LCn = LCn-1}` NOTE - This instruction appears in assembly as a suffix appended to the last packet in the loop. It is encoded in the last packet.
SAn = Rs	Set loop start address to general register Rs
LCn = Rs	Set loop count to general register Rs

NOTE The loop instructions are assigned to instruction class CR.

7.3.1 Loop setup

To set up a hardware loop, the loop registers `SAn` and `LCn` must be set to the proper values. This can be done in two ways:

- A `loopN` instruction
- Register transfers to `SAn` and `LCn`

The `loopN` instruction performs all the work of setting `SAn` and `LCn`. For example:

```
  loop0(start,#3)           // SA0=&start, LC0=3
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

In this example the hardware loop (consisting of a single multiply instruction) is executed three times. The `loop0` instruction sets register `SA0` to the address value of label `start`, and `LC0` to 3.

Loop counts are limited to the range 0-1023 when they are expressed as immediate values in `loopN`. If the desired loop count exceeds this range, it must be specified as a register value. For example:

Using `loopN`:

```
    R1 = #20000;
    loop0(start,R1)          // LC0=20000, SA0=&start
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

Using register transfers:

```
    R1 = #20000
    LC0 = R1                 // LC0=20000
    R1 = #start
    SA0 = R1                 // SA0=&start
start:
    { R0 = mpyi(R0,R0) } :endloop0
```

If a `loopN` instruction is located too far from its loop start address, the PC-relative offset value that is used to specify the start address may exceed the maximum range of the instruction's start-address operand. If this occurs, either move the `loopN` instruction closer to the loop start, or specify the loop start address as a 32-bit constant (Section 10.10). For example:

Using 32-bit constants:

```
    R1 = #20000;
    loop0(##start,R1)        // LC0=20000, SA0=&start
    ...
```

7.3.2 Loop end

The loop end instruction indicates the last packet in a hardware loop. It is expressed in assembly language by appending the packet with the symbol "`:endloopN`", where `N` specifies the hardware loop (0 or 1). For example:

```
    loop0(start,#3)
start:
    { R0 = mpyi(R0,R0) } :endloop0  // last packet in loop
```

The last instruction in the loop must always be expressed in assembly language as a packet (using curly braces), even if it is the only instruction in the packet.

Nested hardware loops can specify the same instruction as the end of both the inner and outer loops. For example:

```
// Sum the rows of a 100x200 matrix.
// Software pipeline the outer loop.

     p0 = cmp.gt(R0,R0)              // p0 = false
     loop1(outer_start,#100)
outer_start:
     { if (p0) memw(R2++#4) = R0
       p0 = cmp.eq(R0,R0)            // p0 = true
       R0 = #0
       loop0(inner_start,#200) }
inner_start:
     R3 = memw(R1++#4)
     { R0 = add(R0,R3) }:endloop0:endloop1
     memw(R2++#4) = R0
```

Though `endloopN` behaves like a regular instruction (by implementing the loop test and branch), note that it does not execute in any instruction slot, and does not count as an instruction in the packet. Therefore a single instruction packet which is marked as a loop end can perform up to six operations:

- Four regular instructions (the normal limit for an instruction packet)
- The `endloop0` test and branch
- The `endloop1` test and branch

NOTE The `endloopN` instruction is encoded in the instruction packet (Section 10.7).

7.3.3 Loop execution

After a hardware loop is set up, the loop body always executes at least once regardless of the specified loop count (because the loop count is not examined until the last instruction in the loop). Therefore, if a loop needs to be optionally executed zero times, it must be preceded with an explicit conditional branch. For example:

```
  loop0(start,R1)
  P0 = cmp.eq(R1,#0)
  if (P0) jump skip
start:
    { R0 = mpyi(R0,R0) } :endloop0
skip:
```

In this example a hardware loop is set up with the loop count in R1, but if the value in R1 is zero a software branch skips over the loop body.

After the loop end instruction of a hardware loop is executed, the Hexagon processor examines the value in the corresponding loop count register:

- If the value is greater than 1, the processor decrements the loop count register and performs a zero-cycle branch to the loop start address.
- If the value is less than or equal to 1, the processor resumes program execution at the instruction immediately following the loop end instruction.

NOTE Because nested hardware loops can share the same loop end instruction, the processor may examine both loop count registers in a single operation.

7.3.4 Pipelined hardware loops

Software pipelined loops are common for VLIW architectures such as the Hexagon processor. They offer increased code performance in loops by overlapping multiple loop iterations.

A software pipeline has three sections:

- A *prologue* in which the loop is primed
- A *kernel* (or steady-state) portion
- An *epilogue* which drains the pipeline

This is best illustrated with a simple example, as shown in Table 7-2.

Table 7-2 Software pipelined loop

```
int foo(int *A, int *result)
{
    int i;
    for (i=0;i<100;i++) {
        result[i]= A[i]*A[i];
    }
}
```

```
foo:
{       R3 = R1
        loop0(.kernel,#98)          // Decrease loop count by 2
}
        R1 = memw(R0++#4)           // 1st prologue stage
{       R1 = memw(R0++#4)           // 2nd prologue stage
        R2 = mpyi(R1,R1)
}
        .falign
.kernel:
{       R1 = memw(R0++#4)           // kernel
        R2 = mpyi(R1,R1)
        memw(R3++#4) = R2
}:endloop0
{       R2 = mpyi(R1,R1)            // 1st epilogue stage
        memw(R3++#4) = R2
}
        memw(R3++#4) = R2           // 2nd epilogue stage
        jumpr lr
```

In Table 7-2 the kernel section of the pipelined loop performs three iterations of the loop in parallel:

- The load for iteration N+2
- The multiply for iteration N+1
- The store for iteration N

One drawback to software pipelining is the extra code necessary for the prologue and epilogue sections of a pipelined loop.

To address this issue the Hexagon processor provides the `spNloop0` instruction, where the "N" in the instruction name indicates a digit in the range 1-3. For example:

```
P3 = sp2loop0(start,#10)     // Set up pipelined loop
```

spNloop0 is a variant of the loop0 instruction: it sets up a normal hardware loop using SA0 and LC0, but also performs the following addtional operations:

- When the spNloop0 instruction is executed, it assigns the truth value false to the predicate register P3.
- After the associated loop has executed N times, P3 is automatically set to true.

This feature (which is known as *automatic predicate control*) enables the store instructions in the kernel section of a pipelined loop to be conditionally executed by P3 and thus – because of the way spNloop0 controls P3 – not be executed during the pipeline warm-up. This can reduce the code size of many software pipelined loops by eliminating the need for prologue code.

spNloop0 cannot be used to eliminate the epilogue code from a pipelined loop; however, in some cases it is possible to do this through the use of programming techniques.

Typically, the issue affecting the removal of epilogue code is *load safety*. If the kernel section of a pipelined loop can safely access past the end of its arrays – either because it is known to be safe, or because the arrays have been padded at the end – then epilogue code is unnecessary. However, if load safety cannot be ensured, then explicit epilogue code is required to drain the software pipeline.

Table 7-3 shows how spNloop0 and load safety simplify the code shown in Table 7-2.

Table 7-3 Software pipelined loop (using spNloop0)

```
int foo(int *A, int *result)
{
    int i;
    for (i=0;i<100;i++) {
        result[i]= A[i]*A[i];
    }
}
```

```
foo:
{ // load safety assumed
        P3 = sp2loop0(.kernel,#102)     // set up pipelined loop
        R3 = R1
}
.falign
.kernel:
{       R1 = memw(R0++#4)               // kernel
        R2 = mpyi(R1,R1)
        if (P3) memw(R3++#4) = R2
}:endloop0

        jumpr lr
```

NOTE The count value that spNloop0 uses to control the P3 setting is stored in the user status register USR.LPCFG.

7.3.5 Loop restrictions

Hardware loops have the following restrictions:

- The loop setup packet in `loopN` or `spNloop0` (Section 7.3.4) cannot contain a speculative indirect jump, new-value compare jump, or `dealloc_return`.
- The last packet in a hardware loop cannot contain any program flow instructions (including jumps or calls).
- The loop end packet in `loop0` cannot contain any instruction that changes `SA0` or `LC0`. Similarly, the loop end packet in `loop1` cannot contain any instruction that changes `SA1` or `LC1`.
- The loop end packet in `spNloop0` cannot contain any instruction that changes `P3`.

NOTE `SA1` and `LC1` can be changed at the end of `loop0`, while `SA0` and `LC0` can be changed at the end of `loop1`.

7.4 Software branches

Unlike hardware loops, *software branches* use an explicit instruction to perform a branch operation. Software branches include the following instructions:

- Jumps
- Calls
- Returns

The target address for branch instructions can be specified as register indirect or PC-relative offsets. PC-relative offsets are normally less than 32 bits, but can be specified as 32 bits by using the appropriate syntax in the target operand (Section 7.4.4).

Branch instructions can be unconditional or conditional, with the execution of conditional instructions controlled by a predicate expression.

Table 7-4 summarizes the software branch instructions.

Table 7-4 Software branch instructions

Syntax	Operation
[if (pred_expr)] jump label [if (pred_expr)] jumpr Rs	Branch to address specified by register Rs or PC-relative offset. Can be conditionally executed.
[if (pred_expr)] call label [if (pred_expr)] callr Rs	Branch to address specified by register Rs or PC-relative offset. Store subroutine return address in link register LR. Can be conditionally executed.
[if (pred_expr)] jumpr LR	Branch to subroutine return address contained in link register LR. Can be conditionally executed.

7.4.1 Jumps

Jump instructions change the program flow to a target address which can be specified by either a register or a PC-relative immediate value. Jump instructions can be conditional based on the value of a predicate expression.

Table 7-5 lists the jump instructions.

Table 7-5 Jump instructions

Syntax	Operation
jump label	Direct jump. Branch to address specified by label. Label is encoded as PC-relative signed immediate value.
jumpr Rs	Indirect jump. Branch to address contained in general register Rs.
if ([!]Ps) jump label if ([!]Ps) jumpr Rs	Conditional jump. Perform jump if predicate expression evaluates to true.

NOTE Conditional jumps can be specified as speculative (Section 7.5).

7.4.2 Calls

Call instructions are used to jump to subroutines. The instruction performs a jump to the target address and also stores the return address in the link register `LR`.

The forms of call are functionally similar to jump instructions and include both PC-relative and register indirect in both unconditional and conditional forms.

Table 7-6 lists the call instructions.

Table 7-6 Call instructions

Syntax	Operation
call label	Direct subroutine call. Branch to address specified by label, and store return address in register LR. Label is encoded as PC-relative signed immediate value.
callr Rs	Indirect subroutine call. Branch to address contained in general register Rs, and store return address in register LR.
if ([!]Ps) call label if ([!]Ps) callr Rs	Conditional call. If predicate expression evaluates to true, perform subroutine call to specified target address.

7.4.3 Returns

Return instructions are used to return from a subroutine. The instruction performs an indirect jump to the subroutine return address stored in link register `LR`.

Returns are implemented as jump register indirect instructions, and support both unconditional and conditional forms.

Table 7-7 lists the return instructions.

Table 7-7 Return instructions

Syntax	Operation
jumpr LR	Subroutine return. Branch to subroutine return address contained in link register LR.
if ([!]Ps) jumpr LR	Conditional subroutine return. If predicate expression evaluates to true, perform subroutine return to specified target address.
dealloc_return	Subroutine return with stack frame deallocate. Perform `deallocframe` operation (Section 8.6) and then perform subroutine return to the target address loaded by `deallocframe` from the link register.
if ([!]Ps) dealloc_return	Conditional subroutine return with stack frame deallocate. If predicate expression evaluates to true, perform `deallocframe` and then subroutine return to the target address loaded by `deallocframe` from the link register.

NOTE The link register `LR` is an alias of general register `R31`. Therefore subroutine returns can be performed with the instruction `jumpr R31`.

The conditional subroutine returns (including `dealloc_return`) can be specified as speculative (Section 7.5).

7.4.4 Extended branches

When a `jump` or `call` instruction specifies a PC-relative offset as the branch target, the offset value is normally encoded in significantly less than 32 bits. This can limit the ability for programs to specify "long" branches which span a large range of the processor's memory address space.

To support long branches, the `jump` and `call` instructions have special versions which encode a full 32-bit value as the PC-relative offset.

NOTE Such instructions use an extra word to store the 32-bit offset (Section 10.10).

The size of a PC-relative branch offset is expressed in assembly language by optionally prefixing the target label with the symbol "`##`" or "`#`":

- "`##`" specifies that the assembler *must* use a 32-bit offset
- "`#`" specifies that the assembler must *not* use a 32-bit offset
- No "`#`" specifies that the assembler use a 32-bit offset only if necessary

For example:

```
jump ##label   // 32-bit offset
call #label    // non 32-bit offset
jump label     // offset size determined by assembler
```

7.4.5 Branches to and from packets

Instruction packets are atomic: even if they contain multiple instructions, they can be referenced only by the address of the first instruction in the packet. Therefore, branches to a packet can target only the packet's first instruction.

Packets can contain up to two branches (Section 7.8). The branch destination can target the current packet or the beginning of another packet.

A branch does not interrupt the execution of the current packet: all the instructions in the packet are executed, even if they appear in the assembly source after the branch instruction.

If a packet is at the end of a hardware loop, it cannot contain a branch instruction.

7.5 Speculative jumps

Conditional instructions normally depend on predicates that are generated in a previous instruction packet. However, dot-new predicates (Section 6.2.3) enable conditional instructions to use a predicate generated in the same packet that contains the conditional instruction.

When dot-new predicates are used with a conditional jump, the resulting instruction is called a *speculative jump*. For example:

```
{
   P0 = cmp.eq(R9,#16)         // single-packet compare-and-jump
   IF (P0.new) jumpr:t R11     // ... enabled by use of P0.new
}
```

Speculative jumps require the programmer to specify a *direction hint* in the jump instruction, which indicates whether the conditional jump is expected to be taken or not.

The hint is used to initialize the Hexagon processor's dynamic branch predictor. Whenever the predictor is wrong, the speculative jump instruction takes two cycles to execute instead of one (due to a pipeline stall).

Hints can improve program performance by indicating how speculative jumps are expected to execute over the course of a program: the more often the specified hint indicates how the instruction actually executes, the better the performance.

Hints are expressed in assembly language by appending the suffix "`:t`" or "`:nt`" to the `jump` instruction symbol. For example:

- `jump:t` – The jump instruction will most often be taken
- `jump:nt` – The jump instruction will most often be not taken

In addition to dot-new predicates, speculative jumps also accept conditional arithmetic expressions (=0, !=0, >=0, <=0) involving the general register `Rs`.

Table 7-8 lists the speculative jump instructions.

Table 7-8 Speculative jump instructions

Syntax	Operation
if ([!]Ps.new) jump:t label if ([!]Ps.new) jump:nt label	Speculative direct jump. If predicate expression evaluates to true, jump to address specified by label.
if ([!]Ps.new) jumpr:t Rs if ([!]Ps.new) jumpr:nt Rs	Speculative indirect jump. If predicate expression evaluates to true, jump to address in register Rs.
if (Rs == #0) jump:t label if (Rs == #0) jump:nt label	Speculative direct jump. If predicate Rs = 0 is true, jump to address specified by label.
if (Rs != #0) jump:t label if (Rs != #0) jump:nt label	Speculative direct jump. If predicate Rs != 0 is true, jump to address specified by label.

Table 7-8 Speculative jump instructions (Continued)

Syntax	Operation
if (Rs >= #0) jump:t label if (Rs >= #0) jump:nt label	Speculative direct jump. If predicate Rs >= 0 is true, jump to address specified by label.
if (Rs <= #0) jump:t label if (Rs <= #0) jump:nt label	Speculative direct jump. If predicate Rs <= 0 is true, jump to address specified by label.

NOTE The hints `:t` and `:nt` interact with the predicate value to determine the instruction cycle count.

Speculative indirect jumps are not supported with register `Rs` predicates.

7.6 Compare jumps

To reduce code size the Hexagon processor supports a compound instruction which combines a compare with a speculative jump in a single 32-bit instruction.

For example:

```
{
p0 = cmp.eq (R2,R5)           // single-instr compare-and-jump
if (p0.new) jump:nt target    // enabled by compound instr
}
```

The register operands used in a compare jump are limited to R0-R7 or R16-R23 (Table 10-3).

The compare and jump instructions that can be used in a compare jump are limited to the instructions listed in Table 7-9. The compare can use predicate P0 or P1, while the jump must specify the same predicate that is set in the compare.

A compare jump instruction is expressed in assembly source as two independent compare and jump instructions in a packet. The assembler translates the two instructions into a single compound instruction.

Table 7-9 Compare jump instructions

Compare Instruction	Jump Instruction

Pd = cmp.eq (Rs, Rt)
Pd = cmp.gt (Rs, Rt)
Pd = cmp.gtu (Rs, Rt)
Pd = cmp.eq (Rs,#U5)
Pd = cmp.gt (Rs,#U5)
Pd = cmp.gtu (Rs,#U5)
Pd = cmp.eq (Rs,#-1)
Pd = cmp.gt (Rs,#-1)
Pd = tstbit (Rs, #0)
	IF (Pd.new) jump:t label
	IF (Pd.new) jump:nt label
	IF (!Pd.new) jump:t label
	IF (!Pd.new) jump:nt label

7.6.1 New-value compare jumps

A compare jump instruction can access a register that is assigned a new value in the same instruction packet (Section 3.4). This feature is expressed in assembly language by the following changes:

- Appending the suffix ".new" to the new-value register in the compare
- Rewriting the compare jump so its constituent compare and jump operations appear as a single conditional instruction

For example:

```
// load-compare-and-jump packet enabled by new-value compare jump

{
R0 = memw(R2+#8)
if (cmp.eq(R0.new,#0)) jump:nt target
}
```

New-value compare jump instructions have the following restrictions:

- They are limited to the instruction forms listed in Table 7-10.
- They cannot be combined with another jump instruction in the same packet.
- If an instruction produces a 64-bit result or performs a floating-point operation (Section 1.4.4), its result registers cannot be used as the new-value register.
- If an instruction uses auto-increment or absolute-set addressing mode (Section 5.8), its address register cannot be used as the new-value register.
- If the instruction that sets a new-value register is conditional (Section 6.2.2), it must always be executed.

If the specified jump direction hint is wrong (Section 7.5), a new-value compare jump takes three cycles to execute instead of one. While this penalty is one cycle longer than in a regular speculative jump, the overall performance is still better than using a regular speculative jump (which must execute an extra packet in all cases).

NOTE New-value compare jump instructions are assigned to instruction class NV, which can execute only in Slot 0. The instruction that assigns the new value must execute in Slot 1, 2, or 3.

Table 7-10 New-value compare jump instructions

if ([!]cmp.eq (Rs.new, Rt)) jump:[hint] label
if ([!]cmp.gt (Rs.new, Rt)) jump:[hint] label
if ([!]cmp.gtu (Rs.new, Rt)) jump:[hint] label
 	
if ([!]cmp.gt (Rs, Rt.new)) jump:[hint] label
if ([!]cmp.gtu (Rs, Rt.new)) jump:[hint] label

if ([!]cmp.eq (Rs.new, #u5)) jump:[hint] label
if ([!]cmp.gt (Rs.new, #u5)) jump:[hint] label
if ([!]cmp.gtu (Rs.new ,#u5)) jump:[hint] label
 	
if ([!]cmp.eq (Rs.new, #-1)) jump:[hint] label
if ([!]cmp.gt (Rs.new, #-1)) jump:[hint] label
 	
if ([!]tstbit (Rs.new, #0)) jump:[hint] label

7.7 Register transfer jumps

To reduce code size the Hexagon processor supports a compound instruction which combines a register transfer with an unconditional jump in a single 32-bit instruction.

For example:

```
{
jump target     // jump to label "target"
R1 = R2         // assign contents of reg R2 to R1
}
```

The source and target register operands in the register transfer are limited to R0-R7 or R16-R23 (Table 10-3).

The target address in the jump is a scaled 9-bit PC-relative address value (as opposed to the 22-bit value in the regular unconditional jump instruction).

A register transfer jump instruction is expressed in assembly source as two independent instructions in a packet. The assembler translates the instructions into a single compound instruction.

Table 7-11 lists the register transfer jump instructions.

Table 7-11 Register transfer jump instructions

Syntax	Operation
jump label; Rd=Rs	Register transfer jump. Perform register transfer and branch to address specified by label. Label is encoded as PC-relative 9-bit signed immediate value.
jump label; Rd=#u6	Register transfer immediate jump. Perform register transfer (of 6-bit unsigned immediate value) and branch to address specified by label. Label is encoded as PC-relative 9-bit signed immediate value.

7.8 Dual jumps

Two software branch instructions (referred to here as "jumps") can appear in the same instruction packet, under the conditions listed in Table 7-12.

The first jump is defined as the jump instruction at the lower address, and the second jump as the jump instruction at the higher address.

Unlike most packetized operations, dual jumps are not executed in parallel (Section 3.4.1). Instead, the two jumps are processed in a well-defined order in a packet:

1. The predicate in the first jump is evaluated.
2. If the first jump is taken, the second jump is ignored.
3. If the first jump is not taken, the second jump is performed.

Table 7-12 Dual jump instructions

Instruction	Description	First jump in packet?	Second jump in packet?
jump	Direct jump	No	Yes
if ([!]Ps[.new]) jump	Conditional jump	Yes	Yes
call if ([!]Ps) call	Direct calls	No	Yes
Pd=cmp.xx ; if ([!]Pd.new) jump	Compare jump	Yes	Yes

Table 7-12 Dual jump instructions

Instruction	Description	First jump in packet?	Second jump in packet?
if ([!]cmp.xx(Rs.new, Rt)) jump	New-value compare jump	No	No

jumpr
if ([!]Ps[.new]) jumpr
callr
if ([!]Ps) callr
dealloc_return
if ([!]Ps[.new]) dealloc_return
	Indirect jumps
	Indirect calls
	`dealloc_return`
		No	No

endloopN	Hardware loop end	No	No

NOTE If a call is ignored in a dual jump, the link register LR is not changed.

7.9 Hint indirect jump target

Because it obtains the jump target address from a register, the `jumpr` instruction (Section 7.4.1) normally causes the processor to stall for one cycle.

To avoid the stall penalty caused by a `jumpr` instruction, the Hexagon processor supports the jump hint instruction `hintjr`, which can be specified before the `jumpr` instruction.

The `hintjr` instruction indicates that the program is about to execute a `jumpr` to the address contained in the specified register.

Table 7-13 lists the speculative jump instructions.

Table 7-13 Jump hint instruction

Syntax	Operation
hintjr(Rs)	Inform processor that `jumpr(Rs)` instruction is about to be performed.

NOTE In order to prevent a stall, the `hintjr` instruction must be executed at least 2 packets before the corresponding `jumpr` instruction.

The `hintjr` instruction is not needed for `jumpr` instructions used as returns (Section 7.4.3), because in this case the Hexagon processor automatically predicts the jump targets based on the most recent nested `call` instructions.

7.10 Pauses

Pauses suspend the execution of a program for a period of time, and put it into low-power mode. The program remains suspended for the duration specified in the instruction.

The `pause` instruction accepts an unsigned 8-bit immediate operand which specifies the pause duration in terms of cycles. The maximum possible duration is 263 cycles (255+8).

Hexagon processor interrupts cause a program to exit the paused state before its specified duration has elapsed.

The `pause` instruction is useful for implementing user-level low-power synchronization operations (such as spin locks).

Table 7-14 lists the `pause` instruction.

Table 7-14 Pause instruction

Syntax	Operation
`pause(#u8)`	Suspend program in low-power mode for specified cycle duration.

7.11 Exceptions

Exceptions are internally-generated disruptions to the program flow.

The Hexagon processor OS handles fatal exceptions by terminating the execution of the application system. The user is responsible for fixing the problem and recompiling their applications.

The error messages generated by exceptions include the following information to assist in locating the problem:

- Cause code – Hexadecimal value indicating the type of exception that occurred
- User IP – PC value indicating the instruction executed when exception occurred
- Bad VA – Virtual address indicating the data accessed when exception occurred

NOTE The cause code, user IP, and Bad VA values are stored in the Hexagon processor system control registers `SSR[7:0]`, `ELR`, and `BADVA` respectively.

If multiple exceptions occur simultaneously, the exception with the lowest error code value has the highest exception priority.

If a packet contains multiple loads, or a load and a store, and both operations have an exception of any type, then all Slot 1 exceptions are processed before any Slot 0 exception is processed.

NOTE The exceptions defined for processor versions V60 and V61 are identical, with one exception: V61 defines an additional event (with cause code 0x27) to indicate stack overflow.

Table 7-15 lists the exceptions for the V60 and V61 processor versions.

Table 7-15 V60/61 exceptions

Cause Code	Event Type	Event Description	Notes
0x0	Reset	Software thread reset.	Non-maskable, Highest Priority
0x01	Precise, Unrecoverable	Unrecoverable BIU error (bus error, timeout, L2 parity error, etc.).	Non-maskable
0x03	Precise, Unrecoverable	Double exception (exception occurs while SSR[EX]=1).	Non-maskable
0x11	Precise	Privilege violation: User/Guest mode execute to page with no execute permissions (X=0).	Non-maskable
0x12	Precise	Privilege violation: User mode execute to a page with no user permissions (X=1, U=0).	Non-maskable
0x15	Precise	Invalid packet.	Non-maskable
0x16	Precise	Illegal Execution of Coprocessor Instruction.	Non-maskable
0x1A	Precise	Privilege violation: Executing a guest mode instruction in user mode.	Non-maskable
0x1B	Precise	Privilege violation: Executing a supervisor instruction in user/guest mode.	Non-maskable
0x1D	Precise, Unrecoverable	Packet with multiple writes to the same destination register.	Non-maskable
0x1E	Precise, Unrecoverable	Program counter values that are not properly aligned.	Non-maskable
0x20	Precise	Load to misaligned address.	Non-maskable
0x21	Precise	Store to misaligned address.	Non-maskable
0x22	Precise	Privilege violation: User/Guest mode Read to page with no read permission (R=0).	Non-maskable
0x23	Precise	Privilege violation: User/Guest mode Write to page with no write permissions (W=0).	Non-maskable
0x24	Precise	Privilege violation: User mode Read to page with no user permission (R=1, U=0).	Non-maskable
0x25	Precise	Privilege violation: User mode Write to page with no user permissions (W=1, U=0).	Non-maskable
0x26	Precise	Coprocessor VMEM address error.	Non-maskable
0x27	Precise	Stack overflow: Allocframe instruction exceeded FRAMELIMIT.	Non-maskable, V61 and greater
0x42	Imprecise	Data abort.	Non-maskable
0x43	Imprecise	NMI.	Non-maskable
0x44	Imprecise	Multiple TLB match.	Non-maskable
0x45	Imprecise	Livelock exception.	Non-maskable
0x60	TLB miss-X	Due to missing Fetch address on PC-page.	Non-maskable

Table 7-15 V60/61 exceptions (Continued)

Cause Code	Event Type	Event Description	Notes
0x61	TLB miss-X	Due to missing Fetch on second page from packet that spans pages.	Non-maskable
0x62	TLB miss-X	Due to `icinva`.	Non-maskable
	Reserved		
0x70	TLB miss-RW	Due to memory read.	Non-maskable
0x71	TLB miss-RW	Due to memory write.	Non-maskable
	Reserved		
#u8	Trap0	Software Trap0 instruction.	Non-maskable
#u8	Trap1	Software Trap1 instruction.	Non-maskable
	Reserved		
0x80	Debug	Single-step debug exception.	
	Reserved		
0xBF	Floating-Point	Execution of Floating-Point instruction resulted in exception.	Non-maskable
0xC0	Interrupt0	General external interrupt.	Maskable, highest priority general interrupt
0xC1	Interrupt 1	General external interrupt	Maskable
0xC2	Interrupt 2		
0xC3	Interrupt 3		
0xC4	Interrupt 4		
0xC5	Interrupt 5		
0xC6	Interrupt 6		
0xC7	Interrupt 7		
0xC8	Interrupt 8		
0xC9	Interrupt 9		
0xCA	Interrupt 10		
0xCB	Interrupt 11		
0xCC	Interrupt 12		
0xCD	Interrupt 13		

Table 7-15 V60/61 exceptions (Continued)

Cause Code	Event Type	Event Description	Notes
0xCE	Interrupt 14	General external interrupt	Maskable
0xCF	Interrupt 15		
0xD0	Interrupt 16		
0xD1	Interrupt 17		
0xD2	Interrupt 18		
0xD3	Interrupt 19		
0xD4	Interrupt 20		
0xD5	Interrupt 21		
0xD6	Interrupt 22		
0xD7	Interrupt 23		
0xD8	Interrupt 24		
0xD9	Interrupt 25		
0xDA	Interrupt 26		
0xDB	Interrupt 27		
0xDC	Interrupt 28		
0xDD	Interrupt 29		
0xDE	Interrupt 30	General external interrupt.	Maskable, lowest priority internal interrupt
0xDF	Interrupt 31	General external interrupt.	Maskable, lowest priority interrupt reserved for L2 Vectored Interrupt

8 Software Stack

8.1 Overview

The Hexagon processor includes dedicated registers and instructions to support a *call stack* for subroutine execution.

The stack structure follows standard C conventions.

8.2 Stack structure

Figure 8-1 shows the stack structure.

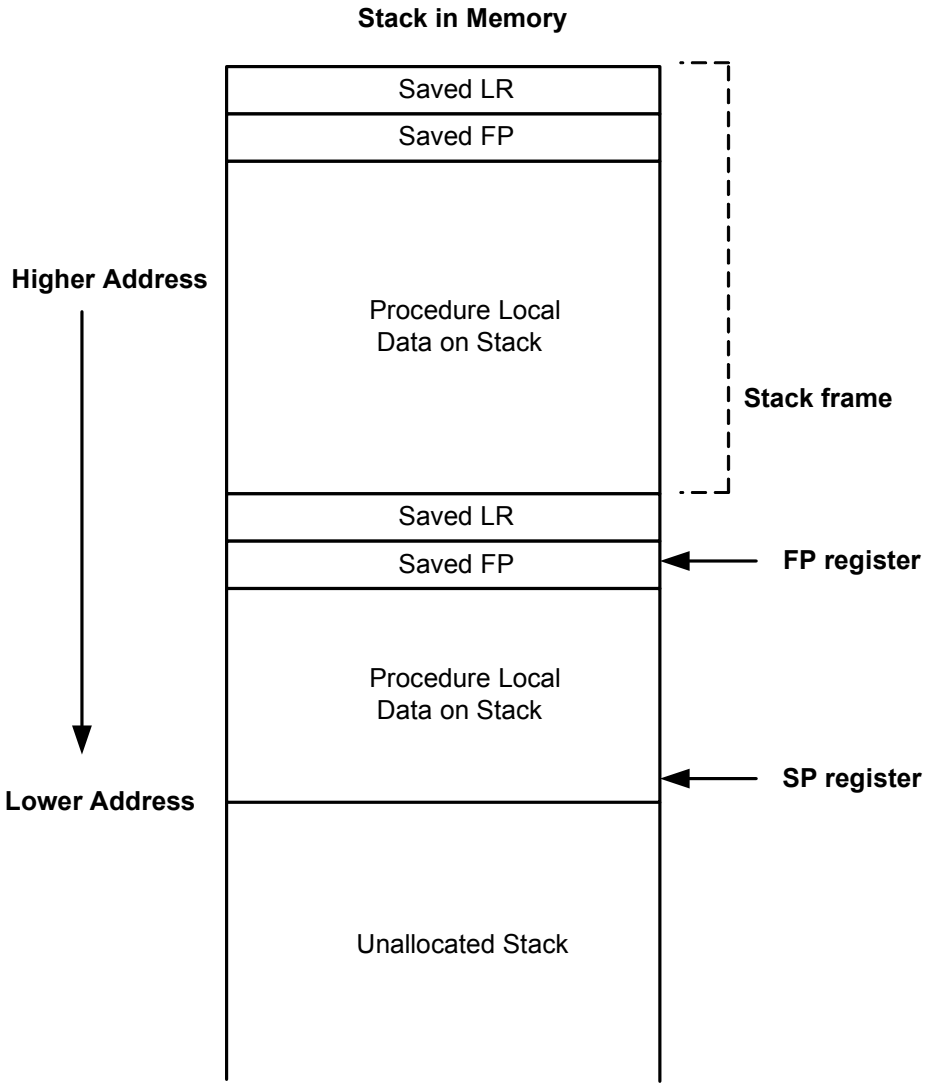


Figure 8-1 Stack structure

The stack is defined to grow from high addresses to low addresses. The stack pointer register `SP` points to the data element that is currently on the top of the stack.

NOTE The Hexagon processor supports three dedicated stack instructions: `allocframe`, `deallocframe`, and `dealloc_return` (Section 8.6).

The `SP` address must always remain 8-byte aligned for the stack instructions to work properly.

8.3 Stack frames

The stack is used to store *stack frames*, which are data structures that store state information on the active subroutines in a program (i.e., those that were called but have not yet returned). Each stack frame corresponds to an active subroutine in the program.

A stack frame contains the following elements:

- The local variables and data used by the subroutine
- The return address for the subroutine call (pushed from the link register `LR`)
- The address of the previous stack frame allocated on the stack (pushed from the frame pointer register `FP`)

The frame pointer register `FP` always contains the address of the saved frame pointer in the current stack frame. It facilitates debugging by enabling a debugger to examine the stack in memory and easily determine the call sequence, function parameters, etc.

NOTE For leaf functions it is often unnecessary to save `FP` and `LR`. In this case `FP` contains the frame pointer of the calling function, not the current function.

8.4 Stack protection

The Hexagon V6x processor supports the following features to protect the integrity of the software stack:

- Stack bounds checking
- Stack smashing protection

8.4.1 Stack bounds checking

Stack bounds checking prevents a stack frame from being allocated past the lower boundary of the software stack.

`FRAMELIMIT` is a 32-bit control register which is used to store a memory address that specifies the lower bound of the memory area reserved for the software stack. When the `allocframe` instruction allocates a new stack frame, it compares the new stack pointer value in `SP` with the stack bound value in `FRAMELIMIT`. If `SP` is less than `FRAMELIMIT`, the Hexagon processor raises exception 0x27 (Section 7.11).

NOTE Stack bounds checking is performed when the processor is in user and guest modes, but not in monitor mode.

8.4.2 Stack smashing protection

Stack smashing is a technique used by malicious code to gain control over an executing program. Malicious code causes buffer overflows to occur in a procedure's local data, with the goal of modifying the subroutine return address stored in a stack frame so it points to the malicious code instead of the intended return code.

Stack smashing protection prevents this from happening by scrambling the subroutine return address whenever a new stack frame is allocated, and then unscrambling the return address when the frame is deallocated. Because the value in FRAMEKEY changes regularly and varies from device to device, it becomes difficult to pre-calculate a malicious return address.

FRAMEKEY is a 32-bit control register which is used to scramble return addresses stored on the stack:

- In the allocframe instruction, the 32-bit return address in link register LR is XOR-scrambled with the value in FRAMEKEY before it is stored in the new stack frame.
- In deallocframe and dealloc_return, the return address loaded from the stack frame is unscrambled with the value in FRAMEKEY before it is stored in LR.

After a processor reset, the default value of FRAMEKEY is 0. If this value is not changed, stack smashing protection is effectively disabled.

NOTE Each hardware thread has its own instance of the FRAMEKEY register.

8.5 Stack registers

Table 8-1 lists the stack registers.

Table 8-1 Stack registers

Register	Name	Description	Alias
SP	Stack pointer	Points to topmost stack element in memory	R29
FP	Frame pointer	Points to previous stack frame on stack	R30
LR	Link register	Contains return address of subroutine call	R31
FRAMELIMIT	Frame limit register	Contains lowest address of stack area	C16
FRAMEKEY	Frame key register	Contains scrambling key for return addresses	C17

NOTE SP, FP, and LR are aliases of three general registers (Section 2.2). These general registers are conventionally dedicated for use as stack registers.

8.6 Stack instructions

The Hexagon processor includes the instructions `allocframe` and `deallocframe` to efficiently allocate and deallocate stack frames on the call stack.

Table 8-2 describes these instructions.

Table 8-2 Stack instructions

Syntax	Operation
allocframe(#u11:3)	Allocate stack frame. This instruction is used after a call. It first XORs the values in LR and FRAMEKEY, and pushes the resulting scrambled return address and FP to the top of stack. Next, it subtracts an unsigned immediate from SP in order to allocate room for local variables. If the resulting SP is less than FRAMELIMIT, the processor raises exception 0x27. Otherwise, SP is set to the new value, and FP is set to the address of the old frame pointer on the stack. The immediate operand as expressed in assembly syntax specifies the byte offset. This value must be 8-byte aligned. The valid range is from 0 to 16 KB.
deallocframe	Deallocate stack frame. This instruction is used before a return in order to free a stack frame. It first loads the saved FP and LR values from the address at FP, and XORs the restored LR with the value in FRAMEKEY to unscramble the return address. SPis then pointed back to the previous frame.
dealloc_return	Subroutine return with stack frame deallocate. Perform `deallocframe` operation, and then perform subroutine return (Section 7.4.3) to the target address loaded from LR by `deallocframe`.

NOTE `allocframe` and `deallocframe` load and store the `LR` and `FP` registers on the stack as a single aligned 64-bit register pair (i.e., `LR:FP`).

9 PMU Events

9.1 Overview

The Hexagon processor can collect execution statistics on the applications it executes. The statistics summarize the various types of Hexagon processor events that occurred while the application was running.

Execution statistics can be collected in hardware or software:

- Statistics can be collected in hardware with the Performance Monitor Unit (PMU), which is defined as part of the Hexagon processor architecture.
- Statistics can be collected in software using the Hexagon simulator. The simulator statistics are presented in the same format used by the PMU.

Execution statistics are expressed in terms of processor events. This chapter defines the event symbols, along with their associated numeric codes.

NOTE Because the types of execution events vary across the Hexagon processor versions, different types of statistics are collected for each version. This chapter lists the event symbols defined for versions V60 and V61.

9.2 V60 processor event symbols

Table 9-1 defines the symbols that are used to represent processor events for the V60 Hexagon processor.

Table 9-1 V60 processor event symbols

Event	Symbol	Definition
0x00	No	Do nothing. This event never causes a counter update.
0x01	COUNTER0_OVERFLOW	Counter0 overflow. This can be used as the event detected by counter1 to build an effective 64-bit counter.
0x02	COUNTER2_OVERFLOW	Counter2 overflow. This can be used as the event detected by counter3 to build an effective 64-bit counter.
0x03	COMMITTED_PKT_ANY	Thread committed a packet. Packets executed.
0x04	COMMITTED_PKT_BSB	Packet committed 2 cycles after previous packet in same thread.
0x05	COUNTER4_OVERFLOW	Counter4 overflow. This can be used as the event detected by counter5 to build an effective 64-bit counter.
0x06	COUNTER6_OVERFLOW	Counter6 overflow. This can be used as the event detected by counter7 to build an effective 64-bit counter.
0x07	COMMITTED_PKT_B2B	Packet committed 1 cycle after previous packet in same thread.
0x08	COMMITTED_PKT_SMT	2 packets committed in the same cycle.
0x11	ICACHE_DEMAND_MISS_CYCLES	Cycles for the stall that event 0x12 creates. Counts in thread cycles when PMUCFG[9] is 0, in processor cycles when it is 1.
0x12	ICACHE_DEMAND_MISS	Icache demand miss. Includes secondary miss and dczero. Excludes uncacheables, prefetch initiated misses, and predicate-killed operations.
0x13	DCACHE_DEMAND_MISS	Dcache cacheable demand primary or secondary miss. Includes dczero stall. Excludes uncacheables, prefetches, and no allocate store misses.
0x18	ARCH_LOCK_TVIEW_CYCLES	Cycles thread could not dispatch due to kernel lock or TLB lock.
0x19	IU_NO_PKT_TVIEW_CYCLES	Cycles thread could not dispatch due to IQ being empty.
0x1a	IU_BRANCH_MISS_TVIEW_CYCLES	Packets in flight killed due to branch mispredict and cycles thread could not dispatch due to IQ being empty because of branch mispredict.
0x1b	DU_CACHE_MISS_TVIEW_CYCLES	Cycles thread could not dispatch due to D-cache miss or uncached access.
0x1c	DU_BUSY_OTHER_TVIEW_CYCLES	Packets in flight killed due to DU replay and cycles thread could not dispatch due to DU busy because of stall or replay.
0x1d	CU_BUSY_TVIEW_CYCLES	Cycles thread could not dispatch due to register interlock, register port conflict, bubbles due to timing class such as tc_3stall, no B2B HVX, or HVX FIFO full.
0x1e	SMT_DU_CONFLICT_TVIEW_ CYCLES	Cycles thread could not dispatch even though the thread was ready due to DU resource conflict.
0x1f	SMT_XU_CONFLICT_TVIEW_ CYCLES	Cycles thread could not dispatch even though the thread was ready due to an XU resource conflict or other non-DU reason.
0x20	ANY_IU_REPLAY	Any IU stall other than icache miss. Includes jump register stall, fetchcross stall, itlb miss stall, etc. Excludes CU replay.

Table 9-1 V60 processor event symbols

Event	Symbol	Definition
0x21	ANY_DU_REPLAY	Any DU replay. Bank conflict, dTLB miss, etc. Excludes stalls due to cache misses.
0x22	CU_EARLY_CANCEL	Any case where a packet was dispatched prematurely based on incomplete information and the packet must be rescheduled after two cycles. This includes incomplete bypass network and multiple producer scenarios.
0x25	COMMITTED_PKT_1_THREAD_ RUNNING	Committed packets with 1 thread running. Not in stop/wait.
0x26	COMMITTED_PKT_2_THREAD_ RUNNING	Committed packets with 2 threads running. Not in stop/wait.
0x27	COMMITTED_PKT_3_THREAD_ RUNNING	Committed packets with 3 threads running. Not in stop/wait.
0x2a	COMMITTED_INSTS	Committed instructions. Increments by up to 8 per cycle. Duplex counts as two instructions. Does not include endloops.
0x2b	COMMITTED_TC1_INSTS	Committed tc1 class instructions. Increments by up to 8 per cycle. Duplex of two tc1 instructions counts as two tc1 instructions.
0x2c	COMMITTED_PRIVATE_INSTS	Committed instructions that have per-cluster (private) execution resources. Increments by up to 8 per cycle. Duplex of two private instructions counts as two private instructions.
0x2f	COMMITTED_PKT_4_THREAD_ RUNNING	Committed packets with 4 threads running. Not in stop/wait.
0x30	COMMITTED_LOADS	Committed load instructions. Includes cached and uncached. Increments by 2 for dual loads. Excludes prefetches, memops.
0x31	COMMITTED_STORES	Committed store instructions. Includes cached and uncached. Increments by 2 for dual stores. Excludes memops.
0x32	COMMITTED_MEMOPS	Committed memop instructions. Cached or uncached.
0x37	COMMITTED_PROGRAM_FLOW_ INSTS	Committed packet contains program flow inst. Includes cr jumps, endloop, j, jr, dealloc_return, system/trap, superset of event 0x38. Dual jumps count as two.
0x38	COMMITTED_PKT_CHANGED_ FLOW	Committed packet resulted in change-of-flow. Any taken jump, includes endloop.
0x39	COMMITTED_PKT_ENDLOOP	Committed packet contains an endloop which was taken.
0x3b	CYCLES_1_THREAD_RUNNING	Processor cycles that 1 thread is running. Not in wait or stop.
0x3c	CYCLES_2_THREAD_RUNNING	Processor cycles that 2 threads are running. Not in wait or stop.
0x3d	CYCLES_3_THREAD_RUNNING	Processor cycles that 3 threads are running. Not in wait or stop.
0x3e	CYCLES_4_THREAD_RUNNING	Processor cycles that 4 threads are running. Not in wait or stop.
0x40	AXI_READ_REQUEST	All read requests issued by primary AXI master. Includes full line and partial line.
0x41	AXI_LINE32_READ_REQUEST	32-byte line read requests issued by primary AXI master.
0x42	AXI_WRITE_REQUEST	All write requests issued by primary AXI master. Includes full line and partial line.
0x43	AXI_LINE32_WRITE_REQUEST	32-byte line write requests issued by primary AXI master, all bytes valid.

Table 9-1 V60 processor event symbols

Event	Symbol	Definition
0x44	AHB_READ_REQUEST	Read requests issued by AHB master.
0x45	AHB_WRITE_REQUEST	Write requests issued by AHB master.
0x47	AXI_SLAVE_MULTI_BEAT_ACCESS	AXI slave multi-beat access.
0x48	AXI_SLAVE_SINGLE_BEAT_ ACCESS	AXI slave single-beat access.
0x49	AXI2_READ_REQUEST	All read requests issued by secondary AXI master. Includes full line and partial line.
0x4a	AXI2_LINE32_READ_REQUEST	32-byte line read requests issued by secondary AXI master
0x4b	AXI2_WRITE_REQUEST	All write requests issued by secondary AXI master. Includes full line and partial line.
0x4c	AXI2_LINE32_WRITE_REQUEST	32-byte line Write requests issued by secondary AXI master.
0x4d	AXI2_CONGESTION	Secondary AXI command or data queue is full and an operation is stuck at the head of the secondary AXI master command queue.
0x50	COMMITTED_FPS	Committed floating point instructions. Increments by 2 for dual FP ops. Excludes conversions.
0x51	BRANCH_MISPREDICT_DIRECTION	Branch mispredict direction. Includes new-predicate, old-predicate, low loop count, new-value, and dealloc_return direction mispredicts. Only counts once if both jumps in a dual jump mispredict. Excludes target mispredicts.
0x52	BRANCH_MISPREDICT_TARGET	Branch mispredict target. Includes RAS mispredict. Excludes indirect jumps and calls other than JUMPR R31 returns. Excludes direction mispredicts.
0x58	JTLB_MISS	Instruction or data address translation request missed in JTLB.
0x5a	COMMITTED_PKT_RETURN	Committed a return instruction. Includes canceled returns.
0x5b	COMMITTED_PKT_INDIRECT_JUMP	Committed an indirect jump or call instruction. Includes canceled instructions. Does not include JUMPR R31 returns.
0x5c	COMMITTED_BIMODAL_ BRANCH_INSTS	Committed bimodal branch. Includes .old and .new. Increments by 2 for dual jumps.
0x5d	BRANCH_QUEUE_FULL	Bimodal branch ignored if queue is full.
0x5e	SMT_XU_CONFLICT	The number of times an XU resource conflict was one of the reasons a packet could not dispatch.
0x66	IU_LINE_FROM_LOOP_BUFFER	IU line from loop buffer. Cacheline from loop cache.
0x68	IU_1_PKT_AVAILABLE_TO_ISSUE	Fetch that could issue one packet independent of available packet queue entries.
0x6a	IU_REQUESTS_TO_L2_REPLAYED	IU requests to L2 replayed. Could be IU and DU collision to L2 or IU credit failure. Could fire multiple times for a single transaction.
0x6b	IU_PREFETCHES_SENT_TO_L2	IU prefetches sent to L2. Counts cachelines not dropped by L2. Excludes replayed prefetches and only counts ones L2 accepts.
0x6c	ITLB_MISS	ITLB miss. One per thread maximum.
0x6d	IU_2_PKT_AVAILABLE_TO_ISSUE	Fetch that could issue two packets independent of available packet queue entries.
0x6e	IU_3_PKT_AVAILABLE_TO_ISSUE	Fetch that could issue three packets independent of available packet queue entries.

Table 9-1 V60 processor event symbols

Event	Symbol	Definition
0x6f	IU_REQUEST_STALLED	IU request stalled due to fill. Demand or prefetch fills from L2 to icache caused IU demand to stall.
0x70	BIMODAL_UPDATE_DROPPED	Bimodal update dropped. Cacheline way (in any set) got replaced before bimodal update so it is dropped.
0x71	IU_0_PKT_AVAILABLE_TO_ISSUE	Fetch which could only issue a partial packet due to a jump to a packet which crosses cachelines boundary.
0x72	IU_LINE_FALL_THROUGH	Front end fall through to new line. Does not include fall through from loop cached line. Helps calculate non-jump ibuf flushes.
0x73	IU_LEFTOVER_DROPPED	Dropped leftover cache line contents because it exceeds IBUF size.
0x74	L2FETCH_IU_ACCESS	L2fetch access from IU. Any access to the L2 cache from the L2 prefetch engine that was initiated by the IU. Access must be L2 cacheable and target primary.
0x75	L2FETCH_IU_MISS	Of the events in 0x74, L2Fetch_IU_Access, the ones that miss in L2.
0x76	L2_IU_ACCESS	L2 access from IU. Any access to L2 cache that was the result of an IU command, either demand or L1 prefetch access. Excludes L2 prefetches, TCM accesses, and uncacheables.
0x77	L2_IU_MISS	L2 misses from IU. Of the events qualified by 0x76, the ones that resulted in an L2 miss (demand miss or L1 prefetch miss). Includes hitting a reserved line.
0x78	L2_IU_PREFETCH_ACCESS	Pre-fetch from the IU to L2. Any IU pre-fetch access sent to the L2 cache. Access must be L2 cacheable and target primary AXI. This does not include L2Fetch generated accesses.
0x79	L2_IU_PREFETCH_MISS	L2 prefetch from IU miss. Of the events qualified by 0x78, the ones that resulted in an L2 miss.
0x7c	L2_DU_READ_ACCESS	L2 read access from DU. Any read access from DU that may cause a lookup in the L2 cache. includes loads, L1 prefetch, dcfetch. excludes initial L2fetch command, uncacheables.
0x7d	L2_DU_READ_MISS	L2 read miss from DU. Of the events qualified by 0x7c, the ones that resulted in a miss and were attempts to load data. Includes prefetches. Excludes TCM.
0x7e	L2FETCH_ACCESS	L2fetch access from DU. Any access to the L2 cache from the L2 prefetch engine that was initiated by programing the L2Fetch engine. Includes only the cache inquire and fetch if not present commands.
0x7f	L2FETCH_MISS	L2fetch miss from a programed inquiry. Of the events qualified by 126, the ones that resulted in L2 miss. i.e. the line was not previously allocated in the L2 cache and will be fetched from backing memory.
0x80	L2_AXI_INTERLEAVE_DROP	L2 drops a return or converts to uncacheable due to AXI interleave buffer overflow.
0x81	L2_ACCESS	All requests to the L2. Does not include internally generated accesses like L2Fetch, however the programing of the L2Fetch engine would be counted. Accesses target odd or even interleave, and may be L2 cacheable or TCM.

Table 9-1 V60 processor event symbols

Event	Symbol	Definition
0x82	L2_PIPE_CONFLICT	Request not taken by the L2 due to a pipe conflict. The conflict may be a tag bank, data bank, or other pipeline conflict.
0x83	L2_TAG_ARRAY_CONFLICT	Tag array conflict. Triggers when L2 tag has a conflict *i.e.,* two operations collide.
0x84	AXI_RD_CONGESTION	Primary AXI read command queue is full and an operation is stuck at the head of the primary AXI master command queue.
0x85	AHB_CONGESTION	AHB congestion. Triggers when AHB interface is full and an operation is stuck at the head of the command queue.
0x87	TCM_DU_ACCESS	TCM access from DU. DU access to the L2 TCM space.
0x88	TCM_DU_READ_ACCESS	TCM read access from DU. DU read access to the L2 TCM space.
0x89	TCM_IU_ACCESS	TCM access from IU. IU access to the L2 TCM space.
0x8a	L2_CASTOUT	L2 castout. Triggers when L2 evicts a dirty line due to an allocation. Not triggered on cache ops.
0x8b	L2_DU_STORE_ACCESS	L2 cacheable store access from DU. Any store access from DU that may cause a lookup in the L2 cache. Excludes cache ops, uncacheables, TCM. Must target primary AXI.
0x8c	L2_DU_STORE_MISS	L2 miss from DU. Of the events qualified by 0x8b, the ones that resulted in a miss. Specifically the cases where the line is not in cache or a coalesce buffer.
0x8d	L2_DU_PREFETCH_ACCESS	L2 prefetch access from DU. Dcfetch and dhwprefetch access to the L2.
0x8e	L2_DU_PREFETCH_MISS	L2 prefetch miss from DU. Of the events qualified by 0x8d, which ones missed the L2.
0x8f	L2_DU_RETURN_NOT_ACKED	L2 return to DU not acknowledged. This is active for any DU return, prefetch or demand. This event was renamed. I was called L2_DU_PREFETCH_NOT_ACKED.
0x90	L2_DU_LOAD_SECONDARY_MISS	L2 load secondary miss from DU. Hit busy line in the scoreboard which prevented a return. Busy condition may include pipeline bubbles caused by back to back loads, as may be seen on L1UC loads.
0x91	L2FETCH_COMMAND	L2fetch command. Excludes stop command.
0x92	L2FETCH_COMMAND_KILLED	L2fetch command killed because a stop command was issued. Increments once for each L2fetch commands that is killed. If multiple commands are queued to the L2Fetch engine, the kill of each one will be recorded.
0x93	L2FETCH_COMMAND_OVERWRITE	L2fetch command overwrite. Kills old L2fetch and replaces with new one.
0x97	L2_ACCESS_EVEN	Of the events in 0x81, the number of accesses made to the even L2 cache.
0xa0	ANY_DU_STALL	Any DU stall. Increments once when the thread has any DU stall (dcache miss or dTLB miss).
0xa1	DU_BANK_CONFLICT_REPLAY	DU bank conflict replay. Dual mem access to same bank but different lines. Banks are determined by bits 11,5,4,3.
0xa3	L2_FIFO_FULL_REPLAY	Counts L2 even/odd FIFO full replays.

Table 9-1 V60 processor event symbols

Event	Symbol	Definition
0xa4	DU_STORE_BUFFER_FULL_ REPLAY	First packet puts access in DU store buffer (memop, store.new, load/store bank conflict, store/store bank conflict). Any later packet tries to use store buffer before the first one evicts, and so must replay so the store buffer can drain.
0xa8	DU_FILL_REPLAY	A fill has a index conflict with an instruction from the same thread in pipeline. Fills and demands might be from different threads if (1) prefetch from deferral queue, (2) store from store buffer, or (3) fill has not be ack'ed for too long and forces itself in to the pipeline.
0xac	DU_READ_TO_L2	DU read to L2. Total of everything that brings data from L2. Includes prefetches (dcfetch and hwprefetch).
0xad	DU_WRITE_TO_L2	DU write to L2. Total of everything that is written out of DU to the L2 array. Includes cleaninv. Excludes tag writes.
0xaf	DCZERO_COMMITTED	Committed a dczeroa instruction.
0xb3	DTLB_MISS	DTLB miss which goes to JTLB. When both slots miss to different pages, increments by 2. When both slots miss to the same page, only counts S1, since S1 goes first and fills for S0.
0xb6	STORE_BUFFER_HIT_REPLAY	Store buffer hit replay due to a packet with 2 stores going to the same bank but different cachelines, followed by a load from an address that was pushed into the store buffer.
0xb7	STORE_BUFFER_FORCE_REPLAY	The store buffer must drain, forcing the current packet to replay. This usually happens on an cache index match between the current packet and store buffer. Could also a store buffer timeout.
0xb9	SMT_BANK_CONFLICT	Counts inter-thread SMT bank conflicts.
0xba	SMT_PORT_CONFLICT	Counts inter-thread SMT port conflicts. This occurs when a packet on one cluster is replayed and goes to DU at the same time as a packet on the other cluster such that the total number of memory accesses in those two packets exceeds the number of DU ports.
0xbd	PAGE_CROSS_REPLAY	Page cross replay. Page cross from valid packet that caused replay. Excludes pdkill packets. Counts twice if both slots cause a page cross.
0xbf	DU_DEMAND_SECONDARY_MISS	DU demand secondary miss.
0xc3	DCFETCH_COMMITTED	Dcfetch committed. Includes hit and dropped. Does not include convert-to-prefetches.
0xc4	DCFETCH_HIT	Dcfetch hit in dcache. Includes hit valid or reserved line.
0xc5	DCFETCH_MISS	Dcfetch missed in L1. Counts the dcfetch issued to L2 FIFO.
0xc8	DU_LOAD_UNCACHEABLE	Uncacheable load in L1 cache. Counts twice for dual uncacheable loads.
0xc9	DU_DUAL_LOAD_UNCACHEABLE	Dual uncacheable loads in L1 cache.
0xca	DU_STORE_UNCACHEABLE	Uncacheable store in L1 cache. Counts twice for dual uncacheable stores.
0xcc	MISS_TO_PREFETCH	Dcache miss is converted to a prefetch. Could be the second load to miss in a packet, or a DMT subsequent packet on same thread. The normal dcache access will proceed when the first miss resolves. Includes converted prefetches into uncacheable space which will be cancelled later in the pipe.
0xce	AXI_LINE64_READ_REQUEST	64-byte line read requests issued by primary AXI master.

Table 9-1 V60 processor event symbols

Event	Symbol	Definition
0xcf	AXI_LINE64_WRITE_REQUEST	64-byte line write requests issued by primary AXI master. All bytes valid.
0xd0	AXI_WR_CONGESTION	Primary AXI write command or data queue is full and an operation is stuck at the head of the primary AXI master command queue.
0xd1	AHB_8_READ_REQUEST	An 8-byte AHB read was made.
0xd2	AXI_INCOMPLETE_WRITE_ REQUEST	A burst write was made by the primary AXI master, but not all of the bytes were valid.
0xd3	L2FETCH_COMMAND_PAGE_ TERMINATION	L2fetch command terminated because it could not get a page translation from VA to PA. Includes termination dues to permission errors. i.e. address translation could fail because the VA to PA is not in the TLB, or the properties in the translation are not acceptable and the command terminates.
0xd4	REQUEST_STALL_WRITE_ BUFFER_EXHAUSTION	Request to L2 stalled due to the lack of a write buffer. This is set for scalar and vector stores.
0xd5	L2_DU_STORE_COALESCE	Events from 0x8b that were coalesced.
0xdc	REQUEST_STALL_EVICTION_ BUFFER_EXHAUSTION	Request to L2 stalled due to the lack of an eviction buffer. This is set for scalar and vector loads.
0xdd	AHB_MULTI_BEAT_READ_ REQUEST	A 32-byte multi-beat AHB read was made.
0xdf	L2_DU_LOAD_SECONDARY_MISS_ ON_SW_PREFETCH	Of the events in 0x90, the ones where the primary miss was a DCFETCH or L2FETCH.
0xe6	ARCH_LOCK_PVIEW_CYCLES	Cycles cluster could not dispatch due to kernel lock or TLB lock.
0xe7	IU_NO_PKT_PVIEW_CYCLES	Cycles cluster could not dispatch due to IQ being empty.
0xe8	IU_BRANCH_MISS_PVIEW_CYCLES	Packets in flight killed due to branch mispredict and cycles cluster could not dispatch due to IQ being empty because of branch mispredict.
0xe9	DU_CACHE_MISS_PVIEW_CYCLES	Cycles cluster could not dispatch due to D-cache miss or uncached access.
0xea	DU_BUSY_OTHER_PVIEW_CYCLES	Packets in flight killed due to DU replay and cycles cluster could not dispatch due to DU busy because of stall or replay.
0xeb	CU_BUSY_PVIEW_CYCLES	Cycles cluster could not dispatch due to register interlock, register port conflict, bubbles due to timing class such as tc_3stall, no B2B HVX, or HVX FIFO full.
0xec	SMT_DU_CONFLICT_PVIEW_ CYCLES	Cycles cluster could not dispatch even though the thread was ready due to DU resource conflict.
0xed	SMT_XU_CONFLICT_PVIEW_ CYCLES	Cycles cluster could not dispatch even though the cluster was ready due to an XU resource conflict or other non-DU reason.
0xef	COPROC_L2_STORE_TO_LOAD_ MISS	Of the events in 0xfb, the ones where the load could not be returned due to the immediately prior access for the line being a pending coprocessor store.
oxf0	COPROC_PKT_THREAD	Committed packets on a thread with XE bit set, whether executed in core or coprocessor.
oxf1	COPROC_PKT_COUNT	Packets with HVX instructions.
oxf2	COPROC_POWER_THROTTLING_ STALL_CYCLES	Stall cycles due to HVX engine being throttled for power reasons.

Table 9-1 V60 processor event symbols

Event	Symbol	Definition
oxf3	COPROC_REGISTER_STALL_ CYCLES	Stall cycles due to interlocks.
oxf4	COPROC_LOAD_STALL_CYCLES	Stall cycles due to load pending.
oxf5	COPROC_STORE_STALL_CYCLES	Stall cycles due to store not yet allocated in L2.
oxf6	COPROC_BUSY_STALL_CYCLES	Stall cycles due to multi-issue packet.
oxf7	COPROC_FREEZE_STALL_CYCLES	Stall cycles due to L2 vector store data FIFO full.
oxf8	COPROC_CORE_VFIFO_FULL_ STALL	Number of times a thread had to stall due to VFIFO being full.
oxf9	COPROC_L2_STORE_ACCESS	L2 cacheable store access from HVX. Any store access from HVX that may cause a lookup in L2 cache. Excludes cache ops, uncacheables, scalar stores.
oxfa	COPROC_L2_STORE_MISS	L2 cacheable miss from HVX. Of the events qualified by 0xf9, the ones that resulted in a miss. Specifically the cases where line is not in cache or a coalesce buffer. DCZero does not count as a miss, if it allocates without a backing store fetch.
oxfb	COPROC_L2_LOAD_ACCESS	L2 cacheable load access from HVX. Any load access from HVX that may cause a lookup in the L2 cache. Excludes cache ops, uncacheables, scalars.
oxfc	COPROC_L2_LOAD_MISS	L2 cacheable miss from HVX. Of the events qualified by 0xFB, the ones that resulted in a miss i.e. the line was not previously allocated in the L2 cache and will be fetched from backing memory.
oxfd	COPROC_TCM_STORE_ACCESS	TCM store access for HVX. HVX store to L2 TCM space.
oxfe	COPROC_TCM_LOAD_ACCESS	TCM load access for HVX. HVX load from L2 TCM space
oxff	COPROC_L2_LOAD_SECONDARY_ MISS	Of the events in 0xfb, the ones where the load could not be returned due to the immediately prior access for the line being a pending load or pending L2Fetch.

9.3 V61 processor event symbols

Table 9-2 defines the symbols that are used to represent processor events for the V61 Hexagon processor.

Table 9-2 V61 processor event symbols

Event	Symbol	Definition
0x00	No	Do nothing. This event never causes a counter update.
0x01	COUNTER0_OVERFLOW	Counter0 overflow. This can be used as the event detected by counter1 to build an effective 64-bit counter.
0x02	COUNTER2_OVERFLOW	Counter2 overflow. This can be used as the event detected by counter3 to build an effective 64-bit counter.
0x03	COMMITTED_PKT_ANY	Thread committed a packet. Packets executed.
0x04	COMMITTED_PKT_BSB	Packet committed 2 cycles after previous packet in same thread.
0x05	COUNTER4_OVERFLOW	Counter4 overflow. This can be used as the event detected by counter5 to build an effective 64-bit counter.
0x06	COUNTER6_OVERFLOW	Counter6 overflow. This can be used as the event detected by counter7 to build an effective 64-bit counter.
0x07	COMMITTED_PKT_B2B	Packet committed 1 cycle after previous packet in same thread.
0x08	COMMITTED_PKT_SMT	2 packets committed in the same cycle.
0x12	ICACHE_DEMAND_MISS	Icache demand miss. Includes secondary miss and dczero. Excludes uncacheables, prefetch initiated misses, and predicate-killed operations.
0x13	DCACHE_DEMAND_MISS	Dcache cacheable demand primary or secondary miss. Includes dczero stall. Excludes uncacheables, prefetches, and no allocate store misses.
0x14	DCACHE_STORE_MISS	Dcache cacheable store miss.
0x17	CU_PKT_READY_NOT_DISPATCHED	Packet ready at scheduler but not scheduled because a) its thread was not picked or b) an intercluster resource conflict occurred.
0x20	ANY_IU_REPLAY	Any IU stall other than icache miss. Includes jump register stall, fetchcross stall, itlb miss stall, etc. Excludes CU replay.
0x21	ANY_DU_REPLAY	Any DU replay. Bank conflict, dTLB miss, etc. Excludes stalls due to cache misses.
0x25	COMMITTED_PKT_1_THREAD_ RUNNING	Committed packets with 1 thread running. Not in stop/wait.
0x26	COMMITTED_PKT_2_THREAD_ RUNNING	Committed packets with 2 threads running. Not in stop/wait.
0x27	COMMITTED_PKT_3_THREAD_ RUNNING	Committed packets with 3 threads running. Not in stop/wait.
0x2a	COMMITTED_INSTS	Committed instructions. Increments by up to 8 per cycle. Duplex counts as two instructions. Does not include endloops.
0x2b	COMMITTED_TC1_INSTS	Committed tc1 class instructions. Increments by up to 8 per cycle. Duplex of two tc1 instructions counts as two tc1 instructions.
0x2c	COMMITTED_PRIVATE_INSTS	Committed instructions that have per-cluster (private) execution resources. Increments by up to 8 per cycle. Duplex of two private instructions counts as two private instructions.

Table 9-2 V61 processor event symbols

Event	Symbol	Definition
0x2f	COMMITTED_PKT_4_THREAD_ RUNNING	Committed packets with 4 threads running. Not in stop/wait.
0x30	COMMITTED_LOADS	Committed load instructions. Includes cached and uncached. Increments by 2 for dual loads. Excludes prefetches, memops.
0x31	COMMITTED_STORES	Committed store instructions. Includes cached and uncached. Increments by 2 for dual stores. Excludes memops.
0x32	COMMITTED_MEMOPS	Committed memop instructions. Cached or uncached.
0x37	COMMITTED_PROGRAM_FLOW_ INSTS	Committed packet contains program flow inst. Includes cr jumps, endloop, j, jr, dealloc_return, system/trap, superset of event 0x38. Dual jumps count as two.
0x38	COMMITTED_PKT_CHANGED_ FLOW	Committed packet resulted in change-of-flow. Any taken jump, includes endloop.
0x39	COMMITTED_PKT_ENDLOOP	Committed packet contains an endloop which was taken.
0x3b	CYCLES_1_THREAD_RUNNING	Processor cycles that 1 thread is running. Not in wait or stop.
0x3c	CYCLES_2_THREAD_RUNNING	Processor cycles that 2 threads are running. Not in wait or stop.
0x3d	CYCLES_3_THREAD_RUNNING	Processor cycles that 3 threads are running. Not in wait or stop.
0x3e	CYCLES_4_THREAD_RUNNING	Processor cycles that 4 threads are running. Not in wait or stop.
0x40	AXI_READ_REQUEST	All read requests issued by primary AXI master. Includes full line and partial line.
0x41	AXI_LINE32_READ_REQUEST	32-byte line read requests issued by primary AXI master.
0x42	AXI_WRITE_REQUEST	All write requests issued by primary AXI master. Includes full line and partial line.
0x43	AXI_LINE32_WRITE_REQUEST	32-byte line write requests issued by primary AXI master, all bytes valid.
0x44	AHB_READ_REQUEST	Read requests issued by AHB master.
0x45	AHB_WRITE_REQUEST	Write requests issued by AHB master.
0x47	AXI_SLAVE_MULTI_BEAT_ACCESS	AXI slave multi-beat access.
0x48	AXI_SLAVE_SINGLE_BEAT_ ACCESS	AXI slave single-beat access.
0x49	AXI2_READ_REQUEST	All read requests issued by secondary AXI master. Includes full line and partial line.
0x4a	AXI2_LINE32_READ_REQUEST	32-byte line read requests issued by secondary AXI master
0x4b	AXI2_WRITE_REQUEST	All write requests issued by secondary AXI master. Includes full line and partial line.
0x4c	AXI2_LINE32_WRITE_REQUEST	32-byte line write requests issued by secondary AXI master.
0x4d	AXI2_CONGESTION	Secondary AXI command or data queue is full and an operation is stuck at the head of the secondary AXI master command queue.
0x50	COMMITTED_FPS	Committed floating point instructions. Increments by 2 for dual FP ops. Excludes conversions.

Table 9-2 V61 processor event symbols

Event	Symbol	Definition
0x51	BRANCH_MISPREDICT_DIRECTION	Branch mispredict direction. Includes new-predicate, old-predicate, low loop count, new-value, and dealloc_return direction mispredicts. Only counts once if both jumps in a dual jump mispredict. Excludes target mispredicts.
0x52	BRANCH_MISPREDICT_TARGET	Branch mispredict target. Includes RAS mispredict. Excludes indirect jumps and calls other than JUMPR R31 returns. Excludes direction mispredicts.
0x58	JTLB_MISS	Instruction or data address translation request missed in JTLB.
0x5a	COMMITTED_PKT_RETURN	Committed a return instruction. Includes canceled returns.
0x5b	COMMITTED_PKT_INDIRECT_JUMP	Committed an indirect jump or call instruction. Includes canceled instructions. Does not include JUMPR R31 returns.
0x5c	COMMITTED_BIMODAL_ BRANCH_INSTS	Committed bimodal branch. Includes .old and .new. Increments by 2 for dual jumps.
0x5d	BRANCH_QUEUE_FULL	Bimodal branch ignored if queue is full.
0x5e	CU_DU_REQUESTED_BUBBLE_ INSERTED	The number of bubbles CU inserted at the request of DU in an effort to avoid DU port conflicts.
0x65	IU_DEMAND_SECONDARY_MISS	Icache secondary miss.
0x66	IU_LINE_FROM_LOOP_BUFFER	IU line from loop buffer. Cacheline from loop cache.
0x68	IU_1_PKT_AVAILABLE_TO_ISSUE	Fetch that could issue one packet independent of available packet queue entries.
0x6a	IU_REQUESTS_TO_L2_REPLAYED	IU requests to L2 replayed. Could be IU and DU collision to L2 or IU credit failure. Could fire multiple times for a single transaction.
0x6b	IU_PREFETCHES_SENT_TO_L2	IU prefetches sent to L2. Counts cachelines not dropped by L2. Excludes replayed prefetches and only counts ones L2 accepts.
0x6c	ITLB_MISS	ITLB miss. One per thread maximum.
0x6d	IU_2_PKT_AVAILABLE_TO_ISSUE	Fetch that could issue two packets independent of available packet queue entries.
0x6e	IU_3_PKT_AVAILABLE_TO_ISSUE	Fetch that could issue three packets independent of available packet queue entries.
0x6f	IU_REQUEST_STALLED	IU request stalled due to fill. Demand or prefetch fills from L2 to icache caused IU demand to stall.
0x70	IU_BIMODAL_L2_ELIGIBLE	Counts the number of branches which resolved can go to L2.
0x71	IU_0_PKT_AVAILABLE_TO_ISSUE	Fetch which could only issue a partial packet due to a jump to a packet which crosses cachelines boundary.
0x76	L2_IU_ACCESS	L2 access from IU. Any access to L2 cache that was the result of an IU command, either demand or L1 prefetch access. Excludes L2 prefetches, TCM accesses, and uncacheables.
0x77	L2_IU_MISS	L2 misses from IU. Of the events qualified by 0x76, the ones that resulted in an L2 miss (demand miss or L1 prefetch miss). Includes hitting a reserved line.
0x78	L2_IU_PREFETCH_ACCESS	Pre-fetch from the IU to L2. Any IU pre-fetch access sent to the L2 cache. Access must be L2 cacheable and target primary AXI. This does not include L2Fetch generated accesses.

Table 9-2 V61 processor event symbols

Event	Symbol	Definition
0x79	L2_IU_PREFETCH_MISS	L2 prefetch from IU miss. Of the events qualified by 0x78, the ones that resulted in an L2 miss.
0x7a	L2_IU_BRANCH_CACHE_WRITE_ REQUEST	Requests sent from IU to L2 to write the bimodal bits of instructions in L2 cache. Includes all requests, without regard to target.
0x7b	L2_IU_BRANCH_CACHE_WRITE	Writes to the bimodal bits of instructions in L2 cache. This is the number of event 0x7a requests that completed by updating memory in the L2 cache or TCM.
0x7c	L2_DU_READ_ACCESS	L2 read access from DU. Any read access from DU that may cause a lookup in the L2 cache. includes loads, L1 prefetch, dcfetch. excludes initial L2fetch command, uncacheables.
0x7d	L2_DU_READ_MISS	L2 read miss from DU. Of the events qualified by 0x7c, the ones that resulted in a miss and were attempts to load data. Includes prefetches. Excludes TCM.
0x7e	L2FETCH_ACCESS	L2fetch access from DU. Any access to the L2 cache from the L2 prefetch engine that was initiated by programing the L2Fetch engine. Includes only the cache inquire and fetch if not present commands.
0x7f	L2FETCH_MISS	L2fetch miss from a programed inquiry. Of the events qualified by 0x7e, the ones that resulted in L2 miss. i.e. the line was not previously allocated in the L2 cache and will be fetched from backing memory.
0x80	L2_AXI_INTERLEAVE_DROP	L2 drops a return or converts to uncacheable due to AXI interleave buffer overflow.
0x81	L2_ACCESS	All requests to the L2. Does not include internally generated accesses like L2Fetch, however the programing of the L2Fetch engine would be counted. Accesses target odd or even interleave, and may be L2 cacheable or TCM.
0x82	L2_PIPE_CONFLICT	Request not taken by the L2 due to a pipe conflict. The conflict may be a tag bank, data bank, or other pipeline conflict.
0x83	L2_TAG_ARRAY_CONFLICT	Tag array conflict. Triggers when L2 tag has a conflict *i.e.,* two operations collide.
0x84	AXI_RD_CONGESTION	Primary AXI read command queue is full and an operation is stuck at the head of the primary AXI master command queue.
0x85	AHB_CONGESTION	AHB congestion. Triggers when AHB interface is full and an operation is stuck at the head of the command queue.
0x86	SNOOP_BLOCK	Snoop block. Triggers when a snoop is not accepted and the snoop is blocking the head of the return FIFO, and the return FIFO has other items.
0x87	TCM_DU_ACCESS	TCM access from DU. DU access to the L2 TCM space.
0x88	TCM_DU_READ_ACCESS	TCM read access from DU. DU read access to the L2 TCM space.
0x89	TCM_IU_ACCESS	TCM access from IU. IU access to the L2 TCM space.
0x8a	L2_CASTOUT	L2 castout. Triggers when L2 evicts a dirty line due to an allocation. Not triggered on cache ops.
0x8b	L2_DU_STORE_ACCESS	L2 cacheable store access from DU. Any store access from DU that may cause a lookup in the L2 cache. Excludes cache ops, uncacheables, TCM. Must target primary AXI.

Table 9-2 V61 processor event symbols

Event	Symbol	Definition
0x8c	L2_DU_STORE_MISS	L2 miss from DU. Of the events qualified by 0x8b, the ones that resulted in a miss. Specifically the cases where the line is not in cache or a coalesce buffer.
0x8d	L2_DU_PREFETCH_ACCESS	L2 prefetch access from DU. Dcfetch and dhwprefetch access to the L2.
0x8e	L2_DU_PREFETCH_MISS	L2 prefetch miss from DU. Of the events qualified by 0x8d, which ones missed the L2.
0x8f	L2_DU_RETURN_NOT_ACKED	L2 return to DU not acknowledged. This is active for any DU return, prefetch or demand. This event was renamed. I was called L2_DU_PREFETCH_NOT_ACKED.
0x90	L2_DU_LOAD_SECONDARY_MISS	L2 load secondary miss from DU. Hit busy line in the scoreboard which prevented a return. Busy condition may include pipeline bubbles caused by back to back loads, as may be seen on L1UC loads.
0x91	L2FETCH_COMMAND	L2fetch command. Excludes stop command.
0x92	L2FETCH_COMMAND_KILLED	L2fetch command killed because a stop command was issued. Increments once for each L2fetch commands that is killed. If multiple commands are queued to the L2Fetch engine, the kill of each one will be recorded.
0x93	L2FETCH_COMMAND_OVERWRITE	L2fetch command overwrite. Kills old L2fetch and replaces with new one.
0x94	L2FETCH_ACCESS_CREDIT_FAIL	L2fetch access could not get a credit. L2fetch blocked because of missing L2fetch or L2evict credit.
0xa0	ANY_DU_STALL	Any DU stall. Increments once when the thread has any DU stall (dcache miss or dTLB miss).
0xa1	DU_BANK_CONFLICT_REPLAY	DU bank conflict replay. Dual mem access to same bank but different lines. Banks are determined by bits 11,5,4,3.
0xa2	DU_CREDIT_REPLAY	Number of times packet took a replay because insufficient QoS DU credits were available.
0xa3	L2_FIFO_FULL_REPLAY	Counts L2 even/odd FIFO full replays.
0xa4	DU_STORE_BUFFER_FULL_ REPLAY	First packet puts access in DU store buffer (memop, store.new, load/store bank conflict, store/store bank conflict). Any later packet tries to use store buffer before the first one evicts, and so must replay so the store buffer can drain.
0xa5	DU_DIRECT_WRITE	Number of times a store wrote to cache directly, without going through store buffer.
0xa6	DU_SNOOP_CONFLICT_REPLAY	DU snoop conflict replay. A snoop occurs during a set-based match with stores in-flight, so the store is replayed. The snoop reads and writes the cache, and needs to finish before the store can write. Excludes the fill-replay case that a snoop can cause.
0xa7	DU_SNOOP_REQUEST	Number of DU snoop requests that were accepted.
0xa8	DU_FILL_REPLAY	A fill has a index conflict with an instruction from the same thread in pipeline. Fills and demands might be from different threads if (1) prefetch from deferral queue, (2) store from store buffer, or (3) fill has not be ack'ed for too long and forces itself in to the pipeline.
0xaa	DU_SNOOP_REQUEST_CLEAN_HIT	DU snoop request clean hit. Snoop hitting a clean line.

Table 9-2 V61 processor event symbols

Event	Symbol	Definition
0xac	DU_READ_TO_L2	DU read to L2. Total of everything that brings data from L2. Includes prefetches (dcfetch and hwprefetch).
0xad	DU_WRITE_TO_L2	DU write to L2. Total of everything that is written out of DU to the L2 array. Includes cleaninv. Excludes tag writes.
0xaf	DCZERO_COMMITTED	Committed a dczeroa instruction.
0xb3	DTLB_MISS	DTLB miss which goes to JTLB. When both slots miss to different pages, increments by 2. When both slots miss to the same page, only counts S1, since S1 goes first and fills for S0.
0xb5	DU_STORE_BUFFER_ACCESS	Number of times an entry went to the store buffer.
0xb6	STORE_BUFFER_HIT_REPLAY	Store buffer hit replay due to a packet with 2 stores going to the same bank but different cachelines, followed by a load from an address that was pushed into the store buffer.
0xb7	STORE_BUFFER_FORCE_REPLAY	The store buffer must drain, forcing the current packet to replay. This usually happens on an cache index match between the current packet and store buffer. Could also a store buffer timeout.
0xb9	SMT_BANK_CONFLICT	Counts inter-thread SMT bank conflicts.
0xba	PORT_CONFLICT_REPLAY	Counts all port conflict replays, including the same cluster replays that were caused by high-priority fills and store buffer force drains (and intercluster ones as well).
0xbd	PAGE_CROSS_REPLAY	Page cross replay. Page cross from valid packet that caused replay. Excludes pdkill packets. Counts twice if both slots cause a page cross.
0xbe	DU_DEALLOC_SECURITY_REPLAY	Replay caused by executing the instructions deallocframe or dealloc_return when FRAMEKEY is not equal to zero.
0xbf	DU_DEMAND_SECONDARY_MISS	DU demand secondary miss.
0xc3	DCFETCH_COMMITTED	Dcfetch committed. Includes hit and dropped. Does not include convert-to-prefetches.
0xc4	DCFETCH_HIT	Dcfetch hit in dcache. Includes hit valid or reserved line.
0xc5	DCFETCH_MISS	Dcfetch missed in L1. Counts the dcfetch issued to L2 FIFO.
0xc8	DU_LOAD_UNCACHEABLE	Uncacheable load in L1 cache. Counts twice for dual uncacheable loads.
0xc9	DU_DUAL_LOAD_UNCACHEABLE	Dual uncacheable loads in L1 cache.
0xca	DU_STORE_UNCACHEABLE	Uncacheable store in L1 cache. Counts twice for dual uncacheable stores.
0xcc	MISS_TO_PREFETCH	Dcache miss is converted to a prefetch. Could be the second load to miss in a packet, or a DMT subsequent packet on same thread. The normal dcache access will proceed when the first miss resolves. Includes converted prefetches into uncacheable space which will be cancelled later in the pipe.
0xce	AXI_LINE64_READ_REQUEST	64-byte line read requests issued by primary AXI master.
0xcf	AXI_LINE64_WRITE_REQUEST	64-byte line write requests issued by primary AXI master. All bytes valid.
0xd0	AXI_WR_CONGESTION	Primary AXI write command or data queue is full and an operation is stuck at the head of the primary AXI master command queue.

Table 9-2 V61 processor event symbols

Event	Symbol	Definition
0xd1	AHB_8_READ_REQUEST	An 8-byte AHB read was made.
0xd2	AXI_INCOMPLETE_WRITE_ REQUEST	A burst write was made by the primary AXI master, but not all of the bytes were valid.
0xd3	L2FETCH_COMMAND_PAGE_ TERMINATION	L2fetch command terminated because it could not get a page translation from VA to PA. Includes termination dues to permission errors. i.e. address translation could fail because the VA to PA is not in the TLB, or the properties in the translation are not acceptable and the command terminates.
0xd4	REQUEST_STALL_WRITE_ BUFFER_EXHAUSTION	Request to L2 stalled due to the lack of a write buffer. This is set for scalar and vector stores.
0xd5	L2_DU_STORE_COALESCE	Events from 0x8b that were coalesced.
0xd6	L2_STORE_LINK	Number of times a new store links to something else in the scoreboard.
0xd7	L2_SCOREBOARD_70_PERCENT_ FULL	Increments by 1 every cycle that the L2 scoreboard is at least 70% full. For a 32-entry scoreboard, this indicates that 23 or more entries are consumed. This event continues to count even if the scoreboard is more than 80% full. For more than one interleave, this event considers only the scoreboard with the most entries consumed.
0xd8	L2_SCOREBOARD_80_PERCENT_ FULL	Increments by 1 every cycle that the L2 scoreboard is at least 80% full. For a 32-entry scoreboard, this indicates that 26 or more entries are consumed. This event continues to count even if the scoreboard is more than 90% full. For more than one interleave, this event considers only the scoreboard with the most entries consumed.
0xd9	L2_SCOREBOARD_90_PERCENT_ FULL	Increments by 1 every cycle that the L2 scoreboard is at least 90% full. For a 32-entry scoreboard, this indicates that 29 or more entries are consumed. For more than one interleave, this event considers only the scoreboard with the most entries consumed.
0xda	L2_SCOREBOARD_FULL_REJECT	L2 scoreboard is too full to accept a selector request, and the selector has a request.
0xdc	REQUEST_STALL_EVICTION_ BUFFER_EXHAUSTION	Request to L2 stalled due to the lack of an eviction buffer. This is set for scalar and vector loads.
0xdd	AHB_MULTI_BEAT_READ_ REQUEST	A 32-byte multi-beat AHB read was made.
0xdf	L2_DU_LOAD_SECONDARY_MISS_ ON_SW_PREFETCH	Of the events in 0x90, the ones where the primary miss was a DCFETCH or L2FETCH.
0xe6	ARCH_LOCK_PVIEW_CYCLES	Cycles cluster could not dispatch due to kernel lock or TLB lock.
0xe7	IU_BRANCH_MISS_PVIEW_CYCLES	Packets in-flight killed due to branch mispredict, and cycles cluster could not dispatch due to IQ being empty because of branch mispredict.
0xe8	IU_NO_PKT_PVIEW_CYCLES	Cycles cluster could not dispatch due to IQ being empty.
0xe9	DU_CACHE_MISS_PVIEW_CYCLES	Cycles cluster could not dispatch due to D-cache miss or uncached access.
0xea	DU_BUSY_OTHER_PVIEW_CYCLES	Packets in flight killed due to DU replay and cycles cluster could not dispatch due to DU busy because of stall or replay.

Table 9-2 V61 processor event symbols

Event	Symbol	Definition
0xeb	CU_BUSY_PVIEW_CYCLES	Cycles cluster could not dispatch due to register interlock, register port conflict, bubbles due to timing class such as tc_3stall, no B2B HVX, or HVX FIFO full.
0xec	SMT_DU_CONFLICT_PVIEW_ CYCLES	Cycles cluster could not dispatch even though the thread was ready due to DU resource conflict.
0xed	SMT_XU_CONFLICT_PVIEW_ CYCLES	Cycles cluster could not dispatch even though the cluster was ready due to an XU resource conflict or other non-DU reason.
oxf0	COPROC_PKT_THREAD	Committed packets on a thread with XE bit set, whether executed in core or coprocessor.
oxf1	COPROC_PKT_COUNT	Packets with HVX instructions.
oxf2	COPROC_POWER_THROTTLING_ STALL_CYCLES	Stall cycles due to HVX engine being throttled for power reasons.
oxf3	COPROC_REGISTER_STALL_ CYCLES	Stall cycles due to interlocks.
oxf4	COPROC_LOAD_STALL_CYCLES	Stall cycles due to load pending.
oxf5	COPROC_STORE_STALL_CYCLES	Stall cycles due to store not yet allocated in L2.
oxf6	COPROC_BUSY_STALL_CYCLES	Stall cycles due to multi-issue packet.
oxf7	COPROC_FREEZE_STALL_CYCLES	Stall cycles due to L2 vector store data FIFO full.
oxf8	COPROC_CORE_VFIFO_FULL_ STALL	Number of times a thread had to stall due to VFIFO being full.
oxf9	COPROC_L2_STORE_ACCESS	L2 cacheable store access from HVX. Any store access from HVX that may cause a lookup in L2 cache. Excludes cache ops, uncacheables, scalar stores.
oxfa	COPROC_L2_STORE_MISS	L2 cacheable miss from HVX. Of the events qualified by 0xf9, the ones that resulted in a miss. Specifically the cases where line is not in cache or a coalesce buffer. DCZero does not count as a miss, if it allocates without a backing store fetch.
oxfb	COPROC_L2_LOAD_ACCESS	L2 cacheable load access from HVX. Any load access from HVX that may cause a lookup in the L2 cache. Excludes cache ops, uncacheables, scalars.
oxfc	COPROC_L2_LOAD_MISS	L2 cacheable miss from HVX. Of the events qualified by 0xFB, the ones that resulted in a miss i.e. the line was not previously allocated in the L2 cache and will be fetched from backing memory.
oxfd	COPROC_TCM_STORE_ACCESS	TCM store access for HVX. HVX store to L2 TCM space.
oxfe	COPROC_TCM_LOAD_ACCESS	TCM load access for HVX. HVX load from L2 TCM space
oxff	COPROC_L2_LOAD_SECONDARY_ MISS	Of the events in 0xfb, the ones where the load could not be returned due to the immediately prior access for the line being a pending load or pending L2Fetch.

10 Instruction Encoding

10.1 Overview

This chapter describes the binary encoding of Hexagon processor instructions and instruction packets. It covers the following topics:

- Instructions
- Sub-instructions
- Duplex instructions
- Instruction classes
- Instruction packets
- Loop packets
- Immediate operands
- Scaled immediate operands
- Constant extenders
- New-value operands
- Instruction mapping

10.2 Instructions

All Hexagon processor instructions are encoded in a 32-bit instruction word. The instruction word format varies according to the instruction type.

The instruction words contain two types of bit fields:

- *Common fields* appear in every processor instruction, and are defined the same in all instructions.
- *Instruction-specific fields* appear only in some instructions, or vary in definition across the instruction set.

Table 10-1 lists the instruction bit fields.

Table 10-1 Instruction fields

Name	Description	Type
ICLASS	Instruction class	Common
Parse	Packet / loop bits	
MajOp Maj	Major opcode	Instruction-specific
MinOp Min	Minor opcode	
RegType	Register type (32-bit, 64-bit)	
Type	Operand type (byte, halfword, etc.)	
Amode	Addressing mode	
d*n*	Destination register operand	
s*n*	Source register operand	
t*n*	Source register operand #2	
x*n*	Source and destination register operand	
u*n*	Predicate or modifier register operand	
sH	Source register bit field (Rs.H or Rs.L)	
tH	Source register #2 bit field (Rt.H or Rt.L)	
UN	Unsigned operand	
Rs	No source register read	
P	Predicate expression	
PS	Predicate sense (Pu or !Pu)	
DN	Dot-new predicate	
PT	Predict taken	
sm	Supervisor mode only	

NOTE In some cases instruction-specific fields are used to encode instruction attributes other than the ones described for the fields in Table 10-1.

Reserved bits

Some instructions contain *reserved bits* which are not currently used to encode instruction attributes. These bits should always be set to 0 to ensure compatibility with any future changes in the instruction encoding.

NOTE Reserved bits appear as '-' characters in the instruction encoding tables.

10.3 Sub-instructions

To reduce code size the Hexagon processor supports the encoding of certain pairs of instructions in a single 32-bit container. Instructions encoded this way are called *sub-instructions*, and the containers are called *duplexes* (Section 10.4).

Sub-instructions are limited to certain commonly-used instructions:

- Arithmetic and logical operations
- Register transfer
- Loads and stores
- Stack frame allocation/deallocation
- Subroutine return

Table 10-2 lists the sub-instructions along with the group identifiers that are used to encode them in duplexes.

Sub-instructions can access only a subset of the general registers (R0-R7, R16-R23). Table 10-3 lists the sub-instruction register encodings.

NOTE Certain sub-instructions implicitly access registers such as SP (R29).

Table 10-2 Sub-instructions

Group	Instruction	Description
L1	`Rd = memw(Rs+#u4:2)`	Word load
L1	`Rd = memub(Rs+#u4:0)`	Unsigned byte load
Group	**Instruction**	**Instruction**
L2	`Rd = memh/memuh(Rs+#u3:1)`	Halfword loads
L2	`Rd = memb(Rs+#u3:0)`	Signed byte load
L2	`Rd = memw(r29+#u5:2)`	Load word from stack
L2	`Rdd = memd(r29+#u5:3)`	Load pair from stack
L2	`deallocframe`	Dealloc stack frame
L2	`if ([!]P0) dealloc_return` `if ([!]P0.new) dealloc_return:nt`	Dealloc stack frame and return

L2	`jumpr R31`
	`if ([!]P0) jumpr R31`
	`if ([!]P0.new) jumpr:nt R31`
		Return

Table 10-2 Sub-instructions (Continued)

Group	Instruction	Description
Group	**Instruction**	**Instruction**
S1	`memw(Rs+#u4:2) = Rt`	Store word
S1	`memb(Rs+#u4:0) = Rt`	Store byte
Group	**Instruction**	**Instruction**
S2	`memh(Rs+#u3:1) = Rt`	Store halfword
S2	`memw(r29+#u5:2) = Rt`	Store word to stack
S2	`memd(r29+#s6:3) = Rtt`	Store pair to stack
S2	`memw(Rs+#u4:2) = #U1`	Store immediate word #0 or #1
S2	`memb(Rs+#u4) = #U1`	Store immediate byte #0 or #1
S2	`allocframe(#u5:3)`	Allocate stack frame
Group	**Instruction**	**Instruction**
A	`Rx = add(Rx,#s7)`	Add immediate
A	`Rd = Rs`	Transfer
A	`Rd = #u6`	Set to unsigned immediate
A	`Rd = #-1`	Set to -1
A	`if ([!]P0[.new]) Rd = #0`	Conditional clear
A	`Rd = add(r29,#u6:2)`	Add immediate to stack pointer
A	`Rx = add(Rx,Rs)`	Register add
A	`P0 = cmp.eq(Rs,#u2)`	Compare register equal immed
A	`Rdd = combine(#0,Rs)`	Combine zero and register into pair
A	`Rdd = combine(Rs,#0)`	Combine register and zero into pair
A	`Rdd = combine(#u2,#U2)`	Combine immediates into pair
A	`Rd = add(Rs,#1)` `Rd = add(Rs,#-1)`	Add and Subtract 1
A	`Rd = sxth/sxtb/zxtb/zxth(Rs)`	Sign- and zero-extends
A	`Rd = and(Rs,#1)`	And with 1

Table 10-3 Sub-instruction registers

Register	Encoding
Rs,Rt,Rd,Rx	0000 = R0 0001 = R1 0010 = R2 0011 = R3 0100 = R4 0101 = R5 0110 = R6 0111 = R7 1000 = R16 1001 = R17 1010 = R18 1011 = R19 1100 = R20 1101 = R21 1110 = R22 1111 = R23
Rdd,Rtt	000 = R1:0 001 = R3:2 010 = R5:4 011 = R7:6 100 = R17:16 101 = R19:18 110 = R21:20 111 = R23:22

10.4 Duplexes

A *duplex* is encoded as a 32-bit instruction with bits [15:14] set to 00. The sub-instructions (Section 10.3) that comprise a duplex are encoded as 13-bit fields in the duplex.

Table 10-4 shows the encoding details for a duplex.

An instruction packet can contain one duplex and up to two other (non-duplex) instructions. The duplex must always appear as the last word in a packet.

The sub-instructions in a duplex are always executed in Slot 0 and Slot 1.

Table 10-4 Duplex instruction

Bits	Name	Description
15:14	Parse Bits	00 = Duplex type, ends the packet and indicates that word contains two sub-instructions
12:0	Sub-insn low	Encodes Slot 0 sub-instruction
28:16	Sub-insn high	Encodes Slot 1 sub-instruction
31:29, 13	4-bit ICLASS	Indicates which group the low and high sub-instructions below to.

Table 10-5 lists the duplex ICLASS field values, which specify the group of each sub-instruction in a duplex.

Table 10-5 Duplex ICLASS field

ICLASS	Low Slot 0 subinsn type	High Slot 1 subinsn type
0x0	L1-type	L1-type
0x1	L2-type	L1-type
0x2	L2-type	L2-type
0x3	A-type	A-type
0x4	L1-type	A-type
0x5	L2-type	A-type
0x6	S1-type	A-type
0x7	S2-type	A-type
0x8	S1-type	L1-type
0x9	S1-type	L2-type
0xA	S1-type	S1-type
0xB	S2-type	S1-type
0xC	S2-type	L1-type
0xD	S2-type	L2-type
0xE	S2-type	S2-type
0xF	Reserved	Reserved

Duplexes have the following grouping constraints:

- Constant extenders enable the range of an instruction's immediate operand to be expanded to 32 bits (Section 10.10). The following sub-instructions can be expanded with a constant extender:
  - `Rx = add(Rx,#s7)`
  - `Rd = #u6`

  Note that a duplex can contain only one constant-extended instruction, and it must appear in the Slot 1 position.
- If a duplex contains two instructions with the same sub-instruction group, the instructions must be ordered in the duplex as follows: when the sub-instructions are treated as 13-bit unsigned integer values, the instruction corresponding to the numerically smaller value must be encoded in the Slot 1 position of the duplex.[1]
- Sub-instructions must conform to any slot assignment grouping rules that apply to the individual instructions, even if a duplex pattern exists which violates those assignments. One exception to this rule exists:
  - `jumpr` R31 must appear in the Slot 0 position

[1] Note that the sub-instruction register and immediate fields are assumed to be 0 when performing this comparison.

10.5 Instruction classes

The instruction class (Section 3.3) is encoded in the four most-significant bits of the instruction word (31:28). These bits are referred to as the instruction's ICLASS field.

Table 10-6 lists the encoding values for the instruction classes. The Slots column indicates which slots can receive the instruction class.

Table 10-6 Instruction class encoding

Encoding	Instruction Class	Slots
0000	Constant extender (Section 10.10)	–
0001	J	2,3
0010	J	2,3
0011	LD ST	0,1
0100	LD ST (conditional or GP-relative)	0,1
0101	J	2,3
0110	CR	3
0111	ALU32	0,1,2,3
1000	XTYPE	2,3
1001	LD	0,1
1010	ST	0
1011	ALU32	0,1,2,3
1100	XTYPE	2,3
1101	XTYPE	2,3
1110	XTYPE	2,3
1111	ALU32	0,1,2,3

For details on encoding the individual class types see Chapter 11.

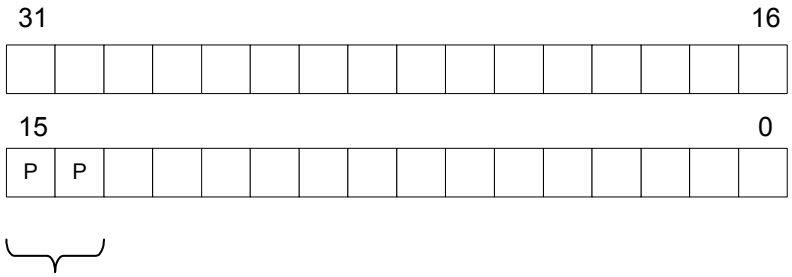
10.6 Instruction packets

Instruction packets are encoded using two bits of the instruction word (15:14), which are referred to as the instruction word's Parse field. The field values have the following definitions:

- '11' indicates that an instruction is the last instruction in a packet (i.e., the instruction word at the highest address).
- '01' or '10' indicate that an instruction is not the last instruction in a packet.
- '00'indicates a duplex (Section 10.4)

If any sequence of four consecutive instructions occurs without one of them containing '11' or '00', the processor will raise an error exception (illegal opcode).

Figure 10-1 shows the location of the Parse field in an instruction word.



Packet / Loop Parse Bits:
01, 10 = not end of packet
11 = end of packet
00 = duplex

Figure 10-1 Instruction packet encoding

The following examples show how the Parse field is used to encode instruction packets:

```
{ A ; B}
  01 11              // Parse fields of instrs A,B

{ A ; B ; C}
  01 01  11          // Parse fields of instrs A,B,C

{ A ; B ; C ; D}
  01 01  01  11      // Parse fields of instrs A,B,C,D
```

10.7 Loop packets

In addition to encoding the last instruction in a packet, the instruction word's Parse field (Section 10.6) is used to encode the last packet in a hardware loop.

The Hexagon processor supports two hardware loops, labelled 0 and 1 (Section 7.3). The last packet in these loops is subject to the following restrictions:

- The last packet in a hardware loop 0 must contain two or more instruction words.
- The last packet in a hardware loop 1 must contain three or more instruction words.

If the last packet in a loop is expressed in assembly language with fewer than the required number of words, the assembler automatically adds one or two NOP instructions to the encoded packet so it contains the minimum required number of instruction words.

The Parse fields in a packet's first and second instruction words (i.e., the words at the lowest addresses) encode whether or not the packet is the last packet in a hardware loop.

Table 10-7 shows how the Parse fields are used to encode loop packets.

Table 10-7 Loop packet encoding

Packet	Parse Field in First Instruction	Parse Field in Second Instruction
Not last in loop	01 or 11	01 or 11 [1]
Last in loop 0	10	01 or 11
Last in loop 1	01	10
Last in loops 0 & 1	10	10

[1] Not applicable for single-instruction packets.

The following examples show how the Parse field is used to encode loop packets:

```
{ A   B}:endloop0
  10  11                          // Parse fields of instrs A,B

{ A   B   C}:endloop0
  10  01  11                      // Parse fields of instrs A,B,C

{ A   B   C   D}:endloop0
  10  01  01  11                  // Parse fields of instrs A,B,C,D

{ A   B   C}:endloop1
  01  10  11                      // Parse fields of instrs A,B,C

{ A   B   C   D}:endloop1
  01  10  01  11                  // Parse fields of instrs A,B,C,D

{ A   B   C}:endloop0:endloop1
  10  10  11                      // Parse fields of instrs A,B,C

{ A   B   C   D}:endloop0:endloop1
  10  10  01  11                  // Parse fields of instrs A,B,C,D
```

10.8 Immediate values

To conserve encoding space, the Hexagon processor often stores immediate values in instruction fields that are smaller (in bit size) than the values actually needed in the instruction operation.

When an instruction operates on one of its immediate operands, the processor automatically extends the immediate value to the bit size required by the operation:

- Signed immediate values are *sign-extended*
- Unsigned immediate values are *zero-extended*

10.9 Scaled immediates

To minimize the number of bits used in instruction words to store certain immediate values, the Hexagon processor stores the values as *scaled immediates*. Scaled immediate values are used when an immediate value needs to represent integral multiples of a power of 2 in a specific range.

For example, consider an instruction operand whose possible values are the following:

-32, -28, -24, -20, -16, -12, -8, -4, 0, 4, 8, 12, 16, 20, 24, 28

Encoding the full range of integers -32..28 would normally require 6 bits. However, if the operand is stored as a scaled immediate, it can first be shifted right by two bits, with only the four remaining bits being stored in the instruction word. When the operand is fetched from the instruction word, the processor automatically shifts the value left by two bits to recreate the original operand value.

NOTE The scaled immediate value in the example above is represented notationally as `#s4:2`. For more information see Section 1.7.

Scaled immediate values are commonly used to encode address offsets which apply to data types of varying size. For example, Table 10-8 shows how the byte offsets used in immediate-with-offset addressing mode are stored as 11-bit scaled immediate values. This enables the offsets to span the same range of data elements regardless of the data type.

Table 10-8 Scaled immediate encoding (indirect offsets)

Data Type	Offset Size (Stored)	Scale Bits	Offset Size (Effective)	Offset Range (Bytes)	Offset Range (Elements)
byte	11	0	11	-1024 ... 1023	-1024 ... 1023
halfword	11	1	12	-2048 ... 2046	-1024 ... 1023
word	11	2	13	-4096 ... 4092	-1024 ... 1023
doubleword	11	3	14	-8192 ... 8184	-1024 ... 1023

10.10 Constant extenders

To support the use of 32-bit operands in a number of instructions, the Hexagon processor defines an instruction word which exists solely to extend the bit range of an immediate or address operand that is contained in an adjacent instruction in a packet. These instruction words are called *constant extenders*.

For example, the absolute addressing mode specifies a 32-bit constant value as the effective address. Instructions using this addressing mode are encoded in a single packet containing both the normal instruction word and a second word with a constant extender that increases the range of the instruction's normal constant operand to a full 32 bits.

NOTE Constant extended operands can encode symbols.

A constant extender is encoded as a 32-bit instruction with the 4-bit ICLASS field set to 0 and the 2-bit Parse field set to its usual value (Section 10.6). The remaining 26 bits in the instruction word store the data bits that are prepended to an operand as small as 6 bits in order to create a full 32-bit value.

Table 10-9 shows the encoding details.

Table 10-9 Constant extender encoding

Bits	Name	Description
31:28	ICLASS	Instruction class = 0000
27:16	Extender high	High 12 bits of 26-bit constant extension
15:14	Parse	Parse bits
13:0	Extender low	Low 14 bits of 26-bit constant extension

Within a packet, a constant extender must be positioned immediately before the instruction that it extends: in terms of memory addresses, the extender word must reside at address (<instr_address> - 4).

The constant extender effectively serves as a prefix for an instruction: it is not executed in a slot, nor does it consume any slot resources. All packets must contain four or fewer words, and the constant extender occupies one word.

If the instruction operand to be extended is longer than 6 bits, the overlapping bits in the base instruction must be encoded as zeros. The value in the constant extender always supplies the upper 26 bits.

Table 10-10 lists the instructions that work with constant extenders.

The `Regclass` field in the table lists the values that bits [27:24] must be set to in the instruction word to identify the instruction as one that may include a constant extender.

NOTE In cases where the base instruction encodes two constant operands, the extended immediate is the one specified in the table.

Constant extenders appear in disassembly listings as Hexagon instructions with the name `immext`.

Table 10-10 Constant extender instructions

ICLASS	Regclass	Instructions

LD	---1	`Rd = mem{b,ub,h,uh,w,d}(##U32)`
		`if ([!]Pt[.new]) Rd = mem{b,ub,h,uh,w,d} (Rs + ##U32)`
		`// predicated loads`

LD	----	`Rd = mem{b,ub,h,uh,w,d} (Rs + ##U32)`
		`Rd = mem{b,ub,h,uh,w,d} (Re=##U32)`
		`Rd = mem{b,ub,h,uh,w,d} (Rt<<#u2 + ##U32)`
		`if ([!]Pt[.new]) Rd = mem{b,ub,h,uh,w,d} (##U32)`

ST	---0	`mem{b,h,w,d}(##U32) = Rs[.new]  // GP-stores`
		`if ([!]Pt[.new]) mem{b,h,w,d}(Rs + ##U32) = Rt[.new]`
		`// predicated stores`

ST	----	`mem{b,h,w,d}(Rs + ##U32) = Rt[.new]`
		`mem{b,h,w,d}(Rd=##U32) = Rt[.new]`
		`mem{b,h,w,d}(Ru<<#u2 + ##U32) = Rt[.new]`
		`if ([!]Pt[.new]) mem{b,h,w,d}(##U32) = Rt[.new]`

MEMOP	----	`[if [!]Ps] memw(Rs + #u6) = ##U32  // constant store` `memw(Rs + Rt<<#u2) = ##U32  // constant store`
NV	----	`if (cmp.xx(Rs.new,##U32)) jump:hint target`

ALU32	----	`Rd = ##u32`
		`Rdd = combine(Rs,##u32)`
		`Rdd = combine(##u32,Rs)`
		`Rdd = combine(##u32,#s8)`
		`Rdd = combine(#s8,##u32)`
		`Rd = mux (Pu, Rs,##u32)`
		`Rd = mux (Pu, ##u32, Rs)`
		`Rd = mux(Pu,##u32,#s8)`
		`if ([!]Pu[.new]) Rd = add(Rs,##u32)`
		`if ([!]Pu[.new]) Rd = ##u32`
		`Pd = [!]cmp.eq (Rs,##u32)`
		`Pd = [!]cmp.gt (Rs,##u32)`
		`Pd = [!]cmp.gtu (Rs,##u32)`
		`Rd = [!]cmp.eq(Rs,##u32)`
		`Rd = and(Rs,##u32)`
		`Rd = or(Rs,##u32)`
		`Rd = sub(##u32,Rs)`

ALU32	----	`Rd = add(Rs,##s32)`

XTYPE	00--	`Rd = mpyi(Rs,##u32)`
		`Rd += mpyi(Rs,##u32)`
		`Rd -= mpyi(Rs,##u32)`
		`Rx += add(Rs,##u32)`
		`Rx -= add(Rs,##u32)`

ALU32	----[1]	`Rd = ##u32` `Rd = add(Rs,##s32)`

J	1---	`jump (PC + ##s32)`
		`call (PC + ##s32)`
		`if ([!]Pu) call (PC + ##s32)`

Table 10-10 Constant extender instructions (Continued)

ICLASS	Regclass	Instructions
CR	----	Pd = spNloop0(PC+##s32,Rs/#U10) loop0/1 (PC+##s32,#Rs/#U10)
XTYPE	1---	Rd = add(pc,##s32) Rd = add(##u32,mpyi(Rs,#u6)) Rd = add(##u32,mpyi(Rs,Rt)) Rd = add(Rs,add(Rt,##u32)) Rd = add(Rs,sub(##u32,Rt)) Rd = sub(##u32,add(Rs,Rt)) Rd = or(Rs,and(Rt,##u32)) Rx = add/sub/and/or (##u32,asl/asr/lsr(Rx,#U5)) Rx = add/sub/and/or (##u32,asl/asr/lsr(Rs,Rx)) Rx = add/sub/and/or (##u32,asl/asr/lsr(Rx,Rs)) Pd = cmpb/h.{eq,gt,gtu} (Rs,##u32)

[1] Constant extension is only for a Slot 1 sub-instruction.

NOTE If a constant extender is encoded in a packet for an instruction that does not accept a constant extender, the execution result is undefined. The assembler normally ensures that only valid constant extenders are generated.

Encoding 32-bit address operands in load/stores

Two methods exist for encoding a 32-bit absolute address in a load or store instruction:

1) For unconditional load/stores, the GP-relative load/store instruction is used. The assembler encodes the absolute 32-bit address as follows:

- The upper 26 bits are encoded in a constant extender
- The lower 6 bits are encoded in the 6 operand bits contained in the GP-relative instruction

In this case the 32-bit value encoded must be a plain address, and the value stored in the GP register is ignored.

NOTE When a constant extender is explicitly specified with a GP-relative load/store, the processor ignores the value in GP and creates the effective address directly from the 32-bit constant value.

2) For conditional load/store instructions that have their base address encoded only by a 6-bit immediate operand, a constant extender *must* be explicitly specified; otherwise, the execution result is undefined. The assembler ensures that these instructions always include a constant extender.

This case applies also to instructions that use the absolute-set addressing mode or absolute-plus-register-offset addressing mode.

Encoding 32-bit immediate operands

The immediate operands of certain instructions use scaled immediates (Section 10.9) to increase their addressable range. When constant extenders are used, scaled immediates are *not* scaled by the processor. Instead, the assembler must encode the full 32-bit unscaled value as follows:

- The upper 26 bits are encoded in the constant extender
- The lower 6 bits are encoded in the base instruction in the least-significant bit positions of the immediate operand field.
- Any overlapping bits in the base instruction are encoded as zeros.

Encoding 32-bit jump/call target addresses

When a jump/call has a constant extender, the resulting target address is forced to a 32-bit alignment (i.e., bits 1:0 in the address are cleared by hardware). The resulting jump/call operation will never cause an alignment violation.

10.11 New-value operands

Instructions that include a new-value register operand specify in their encodings which instruction in the packet has its destination register accessed as the new-value register.

New-value consumers include a 3-bit instruction field named `Nt` which specifies this information.

- Nt[0] is reserved and should always be encoded as zero. A non-zero value produces undefined results.
- Nt[2:1] encodes the distance (in instructions) from the producer to the consumer, as follows:
  - `Nt[2:1] = 00` // reserved
  - `Nt[2:1] = 01` // producer is +1 instruction ahead of consumer
  - `Nt[2:1] = 10` // producer is +2 instructions ahead of consumer
  - `Nt[2:1] = 11` // producer is +3 instructions ahead of consumer

"ahead" is defined here as the instruction encoded at a lower memory address than the consumer instruction, not counting empty slots or constant extenders. For example, the following producer/consumer relationship would be encoded with `Nt[2:1]` set to 01.

```
...
<producer instruction word>
<consumer constant extender word>
<consumer instruction word>
...
```

NOTE Instructions with 64-bit register pair destinations cannot produce new-values. The assembler flags this case with an error, as the result is undefined.

10.12 Instruction mapping

Some Hexagon processor instructions are encoded by the assembler as variants of other instructions. This is done for operations that are functionally equivalent to other instructions, but are still defined as separate instructions because of their programming utility as common operations.

Table 10-11 lists some of the instructions that are mapped to other instructions.

Table 10-11 Instruction mapping

Instruction	Mapping
`Rd = not(Rs)`	`Rd = sub(#-1,Rs)`
`Rd = neg(Rs)`	`Rd = sub(#0,Rs)`
`Rdd = Rss`	`Rdd = combine(Rss.H32, Rss.L32)`

11 Instruction Set

11.1 Overview

This chapter describes the instruction set for version 6 of the Hexagon processor.

The instructions are listed alphabetically within instruction categories. The following information is provided for each instruction:

- Instruction name
- A brief description of the instruction
- A high-level functional description (syntax and behavior) with all possible operand types
- Instruction class and slot information for grouping instructions in packets
- Notes on miscellaneous issues
- Any C intrinsic functions that provide access to the instruction
- Instruction encoding

11.2 Instruction categories

- ALU32 — 32-bit ALU operations
  - ALU — Arithmetic and Logical
  - PERM — Permute
  - PRED — Predicate
- CR — Control registers, looping
- JR — Jump from Register
- J — Jump
- LD — Load
- MEMOP — Memory operations
- NV — New-value operations
  - J — New-value jumps
  - ST — New-value stores
- ST — Store operations
- SYSTEM
  - User Instructions
- XTYPE — 32-bit and 64-bit operations
  - ALU — Arithmetic and Logical
  - BIT — Bit
  - COMPLEX — Complex
  - FP — Floating point
  - MPY — Multiply
  - PERM — Permute
  - PRED — Predicate
  - SHIFT — Shift

Table 11-1 lists the symbols used to specify the instruction syntax.

Table 11-1 Instruction syntax symbols

Symbol	Example	Meaning
=	R2 = R3;	Assignment of RHS to LHS
;	R2 = R3;	Marks the end of an instruction or group of instructions
{ … }	{R2 = R3; R5 = R6;}	Indicates a group of parallel instructions.
#	#100	Immediate constant value
##	##2147483647	32-bit immediate constant value
0x	R2 = #0x1fe;	Indicates hexadecimal number
MEMxx	R2 = MEMxx(R3)	Access memory. xx specifies the size and type of access.
:sat	R2 = add(r1,r2):sat	Perform optional saturation
:rnd	R2 = mpy(r1.h,r2.h):rnd	Perform optional rounding
:<<16	R2 = add(r1.l,r2.l):<<16	Shift left by a halfword

Table 11-2 lists the symbols used to specify instruction operands.

Table 11-2 Instruction operand symbols

Symbol	Example	Meaning
#uN	R2 = #u16	Unsigned N-bit immediate value
#sN	R2 = add(R3,#s16)	Signed N-bit immediate value
#mN	Rd = mpyi(Rs,#m9)	Signed N-bit immediate value
#uN:S	R2 = memh(#u16:1)	Unsigned N-bit immediate value representing integral multiples of 2S in specified range
#sN:S	Rd = memw(Rs++#s4:2)	Signed N-bit immediate value representing integral multiples of 2S in specified range
#rN:S	call #r22:2	Same as #sN:S, but value is offset from PC of current packet
##	call ##32	Same as #, but associated value (u,s,m,r) is 32 bits

When an instruction contains more than one immediate operand, the operand symbols are specified in upper and lower case (e.g., #uN and #UN) to indicate where they appear in the instruction encodings.

The instruction behavior is specified using a superset of the C language. Table 11-3 lists symbols not defined in C which are used to specify the instruction behavior.

Table 11-3 Instruction behavior symbols

Symbol	Example	Meaning
usat_N	usat_16(Rs)	Saturate a value to an unsigned N-bit
sat_N	sat_16(Rs)	Saturate a value to a signed N-bit number
sxt x->y	sxt32->64(Rs)	Sign-extend value from x to y bits
zxt x->y	zxt32->64(Rs)	Zero-extend value from x to y bits
>>>	Rss >>> offset	Logical right shift

11.3 ALU32

The ALU32 instruction class includes instructions which perform arithmetic and logical operations on 32-bit data.

ALU32 instructions are executable on any slot.

11.3.1 ALU32/ALU

The ALU32/ALU instruction subclass includes instructions which perform arithmetic and logical operations on individual 32-bit items.

Add

Add a source register either to another source register or to a signed 16-bit immediate value. Store result in destination register. Source and destination registers are 32 bits. If the result overflows 32 bits, it wraps around. Optionally saturate result to a signed value between 0x80000000 and 0x7fffffff.

For 64-bit versions of this operation, see the XTYPE add instructions.

Syntax	Behavior
`Rd=add(Rs,#s16)`	`apply_extension(#s);` `Rd=Rs+#s;`
`Rd=add(Rs,Rt)`	`Rd=Rs+Rt;`
`Rd=add(Rs,Rt):sat`	`Rd=sat_32(Rs+Rt);`

Class: ALU32 (slots 0,1,2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=add(Rs,#s16)`	`Word32 Q6_R_add_RI(Word32 Rs, Word32 Is16)`
`Rd=add(Rs,Rt)`	`Word32 Q6_R_add_RR(Word32 Rs, Word32 Rt)`
`Rd=add(Rs,Rt):sat`	`Word32 Q6_R_add_RR_sat(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse											d5					
1	0	1	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=add(Rs,#s16)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=add(Rs,Rt)
1	1	1	1	0	1	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=add(Rs,Rt):sat

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`P`	Predicated
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits

Field name	Description
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Logical operations

Perform bitwise logical operations (AND, OR, XOR, NOT) either on two source registers or on a source register and a signed 10-bit immediate value. Store result in destination register. Source and destination registers are 32 bits.

For 64-bit versions of these operations, see the XTYPE logical instructions.

Syntax	Behavior
`Rd=and(Rs,#s10)`	`apply_extension(#s);` `Rd=Rs&#s;`
`Rd=and(Rs,Rt)`	`Rd=Rs&Rt;`
`Rd=and(Rt,~Rs)`	`Rd = (Rt & ~Rs);`
`Rd=not(Rs)`	`Assembler mapped to: "Rd=sub(#-1,Rs)"`
`Rd=or(Rs,#s10)`	`apply_extension(#s);` `Rd=Rs|#s;`
`Rd=or(Rs,Rt)`	`Rd=Rs|Rt;`
`Rd=or(Rt,~Rs)`	`Rd = (Rt | ~Rs);`
`Rd=xor(Rs,Rt)`	`Rd=Rs^Rt;`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=and(Rs,#s10)`	`Word32 Q6_R_and_RI(Word32 Rs, Word32 Is10)`
`Rd=and(Rs,Rt)`	`Word32 Q6_R_and_RR(Word32 Rs, Word32 Rt)`
`Rd=and(Rt,~Rs)`	`Word32 Q6_R_and_RnR(Word32 Rt, Word32 Rs)`
`Rd=not(Rs)`	`Word32 Q6_R_not_R(Word32 Rs)`
`Rd=or(Rs,#s10)`	`Word32 Q6_R_or_RI(Word32 Rs, Word32 Is10)`
`Rd=or(Rs,Rt)`	`Word32 Q6_R_or_RR(Word32 Rs, Word32 Rt)`
`Rd=or(Rt,~Rs)`	`Word32 Q6_R_or_RnR(Word32 Rt, Word32 Rs)`
`Rd=xor(Rs,Rt)`	`Word32 Q6_R_xor_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse												d5					
0	1	1	1	0	1	1	0	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=and(Rs,#s10)
0	1	1	1	0	1	1	0	1	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=or(Rs,#s10)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	1	0	0	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=and(Rs,Rt)
1	1	1	1	0	0	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=or(Rs,Rt)
1	1	1	1	0	0	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=xor(Rs,Rt)
1	1	1	1	0	0	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=and(Rt,~Rs)
1	1	1	1	0	0	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=or(Rt,~Rs)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`P`	Predicated
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Negate

Perform arithmetic negation on a source register. Store result in destination register. Source and destination registers are 32 bits.

For 64-bit and saturating versions of this instruction, see the XTYPE-class negate instructions.

Syntax	Behavior
`Rd=neg(Rs)`	`Assembler mapped to: "Rd=sub(#0,Rs)"`

Class: N/A

Intrinsics

`Rd=neg(Rs)`	`Word32 Q6_R_neg_R(Word32 Rs)`

Nop

Perform no operation. This instruction is used for padding and alignment.

Within a packet it can be positioned in any slot 0-3.

Syntax	Behavior
`ext-nop`	
`nop`	

Class: ALU32 (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp											Parse																
0	1	1	1	1	1	1	1	-	-	-	-	-	-	-	-	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	nop

Field name	Description
`MajOp`	Major Opcode
`Rs`	No Rs read
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits

Subtract

Subtract a source register from either another source register or from a signed 10-bit immediate value. Store result in destination register. Source and destination registers are 32 bits. If the result underflows 32 bits, it wraps around. Optionally saturate result to a signed value between 0x8000_0000 and 0x7fff_ffff.

For 64-bit versions of this operation, see the XTYPE subtract instructions.

Syntax	Behavior
`Rd=sub(#s10,Rs)`	`apply_extension(#s);` `Rd=#s-Rs;`
`Rd=sub(Rt,Rs)`	`Rd=Rt-Rs;`
`Rd=sub(Rt,Rs):sat`	`Rd=sat_32(Rt - Rs);`

Class: ALU32 (slots 0,1,2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=sub(#s10,Rs)`	`Word32 Q6_R_sub_IR(Word32 Is10, Word32 Rs)`
`Rd=sub(Rt,Rs)`	`Word32 Q6_R_sub_RR(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt,Rs):sat`	`Word32 Q6_R_sub_RR_sat(Word32 Rt, Word32 Rs)`

Encoding

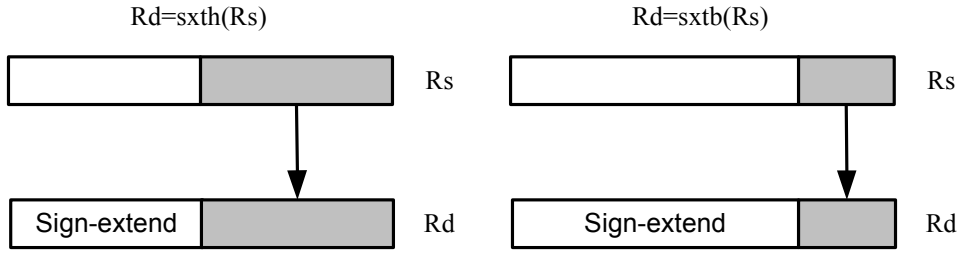
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse											d5					
0	1	1	1	0	1	1	0	0	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=sub(#s10,Rs)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt,Rs)
1	1	1	1	0	1	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt,Rs):sat

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`P`	Predicated

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Sign extend

Sign-extend the least-significant byte or halfword from the source register and place the 32-bit result in the destination register.



Syntax	Behavior
`Rd=sxtb(Rs)`	`Rd = sxt8->32(Rs);`
`Rd=sxth(Rs)`	`Rd = sxt16->32(Rs);`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=sxtb(Rs)`	`Word32 Q6_R_sxtb_R(Word32 Rs)`
`Rd=sxth(Rs)`	`Word32 Q6_R_sxth_R(Word32 Rs)`

Encoding

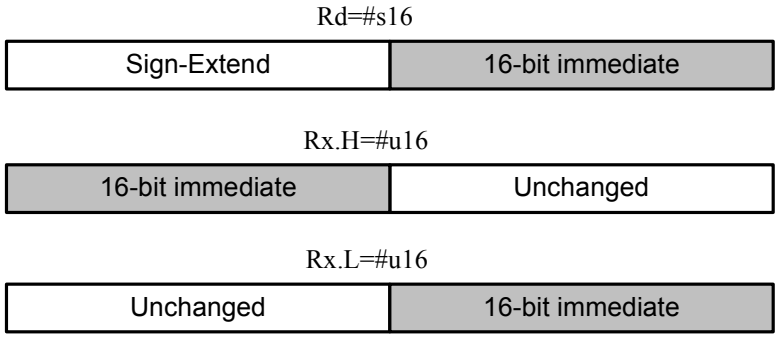
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse		C									d5					
0	1	1	1	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=sxtb(Rs)
0	1	1	1	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=sxth(Rs)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`C`	Conditional
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s

Transfer immediate

Assign an immediate value to a 32-bit destination register.

Two types of assignment are supported. The first sign-extends a 16-bit signed immediate value to 32 bits. The second assigns a 16-bit unsigned immediate value to either the upper or lower 16 bits of the destination register, leaving the other 16 bits unchanged.



Syntax	Behavior
`Rd=#s16`	`apply_extension(#s);` `Rd=#s;`

`Rdd=#s8`	`if ("#s8<0") {`
	`  Assembler mapped to: "Rdd=combine(#-1,#s8)";`
	`} else {`
	`  Assembler mapped to: "Rdd=combine(#0,#s8)";`
	`};`

`Rx.[HL]=#u16`	`Rx.h[01]=#u;`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=#s16`	`Word32 Q6_R_equals_I(Word32 Is16)`
`Rdd=#s8`	`Word64 Q6_P_equals_I(Word32 Is8)`
`Rx.H=#u16`	`Word32 Q6_Rh_equals_I(Word32 Rx, Word32 Iu16)`
`Rx.L=#u16`	`Word32 Q6_Rl_equals_I(Word32 Rx, Word32 Iu16)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			x5					Parse																
0	1	1	1	0	0	0	1	i	i	1	x	x	x	x	x	P	P	i	i	i	i	i	i	i	i	i	i	i	i	i	i	Rx.L=#u16
0	1	1	1	0	0	1	0	i	i	1	x	x	x	x	x	P	P	i	i	i	i	i	i	i	i	i	i	i	i	i	i	Rx.H=#u16
ICLASS				Rs	MajOp			MinOp								Parse											d5					
0	1	1	1	1	0	0	0	i	i	-	i	i	i	i	i	P	P	i	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=#s16

Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
Rs	No Rs read
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
x5	Field to encode register x

Transfer register

Transfer a source register to a destination register. Source and destination registers are either 32 bits or 64 bits.

Syntax	Behavior
`Rd=Rs`	`Rd=Rs;`
`Rdd=Rss`	`Assembler mapped to: "Rdd=combine(Rss.H32,Rss.L32)"`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=Rs`	`Word32 Q6_R_equals_R(Word32 Rs)`
`Rdd=Rss`	`Word64 Q6_P_equals_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse		C									d5					
0	1	1	1	0	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=Rs

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`C`	Conditional
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s

Vector add halfwords

Add the two 16-bit halfwords of Rs to the two 16-bit halfwords of Rt. The results are optionally saturated to signed or unsigned 16-bit values.

Syntax	Behavior

`Rd=vaddh(Rs,Rt)[:sat]`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=[sat_16](Rs.h[i]+Rt.h[i]);`
	`};`

`Rd=vadduh(Rs,Rt):sat`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=usat_16(Rs.uh[i]+Rt.uh[i]);`
	`};`

Class: ALU32 (slots 0,1,2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vaddh(Rs,Rt)`	`Word32 Q6_R_vaddh_RR(Word32 Rs, Word32 Rt)`
`Rd=vaddh(Rs,Rt):sat`	`Word32 Q6_R_vaddh_RR_sat(Word32 Rs, Word32 Rt)`
`Rd=vadduh(Rs,Rt):sat`	`Word32 Q6_R_vadduh_RR_sat(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	1	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vaddh(Rs,Rt)
1	1	1	1	0	1	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vaddh(Rs,Rt):sat
1	1	1	1	0	1	1	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vadduh(Rs,Rt):sat

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`P`	Predicated
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector average halfwords

VAVGH adds the two 16-bit halfwords of Rs to the two 16-bit halfwords of Rd, and shifts the result right by 1 bit. Optionally, a rounding constant is added before shifting.

VNAVGH subtracts the two 16-bit halfwords of Rt from the two 16-bit halfwords of Rs, and shifts the result right by 1 bit. For vector negative average with rounding, see the XTYPE VNAVGH instruction.

Syntax	Behavior

`Rd=vavgh(Rs,Rt)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=((Rs.h[i]+Rt.h[i])>>1);`
	`};`

`Rd=vavgh(Rs,Rt):rnd`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=((Rs.h[i]+Rt.h[i]+1)>>1);`
	`};`

`Rd=vnavgh(Rt,Rs)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=((Rt.h[i]-Rs.h[i])>>1);`
	`};`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=vavgh(Rs,Rt)`	`Word32 Q6_R_vavgh_RR(Word32 Rs, Word32 Rt)`
`Rd=vavgh(Rs,Rt):rnd`	`Word32 Q6_R_vavgh_RR_rnd(Word32 Rs, Word32 Rt)`
`Rd=vnavgh(Rt,Rs)`	`Word32 Q6_R_vnavgh_RR(Word32 Rt, Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	1	1	1	-	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=vavgh(Rs,Rt)
1	1	1	1	0	1	1	1	-	0	1	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=vavgh(Rs,Rt):rnd
1	1	1	1	0	1	1	1	-	1	1	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rd=vnavgh(Rt,Rs)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`P`	Predicated
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector subtract halfwords

Subtract each of the two halfwords in 32-bit vector Rs from the corresponding halfword in vector Rt. Optionally saturate each 16-bit addition to either a signed or unsigned 16-bit value.

Applying saturation to the VSUBH instruction clamps the result to the signed range 0x8000 to 0x7fff, whereas applying saturation to VSUBUH ensures that the unsigned result is in the range 0 to 0xffff. When saturation is not needed, VSUBH should be used.

Syntax	Behavior

`Rd=vsubh(Rt,Rs)[:sat]`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=[sat_16](Rt.h[i]-Rs.h[i]);`
	`};`

`Rd=vsubuh(Rt,Rs):sat`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=usat_16(Rt.uh[i]-Rs.uh[i]);`
	`};`

Class: ALU32 (slots 0,1,2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vsubh(Rt,Rs)`	`Word32 Q6_R_vsubh_RR(Word32 Rt, Word32 Rs)`
`Rd=vsubh(Rt,Rs):sat`	`Word32 Q6_R_vsubh_RR_sat(Word32 Rt, Word32 Rs)`
`Rd=vsubuh(Rt,Rs):sat`	`Word32 Q6_R_vsubuh_RR_sat(Word32 Rt, Word32 Rs)`

Encoding

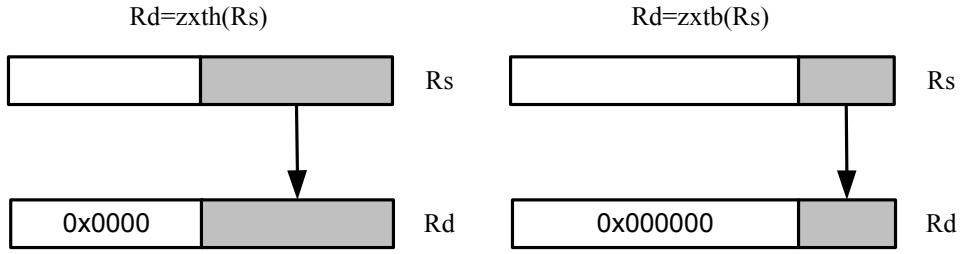
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	1	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vsubh(Rt,Rs)
1	1	1	1	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vsubh(Rt,Rs):sat
1	1	1	1	0	1	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vsubuh(Rt,Rs):sat

Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
P	Predicated
ICLASS	Instruction Class
Parse	Packet/Loop parse bits

Field name	Description
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Zero extend

Zero-extend the least significant byte or halfword from Rs and place the 32-bit result in Rd.



Syntax	Behavior
`Rd=zxtb(Rs)`	`Assembler mapped to: "Rd=and(Rs,#255)"`
`Rd=zxth(Rs)`	`Rd = zxt16->32(Rs);`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=zxtb(Rs)`	`Word32 Q6_R_zxtb_R(Word32 Rs)`
`Rd=zxth(Rs)`	`Word32 Q6_R_zxth_R(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse		C									d5					
0	1	1	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=zxth(Rs)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`C`	Conditional
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s

11.3.2 ALU32/PERM

The ALU32/PERM instruction subclass includes instructions which rearrange or perform format conversion on vector data types.

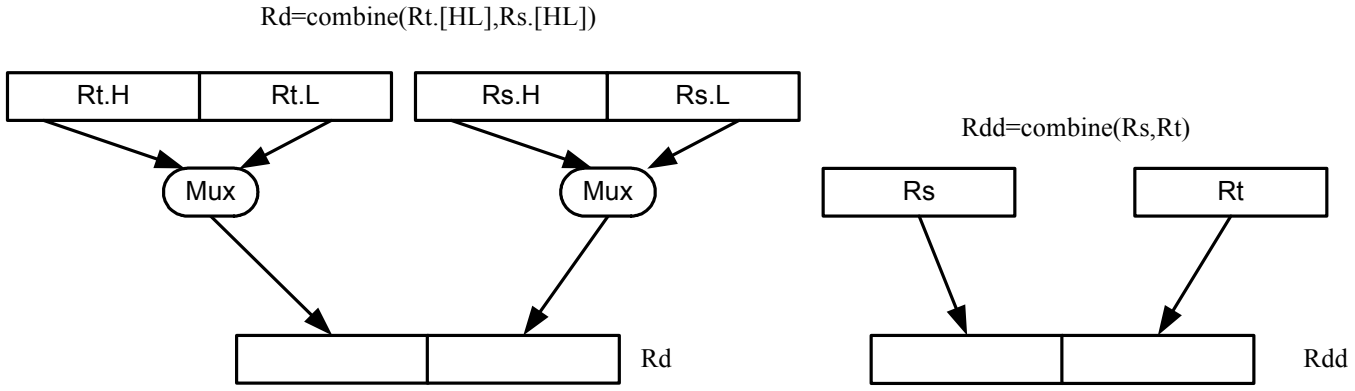
Combine words into doubleword

Combine halfwords or words into larger values.

In a halfword combine, either the high or low halfword of the first source register is transferred to the most-significant halfword of the destination register, while either the high or low halfword of the second source register is transferred to the least-significant halfword of the destination register. Source and destination registers are 32 bits.

In a word combine, the first source register is transferred to the most-significant word of the destination register, while the second source register is transferred to the least-significant word of the destination register. Source registers are 32 bits and the destination register is 64 bits.

In a variant of word combine, signed 8-bit immediate values (instead of registers) are transferred to the most- and least-significant words of the 64-bit destination register. Optionally one of the immediate values can be 32 bits.



Syntax	Behavior
`Rd=combine(Rt.[HL],Rs.[HL])`	`Rd = (Rt.uh[01]<<16) | Rs.uh[01];`

`Rdd=combine(#s8,#S8)`	`apply_extension(#s);`
	`Rdd.w[0]=#S;`
	`Rdd.w[1]=#s;`

`Rdd=combine(#s8,#U6)`	`apply_extension(#U);`
	`Rdd.w[0]=#U;`
	`Rdd.w[1]=#s;`

`Rdd=combine(#s8,Rs)`	`apply_extension(#s);`
	`Rdd.w[0]=Rs;`
	`Rdd.w[1]=#s;`

`Rdd=combine(Rs,#s8)`	`apply_extension(#s);`
	`Rdd.w[0]=#s;`
	`Rdd.w[1]=Rs;`

`Rdd=combine(Rs,Rt)`	`Rdd.w[0]=Rt;` `Rdd.w[1]=Rs;`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

Rd=combine(Rt.H,Rs.H)	Word32 Q6_R_combine_RhRh(Word32 Rt, Word32 Rs)
Rd=combine(Rt.H,Rs.L)	Word32 Q6_R_combine_RhRl(Word32 Rt, Word32 Rs)
Rd=combine(Rt.L,Rs.H)	Word32 Q6_R_combine_RlRh(Word32 Rt, Word32 Rs)
Rd=combine(Rt.L,Rs.L)	Word32 Q6_R_combine_RlRl(Word32 Rt, Word32 Rs)
Rdd=combine(#s8,#S8)	Word64 Q6_P_combine_II(Word32 Is8, Word32 IS8)
Rdd=combine(#s8,Rs)	Word64 Q6_P_combine_IR(Word32 Is8, Word32 Rs)
Rdd=combine(Rs,#s8)	Word64 Q6_P_combine_RI(Word32 Rs, Word32 Is8)
Rdd=combine(Rs,Rt)	Word64 Q6_P_combine_RR(Word32 Rs, Word32 Rt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse											d5					
0	1	1	1	0	0	1	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=combine(Rs,#s8)
0	1	1	1	0	0	1	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=combine(#s8,Rs)
ICLASS				Rs	MajOp			MinOp								Parse											d5					
0	1	1	1	1	1	0	0	0	**I**	**I**	**I**	**I**	**I**	**I**	**I**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=combine(#s8,#S8)
0	1	1	1	1	1	0	0	1	-	-	**I**	**I**	**I**	**I**	**I**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=combine(#s8,#U6)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=combine(Rt.H,Rs.H)
1	1	1	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=combine(Rt.H,Rs.L)
1	1	1	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=combine(Rt.L,Rs.H)
1	1	1	1	0	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=combine(Rt.L,Rs.L)
1	1	1	1	0	1	0	1	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=combine(Rs,Rt)

Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
Rs	No Rs read
MajOp	Major Opcode
MinOp	Minor Opcode
P	Predicated
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Mux

Select between two source registers based on the least-significant bit of a predicate register. If the bit is 1, transfer the first source register to the destination register; otherwise, transfer the second source register. Source and destination registers are 32 bits.

In a variant of mux, signed 8-bit immediate values can be used instead of registers for either or both source operands.

For 64-bit versions of this instruction, see the XTYPE vmux instruction.

Syntax	Behavior
`Rd=mux(Pu,#s8,#S8)`	`apply_extension(#s);` `(Pu[0]) ? (Rd=#s):(Rd=#S);`
`Rd=mux(Pu,#s8,Rs)`	`apply_extension(#s);` `(Pu[0]) ? (Rd=#s):(Rd=Rs);`
`Rd=mux(Pu,Rs,#s8)`	`apply_extension(#s);` `(Pu[0]) ? (Rd=Rs):(Rd=#s);`
`Rd=mux(Pu,Rs,Rt)`	`(Pu[0]) ? (Rd=Rs):(Rd=Rt);`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=mux(Pu,#s8,#S8)`	`Word32 Q6_R_mux_pII(Byte Pu, Word32 Is8, Word32 IS8)`
`Rd=mux(Pu,#s8,Rs)`	`Word32 Q6_R_mux_pIR(Byte Pu, Word32 Is8, Word32 Rs)`
`Rd=mux(Pu,Rs,#s8)`	`Word32 Q6_R_mux_pRI(Byte Pu, Word32 Rs, Word32 Is8)`
`Rd=mux(Pu,Rs,Rt)`	`Word32 Q6_R_mux_pRR(Byte Pu, Word32 Rs, Word32 Rt)`

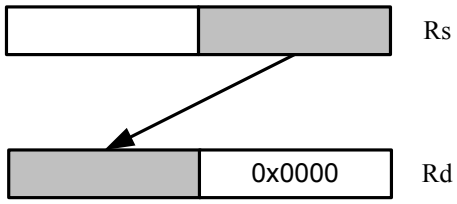
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp				u2		s5					Parse											d5					
0	1	1	1	0	0	1	1	0	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=mux(Pu,Rs,#s8)
0	1	1	1	0	0	1	1	1	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=mux(Pu,#s8,Rs)
ICLASS				Rs			u1									Parse											d5					
0	1	1	1	1	0	1	**u**	**u**	**l**	**l**	**l**	**l**	**l**	**l**	**l**	**P**	**P**	**l**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=mux(Pu,#s8,#S8)
ICLASS				P	MajOp						s5					Parse			t5						u2		d5					
1	1	1	1	0	1	0	0	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	**u**	**u**	**d**	**d**	**d**	**d**	**d**	Rd=mux(Pu,Rs,Rt)

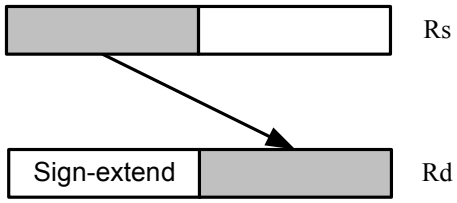
Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
Rs	No Rs read
MajOp	Major Opcode
P	Predicated
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
u2	Field to encode register u

Shift word by 16

ASLH performs an arithmetic left shift of the 32-bit source register by 16 bits (one halfword). The lower 16 bits of the destination are zero-filled.



ASRH performs an arithmetic right shift of the 32-bit source register by 16 bits (one halfword). The upper 16 bits of the destination are sign-extended.



Syntax	Behavior
`Rd=aslh(Rs)`	`Rd=Rs<<16;`
`Rd=asrh(Rs)`	`Rd=Rs>>16;`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=aslh(Rs)`	`Word32 Q6_R_aslh_R(Word32 Rs)`
`Rd=asrh(Rs)`	`Word32 Q6_R_asrh_R(Word32 Rs)`

Encoding

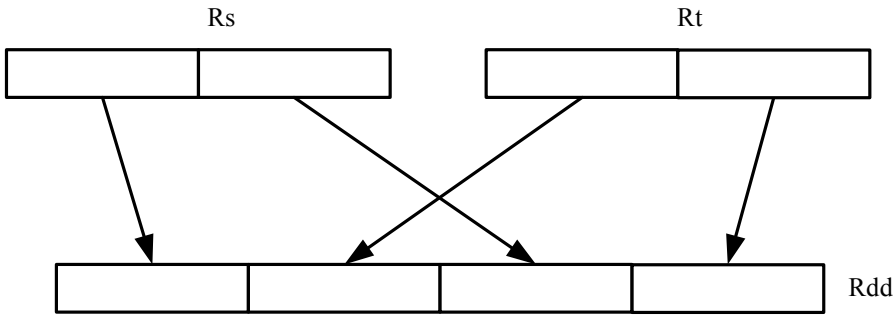
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse		C									d5					
0	1	1	1	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=aslh(Rs)
0	1	1	1	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=asrh(Rs)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`C`	Conditional

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s

Pack high and low halfwords

Pack together the most-significant halfwords from Rs and Rt into the most-significant word of register pair Rdd, and the least-significant halfwords from Rs and Rt into the least-significant halfword of Rdd.



Syntax	Behavior

`Rdd=packhl(Rs,Rt)`	`Rdd.h[0]=Rt.h[0];`
	`Rdd.h[1]=Rs.h[0];`
	`Rdd.h[2]=Rt.h[1];`
	`Rdd.h[3]=Rs.h[1];`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rdd=packhl(Rs,Rt)`	`Word64 Q6_P_packhl_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	1	0	1	1	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	d	d	d	d	d	Rdd=packhl(Rs,Rt)

Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
P	Predicated
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

11.3.3 ALU32/PRED

The ALU32/PRED instruction subclass includes instructions which perform conditional arithmetic and logical operations based on the values stored in a predicate register, and which produce predicate results. They are executable on any slot.

Conditional add

If the least-significant bit of predicate Pu is set, then add a 32-bit source register to either another register or an immediate value. The result is placed in 32-bit destination register. If the predicate is false, the instruction does nothing.

Syntax	Behavior

`if ([!]Pu[.new]) Rd=add(Rs,#s8)`	`if([!]Pu[.new][0]){`
	`  apply_extension(#s);`
	`  Rd=Rs+#s;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pu[.new]) Rd=add(Rs,Rt)`	`if([!]Pu[.new][0]){`
	`  Rd=Rs+Rt;`
	`} else {`
	`  NOP;`
	`};`

Class: ALU32 (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			PS	u2		s5					Parse		DN									d5					
0	1	1	1	0	1	0	0	0	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=add(Rs,#s8)
0	1	1	1	0	1	0	0	0	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=add(Rs,#s8)
0	1	1	1	0	1	0	0	1	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=add(Rs,#s8)
0	1	1	1	0	1	0	0	1	**u**	**u**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=add(Rs,#s8)
ICLASS				P	MajOp			MinOp			s5					Parse		DN	t5					PS	u2		d5					
1	1	1	1	1	0	1	1	0	-	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=add(Rs,Rt)
1	1	1	1	1	0	1	1	0	-	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=add(Rs,Rt)
1	1	1	1	1	0	1	1	0	-	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=add(Rs,Rt)
1	1	1	1	1	0	1	1	0	-	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=add(Rs,Rt)

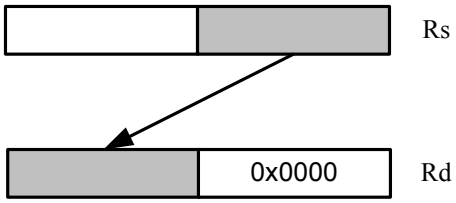
Field name	Description
MajOp	Major Opcode
MinOp	Minor Opcode
Rs	No Rs read
DN	Dot-new
PS	Predicate sense
DN	Dot-new
MajOp	Major Opcode
MinOp	Minor Opcode
P	Predicated
PS	Predicate sense
ICLASS	Instruction Class

Field name	Description
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`u2`	Field to encode register u

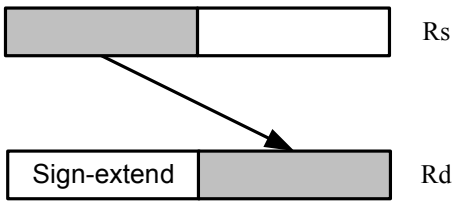
Conditional shift halfword

Conditionally shift a halfword.

ASLH performs an arithmetic left shift of the 32-bit source register by 16 bits (one halfword). The lower 16 bits of the destination are zero-filled.



ASRH performs an arithmetic right shift of the 32-bit source register by 16 bits (one halfword). The upper 16 bits of the destination are sign-extended.



Syntax	Behavior

`if ([!]Pu[.new]) Rd=aslh(Rs)`	`if([!]Pu[.new][0]){`
	`  Rd=Rs<<16;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pu[.new]) Rd=asrh(Rs)`	`if([!]Pu[.new][0]){`
	`  Rd=Rs>>16;`
	`} else {`
	`  NOP;`
	`};`

Class: ALU32 (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse		C		S	dn	u2					d5					
0	1	1	1	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=aslh(Rs)
0	1	1	1	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=aslh(Rs)
0	1	1	1	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=aslh(Rs)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	1	1	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	1	1	**u**	**u**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=aslh(Rs)
0	1	1	1	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	0	0	**u**	**u**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=asrh(Rs)
0	1	1	1	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	0	1	**u**	**u**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=asrh(Rs)
0	1	1	1	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	1	0	**u**	**u**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=asrh(Rs)
0	1	1	1	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**-**	1	1	**u**	**u**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=asrh(Rs)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`C`	Conditional
`S`	Predicate sense
`dn`	Dot-new
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`u2`	Field to encode register u

Conditional combine

If the least-significant bit of predicate Pu is set, then the most-significant word of destination Rdd is taken from the first source register Rs, while the least-significant word is taken from the second source register Rt. If the predicate is false, this instruction does nothing.

Syntax	Behavior
`if ([!]Pu[.new]) Rdd=combine(Rs,Rt)`	`if ([!]Pu[.new][0]) { Rdd.w[0]=Rt; Rdd.w[1]=Rs; } else { NOP; };`

Class: ALU32 (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				P	MajOp						s5					Parse		DN	t5						PS	u2		d5					
1	1	1	1	1	1	0	1	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rdd=combine(Rs,Rt)
1	1	1	1	1	1	0	1	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rdd=combine(Rs,Rt)
1	1	1	1	1	1	0	1	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rdd=combine(Rs,Rt)
1	1	1	1	1	1	0	1	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rdd=combine(Rs,Rt)

Field name	Description
DN	Dot-new
MajOp	Major Opcode
P	Predicated
PS	Predicate sense
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

Conditional logical operations

If the least-significant bit of predicate Pu is set, then do a logical operation on the source values. The result is placed in 32-bit destination register. If the predicate is false, the instruction does nothing.

Syntax	Behavior

`if ([!]Pu[.new]) Rd=and(Rs,Rt)`	`if([!]Pu[.new][0]){`
	`  Rd=Rs&Rt;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pu[.new]) Rd=or(Rs,Rt)`	`if([!]Pu[.new][0]){`
	`  Rd=Rs|Rt;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pu[.new]) Rd=xor(Rs,Rt)`	`if([!]Pu[.new][0]){`
	`  Rd=Rs^Rt;`
	`} else {`
	`  NOP;`
	`};`

Class: ALU32 (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				P	MajOp			MinOp			s5					Parse		DN	t5						PS	u2		d5					
1	1	1	1	1	0	0	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=and(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=and(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=and(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=and(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=or(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=or(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=or(Rs,Rt)
1	1	1	1	1	0	0	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=or(Rs,Rt)
1	1	1	1	1	0	0	1	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=xor(Rs,Rt)
1	1	1	1	1	0	0	1	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=xor(Rs,Rt)
1	1	1	1	1	0	0	1	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=xor(Rs,Rt)
1	1	1	1	1	0	0	1	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=xor(Rs,Rt)

Field name	Description
DN	Dot-new
MajOp	Major Opcode
MinOp	Minor Opcode
P	Predicated
PS	Predicate sense

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

Conditional subtract

If the least-significant bit of predicate Pu is set, then subtract a 32-bit source register Rt from register Rs. The result is placed in a 32-bit destination register. If the predicate is false, the instruction does nothing.

Syntax

```
if ([!]Pu[.new]) Rd=sub(Rt,Rs)
```

Behavior

```
if([!]Pu[.new][0]){
    Rd=Rt-Rs;
} else {
    NOP;
};
```

Class: ALU32 (slots 0,1,2,3)

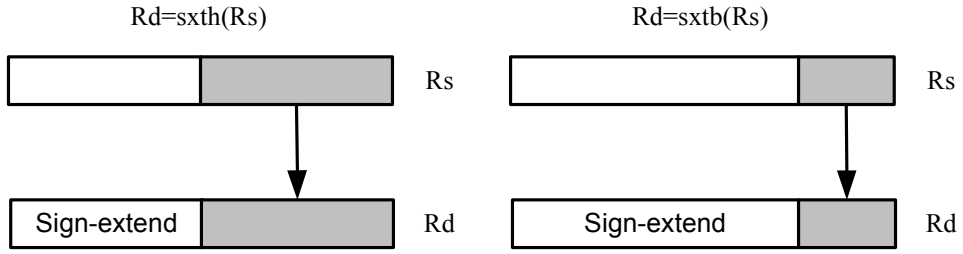
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				P	MajOp			MinOp			s5					Parse		DN	t5					PS	u2		d5					
1	1	1	1	1	0	1	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=sub(Rt,Rs)
1	1	1	1	1	0	1	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=sub(Rt,Rs)
1	1	1	1	1	0	1	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=sub(Rt,Rs)
1	1	1	1	1	0	1	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=sub(Rt,Rs)

Field name	Description
DN	Dot-new
MajOp	Major Opcode
MinOp	Minor Opcode
P	Predicated
PS	Predicate sense
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

Conditional sign extend

Conditionally sign-extend the least-significant byte or halfword from Rs and put the 32-bit result in Rd.



Syntax	Behavior
`if ([!]Pu[.new]) Rd=sxtb(Rs)`	`if([!]Pu[.new][0]){ Rd=sxt8->32(Rs); } else { NOP; };`
`if ([!]Pu[.new]) Rd=sxth(Rs)`	`if([!]Pu[.new][0]){ Rd=sxt16->32(Rs); } else { NOP; };`

Class: ALU32 (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse		C		S	dn	u2					d5					
0	1	1	1	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=sxtb(Rs)
0	1	1	1	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=sxtb(Rs)
0	1	1	1	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=sxtb(Rs)
0	1	1	1	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=sxtb(Rs)
0	1	1	1	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=sxth(Rs)
0	1	1	1	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=sxth(Rs)
0	1	1	1	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=sxth(Rs)
0	1	1	1	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=sxth(Rs)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`C`	Conditional
`S`	Predicate sense
`dn`	Dot-new

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
u2	Field to encode register u

Conditional transfer

If the LSB of predicate Pu is set, transfer register Rs or a signed immediate into destination Rd. If the predicate is false, this instruction does nothing.

Syntax	Behavior

`if ([!]Pu[.new]) Rd=#s12`	`apply_extension(#s);`
	`if ([!]Pu[.new][0]) Rd=#s;`
	`else NOP;`

`if ([!]Pu[.new]) Rd=Rs`	`Assembler mapped to: "if ([!]Pu[.new]) Rd=add(Rs,#0)"`
`if ([!]Pu[.new]) Rdd=Rss`	`Assembler mapped to: "if ([!]Pu[.new]) Rdd=combine(Rss.H32,Rss.L32)"`

Class: ALU32 (slots 0,1,2,3)

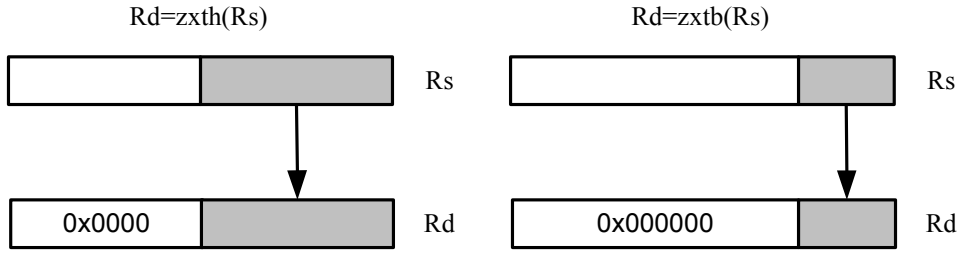
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			PS	u2							Parse		DN									d5					
0	1	1	1	1	1	1	0	0	**u**	**u**	0	**i**	**i**	**i**	**i**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=#s12
0	1	1	1	1	1	1	0	0	**u**	**u**	0	**i**	**i**	**i**	**i**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=#s12
0	1	1	1	1	1	1	0	1	**u**	**u**	0	**i**	**i**	**i**	**i**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=#s12
0	1	1	1	1	1	1	0	1	**u**	**u**	0	**i**	**i**	**i**	**i**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=#s12

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`DN`	Dot-new
`PS`	Predicate sense
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`u2`	Field to encode register u

Conditional zero extend

Conditionally zero-extend the least-significant byte or halfword from Rs and put the 32-bit result in Rd.



Syntax	Behavior
`if ([!]Pu[.new]) Rd=zxtb(Rs)`	`if([!]Pu[.new][0]){ Rd=zxt`$_{8->32}$`(Rs); } else { NOP; };`
`if ([!]Pu[.new]) Rd=zxth(Rs)`	`if([!]Pu[.new][0]){ Rd=zxt`$_{16->32}$`(Rs); } else { NOP; };`

Class: ALU32 (slots 0,1,2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse		C		S	dn	u2					d5					
0	1	1	1	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=zxtb(Rs)
0	1	1	1	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=zxtb(Rs)
0	1	1	1	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=zxtb(Rs)
0	1	1	1	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=zxtb(Rs)
0	1	1	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu) Rd=zxth(Rs)
0	1	1	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	0	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (Pu.new) Rd=zxth(Rs)
0	1	1	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	0	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu) Rd=zxth(Rs)
0	1	1	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	-	1	1	**u**	**u**	-	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pu.new) Rd=zxth(Rs)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`C`	Conditional
`S`	Predicate sense
`dn`	Dot-new

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
u2	Field to encode register u

Compare

The register form compares two 32-bit registers for unsigned greater than, greater than, or equal.

The immediate form compares a register against a signed or unsigned immediate value. The 8-bit predicate register Pd is set to all 1's or all 0's depending on the result. For 64-bit versions of this instruction, see the XTYPE compare instructions.

Syntax	Behavior
`Pd=[!]cmp.eq(Rs,#s10)`	`apply_extension(#s);` `Pd=Rs[!]=#s ? 0xff : 0x00;`
`Pd=[!]cmp.eq(Rs,Rt)`	`Pd=Rs[!]=Rt ? 0xff : 0x00;`
`Pd=[!]cmp.gt(Rs,#s10)`	`apply_extension(#s);` `Pd=Rs[!]>#s ? 0xff : 0x00;`
`Pd=[!]cmp.gt(Rs,Rt)`	`Pd=Rs[!]>Rt ? 0xff : 0x00;`
`Pd=[!]cmp.gtu(Rs,#u9)`	`apply_extension(#u);` `Pd=Rs.uw[0][!]>#u ? 0xff : 0x00;`
`Pd=[!]cmp.gtu(Rs,Rt)`	`Pd=Rs.uw[0][!]>Rt.uw[0] ? 0xff : 0x00;`
`Pd=cmp.ge(Rs,#s8)`	`Assembler mapped to: "Pd=cmp.gt(Rs,#s8-1)"`

`Pd=cmp.geu(Rs,#u8)`	`if ("#u8==0") {`
	`  Assembler mapped to: "Pd=cmp.eq(Rs,Rs)";`
	`} else {`
	`  Assembler mapped to: "Pd=cmp.gtu(Rs,#u8-1)";`
	`};`

`Pd=cmp.lt(Rs,Rt)`	`Assembler mapped to: "Pd=cmp.gt(Rt,Rs)"`
`Pd=cmp.ltu(Rs,Rt)`	`Assembler mapped to: "Pd=cmp.gtu(Rt,Rs)"`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Pd=!cmp.eq(Rs,#s10)`	`Byte Q6_p_not_cmp_eq_RI(Word32 Rs, Word32 Is10)`
`Pd=!cmp.eq(Rs,Rt)`	`Byte Q6_p_not_cmp_eq_RR(Word32 Rs, Word32 Rt)`
`Pd=!cmp.gt(Rs,#s10)`	`Byte Q6_p_not_cmp_gt_RI(Word32 Rs, Word32 Is10)`
`Pd=!cmp.gt(Rs,Rt)`	`Byte Q6_p_not_cmp_gt_RR(Word32 Rs, Word32 Rt)`
`Pd=!cmp.gtu(Rs,#u9)`	`Byte Q6_p_not_cmp_gtu_RI(Word32 Rs, Word32 Iu9)`
`Pd=!cmp.gtu(Rs,Rt)`	`Byte Q6_p_not_cmp_gtu_RR(Word32 Rs, Word32 Rt)`
`Pd=cmp.eq(Rs,#s10)`	`Byte Q6_p_cmp_eq_RI(Word32 Rs, Word32 Is10)`
`Pd=cmp.eq(Rs,Rt)`	`Byte Q6_p_cmp_eq_RR(Word32 Rs, Word32 Rt)`
`Pd=cmp.ge(Rs,#s8)`	`Byte Q6_p_cmp_ge_RI(Word32 Rs, Word32 Is8)`
`Pd=cmp.geu(Rs,#u8)`	`Byte Q6_p_cmp_geu_RI(Word32 Rs, Word32 Iu8)`
`Pd=cmp.gt(Rs,#s10)`	`Byte Q6_p_cmp_gt_RI(Word32 Rs, Word32 Is10)`

`Pd=cmp.gt(Rs,Rt)`	`Byte Q6_p_cmp_gt_RR(Word32 Rs, Word32 Rt)`
`Pd=cmp.gtu(Rs,#u9)`	`Byte Q6_p_cmp_gtu_RI(Word32 Rs, Word32 Iu9)`
`Pd=cmp.gtu(Rs,Rt)`	`Byte Q6_p_cmp_gtu_RR(Word32 Rs, Word32 Rt)`
`Pd=cmp.lt(Rs,Rt)`	`Byte Q6_p_cmp_lt_RR(Word32 Rs, Word32 Rt)`
`Pd=cmp.ltu(Rs,Rt)`	`Byte Q6_p_cmp_ltu_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse														d2		
0	1	1	1	0	1	0	1	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	Pd=cmp.eq(Rs,#s10)
0	1	1	1	0	1	0	1	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	1	0	0	**d**	**d**	Pd=!cmp.eq(Rs,#s10)
0	1	1	1	0	1	0	1	0	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	Pd=cmp.gt(Rs,#s10)
0	1	1	1	0	1	0	1	0	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	1	0	0	**d**	**d**	Pd=!cmp.gt(Rs,#s10)
0	1	1	1	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	Pd=cmp.gtu(Rs,#u9)
0	1	1	1	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	1	0	0	**d**	**d**	Pd=!cmp.gtu(Rs,#u9)
ICLASS				P	MajOp			MinOp			s5					Parse			t5											d2		
1	1	1	1	0	0	1	0	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	0	0	0	**d**	**d**	Pd=cmp.eq(Rs,Rt)
1	1	1	1	0	0	1	0	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	1	0	0	**d**	**d**	Pd=!cmp.eq(Rs,Rt)
1	1	1	1	0	0	1	0	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	0	0	0	**d**	**d**	Pd=cmp.gt(Rs,Rt)
1	1	1	1	0	0	1	0	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	1	0	0	**d**	**d**	Pd=!cmp.gt(Rs,Rt)
1	1	1	1	0	0	1	0	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	0	0	0	**d**	**d**	Pd=cmp.gtu(Rs,Rt)
1	1	1	1	0	0	1	0	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	1	0	0	**d**	**d**	Pd=!cmp.gtu(Rs,Rt)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`P`	Predicated
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Compare to general register

The register form compares two 32-bit registers for unsigned greater than, greater than, or equal. The immediate form compares a register against a signed or unsigned immediate value. The resulting zero or one is placed in a general register.

Syntax	Behavior
`Rd=[!]cmp.eq(Rs,#s8)`	`apply_extension(#s);` `Rd=(Rs[!]=#s);`
`Rd=[!]cmp.eq(Rs,Rt)`	`Rd=(Rs[!]=Rt);`

Class: ALU32 (slots 0,1,2,3)

Intrinsics

`Rd=!cmp.eq(Rs,#s8)`	`Word32 Q6_R_not_cmp_eq_RI(Word32 Rs, Word32 Is8)`
`Rd=!cmp.eq(Rs,Rt)`	`Word32 Q6_R_not_cmp_eq_RR(Word32 Rs, Word32 Rt)`
`Rd=cmp.eq(Rs,#s8)`	`Word32 Q6_R_cmp_eq_RI(Word32 Rs, Word32 Is8)`
`Rd=cmp.eq(Rs,Rt)`	`Word32 Q6_R_cmp_eq_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Rs	MajOp			MinOp			s5					Parse											d5					
0	1	1	1	0	0	1	1	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=cmp.eq(Rs,#s8)
0	1	1	1	0	0	1	1	-	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=!cmp.eq(Rs,#s8)
ICLASS				P	MajOp			MinOp			s5					Parse			t5								d5					
1	1	1	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=cmp.eq(Rs,Rt)
1	1	1	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=!cmp.eq(Rs,Rt)

Field name	Description
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Rs`	No Rs read
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`P`	Predicated
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

11.4 CR

The CR instruction class includes instructions which manage control registers, including hardware looping, modulo addressing, and status flags.

CR instructions are executable on slot 3.

End loop instructions

The endloop instructions mark the end of a hardware loop. If the Loop Count (LC) register indicates that a loop should continue to iterate, the LC register is decremented and the program flow changes to the address in the Start Address (SA) register.

The endloopN instruction is actually a pseudo-instruction encoded in bits 15:14 of each instruction. Therefore, no distinct 32-bit encoding exists for this instruction.

Syntax	Behavior

endloop0	if (USR.LPCFG) {
	  if (USR.LPCFG==1) {
	  P3=0xff;
	  };
	  USR.LPCFG=USR.LPCFG-1;
	};
	if (LC0>1) {
	  PC=SA0;
	  LC0=LC0-1;
	};

endloop01	if (USR.LPCFG) {
	  if (USR.LPCFG==1) {
	  P3=0xff;
	  };
	  USR.LPCFG=USR.LPCFG-1;
	};
	if (LC0>1) {
	  PC=SA0;
	  LC0=LC0-1;
	} else {
	  if (LC1>1) {
	  PC=SA1;
	  LC1=LC1-1;
	  };
	};

endloop1	if (LC1>1) {
	  PC=SA1;
	  LC1=LC1-1;
	};

Class: N/A

Notes

- This instruction cannot be grouped in a packet with any program flow instructions.
- The Next PC value is the address immediately following the last instruction in the packet containing this instruction.
- The PC value is the address of the start of the packet

Corner detection acceleration

The FASTCORNER9 instruction takes the Ps and Pt values and treats them as a circular bit string. If any contiguous nine bits are set around the circle, the result is true, false otherwise. The sense may be optionally inverted. This instruction is used to accelerate FAST corner detection.

Syntax	Behavior

`Pd=[!]fastcorner9(Ps,Pt)`	`tmp.h[0]=(Ps<<8)|Pt;`
	`tmp.h[1]=(Ps<<8)|Pt;`
	`for (i = 1; i < 9; i++) {`
	`  tmp &= tmp >> 1;`
	`};`
	`Pd = tmp == 0 ? 0xff : 0x00;`

Class: CR (slot 2,3)

Notes

- This instruction may execute on either slot2 or slot3, even though it is a CR-type

Intrinsics

`Pd=!fastcorner9(Ps,Pt)`	`Byte Q6_p_not_fastcorner9_pp(Byte Ps, Byte Pt)`
`Pd=fastcorner9(Ps,Pt)`	`Byte Q6_p_fastcorner9_pp(Byte Ps, Byte Pt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm									s2		Parse						t2								d2		
0	1	1	0	1	0	1	1	0	0	0	0	-	-	**s**	**s**	**P**	**P**	1	-	-	-	**t**	**t**	1	-	-	1	-	-	**d**	**d**	Pd=fastcorner9(Ps,Pt)
0	1	1	0	1	0	1	1	0	0	0	1	-	-	**s**	**s**	**P**	**P**	1	-	-	-	**t**	**t**	1	-	-	1	-	-	**d**	**d**	Pd=!fastcorner9(Ps,Pt)

Field name	Description
`sm`	Supervisor mode only
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`s2`	Field to encode register s
`t2`	Field to encode register t

Logical reductions on predicates

The ANY8 instruction sets a destination predicate register to 0xff if any of the low 8 bits in source predicate register Ps are set. Otherwise, the predicate is set to 0x00.

The ALL8 instruction sets a destination predicate register to 0xff if all of the low 8 bits in the source predicate register Ps are set. Otherwise, the predicate is set to 0x00.

Syntax	Behavior
`Pd=all8(Ps)`	`(Ps==0xff) ? (Pd=0xff) : (Pd=0x00);`
`Pd=any8(Ps)`	`Ps ? (Pd=0xff) : (Pd=0x00);`

Class: CR (slot 2,3)

Notes

- This instruction may execute on either slot2 or slot3, even though it is a CR-type

Intrinsics

`Pd=all8(Ps)`	`Byte Q6_p_all8_p(Byte Ps)`
`Pd=any8(Ps)`	`Byte Q6_p_any8_p(Byte Ps)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm									s2		Parse														d2		
0	1	1	0	1	0	1	1	1	0	0	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	-	-	-	**d**	**d**	Pd=any8(Ps)
0	1	1	0	1	0	1	1	1	0	1	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	-	-	-	**d**	**d**	Pd=all8(Ps)

Field name	Description
`sm`	Supervisor mode only
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`s2`	Field to encode register s

Looping instructions

loopN is a single instruction which sets up a hardware loop. The N in the instruction name indicates the set of loop registers to use. Loop0 is the innermost loop, while loop1 is the outer loop.The loopN instruction first sets the Start Address (SA) register based on a PC-relative immediate add. The relative immediate is added to the PC and stored in SA. The Loop Count (LC) register is set to either an unsigned immediate or to a register value.

Syntax	Behavior

`loop0(#r7:2,#U10)`	`apply_extension(#r);`
	`#r=#r & ~0x3;`
	`SA0=PC+#r;`
	`LC0=#U;`
	`USR.LPCFG=0;`

`loop0(#r7:2,Rs)`	`apply_extension(#r);`
	`#r=#r & ~0x3;`
	`SA0=PC+#r;`
	`LC0=Rs;`
	`USR.LPCFG=0;`

`loop1(#r7:2,#U10)`	`apply_extension(#r);`
	`#r=#r & ~0x3;`
	`SA1=PC+#r;`
	`LC1=#U;`

`loop1(#r7:2,Rs)`	`apply_extension(#r);`
	`#r=#r & ~0x3;`
	`SA1=PC+#r;`
	`LC1=Rs;`

Class: CR (slot 3)

Notes

- This instruction cannot execute in the last address of a hardware loop.
- The Next PC value is the address immediately following the last instruction in the packet containing this instruction.
- The PC value is the address of the start of the packet
- A PC-relative address is formed by taking the decoded immediate value and adding it to the current PC value.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm						s5					Parse																
0	1	1	0	0	0	0	0	0	0	0	s	s	s	s	s	P	P	-	i	i	i	i	i	-	-	-	i	i	-	-	-	loop0(#r7:2,Rs)
0	1	1	0	0	0	0	0	0	0	1	s	s	s	s	s	P	P	-	i	i	i	i	i	-	-	-	i	i	-	-	-	loop1(#r7:2,Rs)
ICLASS					sm											Parse																
0	1	1	0	1	0	0	1	0	0	0	I	I	I	I	I	P	P	-	i	i	i	i	i	I	I	I	i	i	-	I	I	loop0(#r7:2,#U10)
0	1	1	0	1	0	0	1	0	0	1	I	I	I	I	I	P	P	-	i	i	i	i	i	I	I	I	i	i	-	I	I	loop1(#r7:2,#U10)

Field name	Description
sm	Supervisor mode only
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s

Add to PC

Add an immediate value to the Program Counter (PC) and place the result in a destination register. This instruction is typically used with a constant extender to add a 32-bit immediate value to PC.

Syntax	Behavior
`Rd=add(pc,#u6)`	`Rd=PC+apply_extension(#u);`

Class: CR (slot 3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm											Parse											d5					
0	1	1	0	1	0	1	0	0	1	0	0	1	0	0	1	**P**	**P**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=add(pc,#u6)

Field name	Description
`sm`	Supervisor mode only
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d

Pipelined loop instructions

spNloop0 is a single instruction which sets up a hardware loop with automatic predicate control. This features saves code size by enabling many software pipelined loops to be generated without prologue code. Upon executing this instruction, the P3 register is automatically cleared. After the loop has been executed N times (where N is selectable from 1-3), the P3 register is set. The intent is that store instructions in the loop are predicated with P3 and thus not enabled during the pipeline warm-up.

In the spNloop0 instruction the loop 0 (inner-loop) registers are used. This instruction sets the Start Address (SA0) register based on a PC-relative immediate add. The relative immediate is added to the PC and stored in SA0. The Loop Count (LC0) is set to either an unsigned immediate or to a register value. The predicate P3 is cleared. The USR.LPCFG bits are set based on the N value.

Syntax	Behavior

`p3=sp1loop0(#r7:2,#U10)`	`apply_extension(#r);`
	`#r=#r & ~0x3;`
	`SA0=PC+#r;`
	`LC0=#U;`
	`USR.LPCFG=1;`
	`P3=0;`

`p3=sp1loop0(#r7:2,Rs)`	`apply_extension(#r);`
	`#r=#r & ~0x3;`
	`SA0=PC+#r;`
	`LC0=Rs;`
	`USR.LPCFG=1;`
	`P3=0;`

`p3=sp2loop0(#r7:2,#U10)`	`apply_extension(#r);`
	`#r=#r & ~0x3;`
	`SA0=PC+#r;`
	`LC0=#U;`
	`USR.LPCFG=2;`
	`P3=0;`

`p3=sp2loop0(#r7:2,Rs)`	`apply_extension(#r);`
	`#r=#r & ~0x3;`
	`SA0=PC+#r;`
	`LC0=Rs;`
	`USR.LPCFG=2;`
	`P3=0;`

`p3=sp3loop0(#r7:2,#U10)`	`apply_extension(#r);`
	`#r=#r & ~0x3;`
	`SA0=PC+#r;`
	`LC0=#U;`
	`USR.LPCFG=3;`
	`P3=0;`

`p3=sp3loop0(#r7:2,Rs)`	`apply_extension(#r);`
	`#r=#r & ~0x3;`
	`SA0=PC+#r;`
	`LC0=Rs;`
	`USR.LPCFG=3;`
	`P3=0;`

Class: CR (slot 3)

Notes

- The predicate generated by this instruction can not be used as a .new predicate, nor can it be automatically ANDed with another predicate.
- This instruction cannot execute in the last address of a hardware loop.
- The Next PC value is the address immediately following the last instruction in the packet containing this instruction.
- The PC value is the address of the start of the packet
- A PC-relative address is formed by taking the decoded immediate value and adding it to the current PC value.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm						s5					Parse																	
0	1	1	0	0	0	0	0	1	0	1	s	s	s	s	s	P	P	-	i	i	i	i	i	-	-	-	i	i	-	-	-	p3=sp1loop0(#r7:2,Rs)
0	1	1	0	0	0	0	0	1	1	0	s	s	s	s	s	P	P	-	i	i	i	i	i	-	-	-	i	i	-	-	-	p3=sp2loop0(#r7:2,Rs)
0	1	1	0	0	0	0	0	1	1	1	s	s	s	s	s	P	P	-	i	i	i	i	i	-	-	-	i	i	-	-	-	p3=sp3loop0(#r7:2,Rs)
ICLASS					sm											Parse																
0	1	1	0	1	0	0	1	1	0	1	I	I	I	I	I	P	P	-	i	i	i	i	i	I	I	I	i	i	-	I	I	p3=sp1loop0(#r7:2,#U10)
0	1	1	0	1	0	0	1	1	1	0	I	I	I	I	I	P	P	-	i	i	i	i	i	I	I	I	i	i	-	I	I	p3=sp2loop0(#r7:2,#U10)
0	1	1	0	1	0	0	1	1	1	1	I	I	I	I	I	P	P	-	i	i	i	i	i	I	I	I	i	i	-	I	I	p3=sp3loop0(#r7:2,#U10)

Field name	Description
sm	Supervisor mode only
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s

Logical operations on predicates

Perform bitwise logical operations on predicate registers.

Syntax	Behavior
`Pd=Ps`	`Assembler mapped to: "Pd=or(Ps,Ps)"`
`Pd=and(Ps,and(Pt,[!]Pu))`	`Pd = Ps & Pt & (~Pu);`
`Pd=and(Ps,or(Pt,[!]Pu))`	`Pd = Ps & (Pt | (~Pu));`
`Pd=and(Pt,[!]Ps)`	`Pd=Pt & (~Ps);`
`Pd=not(Ps)`	`Pd=~Ps;`
`Pd=or(Ps,and(Pt,[!]Pu))`	`Pd = Ps | (Pt & (~Pu));`
`Pd=or(Ps,or(Pt,[!]Pu))`	`Pd = Ps | Pt | (~Pu);`
`Pd=or(Pt,[!]Ps)`	`Pd=Pt | (~Ps);`
`Pd=xor(Ps,Pt)`	`Pd=Ps ^ Pt;`

Class: CR (slot 2,3)

Notes

- This instruction may execute on either slot2 or slot3, even though it is a CR-type

Intrinsics

`Pd=Ps`	`Byte Q6_p_equals_p(Byte Ps)`
`Pd=and(Ps,and(Pt,!Pu))`	`Byte Q6_p_and_and_ppnp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=and(Ps,and(Pt,Pu))`	`Byte Q6_p_and_and_ppp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=and(Ps,or(Pt,!Pu))`	`Byte Q6_p_and_or_ppnp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=and(Ps,or(Pt,Pu))`	`Byte Q6_p_and_or_ppp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=and(Pt,!Ps)`	`Byte Q6_p_and_pnp(Byte Pt, Byte Ps)`
`Pd=and(Pt,Ps)`	`Byte Q6_p_and_pp(Byte Pt, Byte Ps)`
`Pd=not(Ps)`	`Byte Q6_p_not_p(Byte Ps)`
`Pd=or(Ps,and(Pt,!Pu))`	`Byte Q6_p_or_and_ppnp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=or(Ps,and(Pt,Pu))`	`Byte Q6_p_or_and_ppp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=or(Ps,or(Pt,!Pu))`	`Byte Q6_p_or_or_ppnp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=or(Ps,or(Pt,Pu))`	`Byte Q6_p_or_or_ppp(Byte Ps, Byte Pt, Byte Pu)`
`Pd=or(Pt,!Ps)`	`Byte Q6_p_or_pnp(Byte Pt, Byte Ps)`
`Pd=or(Pt,Ps)`	`Byte Q6_p_or_pp(Byte Pt, Byte Ps)`
`Pd=xor(Ps,Pt)`	`Byte Q6_p_xor_pp(Byte Ps, Byte Pt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm									s2		Parse						t2								d2		
0	1	1	0	1	0	1	1	0	0	0	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=and(Pt,Ps)
ICLASS					sm									s2		Parse						t2		u2						d2		
0	1	1	0	1	0	1	1	0	0	0	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=and(Ps,and(Pt,Pu))
ICLASS					sm									s2		Parse						t2								d2		
0	1	1	0	1	0	1	1	0	0	1	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=or(Pt,Ps)
ICLASS					sm									s2		Parse						t2		u2						d2		
0	1	1	0	1	0	1	1	0	0	1	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=and(Ps,or(Pt,Pu))
ICLASS					sm									s2		Parse						t2								d2		
0	1	1	0	1	0	1	1	0	1	0	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=xor(Ps,Pt)
ICLASS					sm									s2		Parse						t2		u2						d2		
0	1	1	0	1	0	1	1	0	1	0	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=or(Ps,and(Pt,Pu))
ICLASS					sm									s2		Parse						t2								d2		
0	1	1	0	1	0	1	1	0	1	1	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=and(Pt,!Ps)
ICLASS					sm									s2		Parse						t2		u2						d2		
0	1	1	0	1	0	1	1	0	1	1	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=or(Ps,or(Pt,Pu))
0	1	1	0	1	0	1	1	1	0	0	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=and(Ps,and(Pt,!Pu))
0	1	1	0	1	0	1	1	1	0	1	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=and(Ps,or(Pt,!Pu))
ICLASS					sm									s2		Parse														d2		
0	1	1	0	1	0	1	1	1	1	0	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	-	-	-	**d**	**d**	Pd=not(Ps)
ICLASS					sm									s2		Parse						t2		u2						d2		
0	1	1	0	1	0	1	1	1	1	0	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=or(Ps,and(Pt,!Pu))
ICLASS					sm									s2		Parse						t2								d2		
0	1	1	0	1	0	1	1	1	1	1	0	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=or(Pt,!Ps)
ICLASS					sm									s2		Parse						t2		u2						d2		
0	1	1	0	1	0	1	1	1	1	1	1	-	-	**s**	**s**	**P**	**P**	0	-	-	-	**t**	**t**	**u**	**u**	-	-	-	-	**d**	**d**	Pd=or(Ps,or(Pt,!Pu))

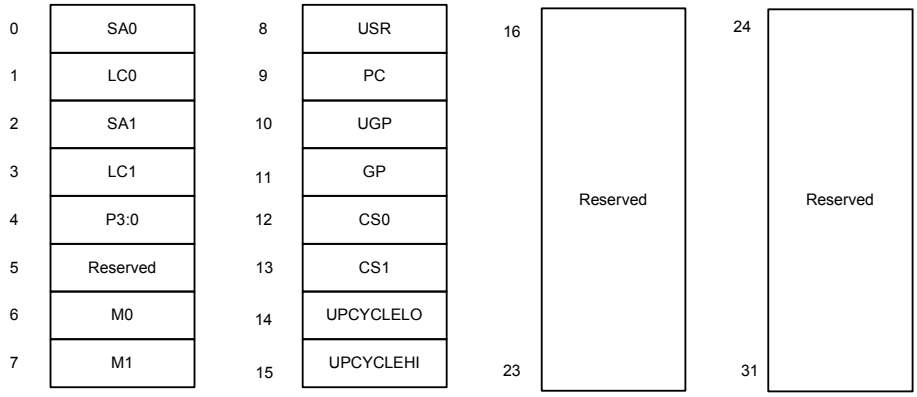
Field name	Description
`sm`	Supervisor mode only
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`s2`	Field to encode register s
`t2`	Field to encode register t
`u2`	Field to encode register u

User control register transfer

Move 32- or 64-bit values between a user control register and a general register. The user control registers include SA, LC, Predicates, M, USR, PC, UGP, GP, and CS, and UPCYCLE. The figure shows the user control registers and their register field encodings.

Registers can be moved as singles or as aligned 64-bit pairs.

Note that the PC register is not writable. A program flow instruction must be used to change the PC value.



Syntax	Behavior
`Cd=Rs`	`Cd=Rs;`
`Cdd=Rss`	`Cdd=Rss;`
`Rd=Cs`	`Rd=Cs;`
`Rdd=Css`	`Rdd=Css;`

Class: CR (slot 3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm						s5					Parse											d5					
0	1	1	0	0	0	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Cd=Rs
0	1	1	0	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Cdd=Rss
0	1	1	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rdd=Css
0	1	1	0	1	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=Cs

Field name	Description
`sm`	Supervisor mode only
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s

11.5 JR

The JR instruction class includes instructions to change the program flow to a new location contained in a register.

JR instructions are executable on slot 2.

Call subroutine from register

Change the program flow to a subroutine. This instruction first transfers the Next Program Counter (NPC) value into the Link Register, and then jumps to a target address contained in a register.

This instruction can only appear in slot 2.

Syntax	Behavior

`callr Rs`	`LR=NPC;`
	`PC=Rs;`
	`;`

`if ([!]Pu) callr Rs`	`;`
	`if ([!]Pu[0]) {`
	`  LR=NPC;`
	`  PC=Rs;`
	`;`
	`};`

Class: JR (slot 2)

Notes

- This instruction can be conditionally executed based on the value of a predicate register. If the instruction is preceded by 'if Pn', then the instruction only executes if the least-significant bit of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', then the instruction is executed only if the least-significant bit of Pn is 0.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse																	
0	1	0	1	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	callr Rs
ICLASS											s5					Parse						u2										
0	1	0	1	0	0	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	**u**	**u**	-	-	-	-	-	-	-	-	if (Pu) callr Rs
0	1	0	1	0	0	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	**u**	**u**	-	-	-	-	-	-	-	-	if (!Pu) callr Rs

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
u2	Field to encode register u

Hint an indirect jump address

Provide a hint indicating that there will soon be an indirect JUMPR to the address specified in Rs.

Syntax	Behavior
`hintjr(Rs)`	`;`

Class: JR (slot 2)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse																	
0	1	0	1	0	0	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	hintjr(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`s5`	Field to encode register s

Jump to address from register

Change the program flow to a target address. This instruction changes the Program Counter to a target address contained in a register.

This instruction can appear only in slot 2.

Syntax	Behavior
`if ([!]Pu) jumpr Rs`	`Assembler mapped to: "if ([!]Pu) ""jumpr"":nt ""Rs"`

`if ([!]Pu[.new]) jumpr:<hint> Rs`	`};`
	`{`
	`if([!]Pu[.new][0]){`
	`PC=Rs;`
	`;`
	`};`

`jumpr Rs`	`PC=Rs;`

Class: JR (slot 2)

Notes

- This instruction can be conditionally executed based on the value of a predicate register. If the instruction is preceded by 'if Pn', then the instruction only executes if the least-significant bit of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', then the instruction is executed only if the least-significant bit of Pn is 0.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse																	
0	1	0	1	0	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	jumpr Rs
ICLASS											s5					Parse						u2										
0	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	0	-	**u**	**u**	-	-	-	-	-	-	-	-	if (Pu) jumpr:nt Rs
0	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	1	-	**u**	**u**	-	-	-	-	-	-	-	-	if (Pu.new) jumpr:nt Rs
0	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	1	0	-	**u**	**u**	-	-	-	-	-	-	-	-	if (Pu) jumpr:t Rs
0	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	1	1	-	**u**	**u**	-	-	-	-	-	-	-	-	if (Pu.new) jumpr:t Rs
0	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	0	-	**u**	**u**	-	-	-	-	-	-	-	-	if (!Pu) jumpr:nt Rs
0	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	1	-	**u**	**u**	-	-	-	-	-	-	-	-	if (!Pu.new) jumpr:nt Rs
0	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	1	0	-	**u**	**u**	-	-	-	-	-	-	-	-	if (!Pu) jumpr:t Rs
0	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	1	1	-	**u**	**u**	-	-	-	-	-	-	-	-	if (!Pu.new) jumpr:t Rs

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
u2	Field to encode register u

11.6 J

The J instruction class includes branch instructions (jumps and calls) that obtain the target address from a (PC-relative) immediate address value.

J instructions are executable on slot 2 and slot 3.

Call subroutine

Change the program flow to a subroutine. This instruction first transfers the Next Program Counter (NPC) value into the Link Register, and then jumps to the target address.

This instruction can appear in slots 2 or 3.

Syntax	Behavior

`call #r22:2`	`apply_extension(#r);`
	`#r=#r & ~0x3;`
	`LR=NPC;`
	`PC=PC+#r;`
	`;`

`if ([!]Pu) call #r15:2`	`apply_extension(#r);`
	`#r=#r & ~0x3;`
	`;`
	`if ([!]Pu[0]) {`
	`  LR=NPC;`
	`  PC=PC+#r;`
	`;`
	`};`

Class: J (slots 2,3)

Notes

- This instruction can be conditionally executed based on the value of a predicate register. If the instruction is preceded by 'if Pn', then the instruction only executes if the least-significant bit of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', then the instruction is executed only if the least-significant bit of Pn is 0.
- The Next PC value is the address immediately following the last instruction in the packet containing this instruction.
- The PC value is the address of the start of the packet
- A PC-relative address is formed by taking the decoded immediate value and adding it to the current PC value.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse																
0	1	0	1	1	0	1	i	i	i	i	i	i	i	i	i	P	P	i	i	i	i	i	i	i	i	i	i	i	i	i	0	call #r22:2
ICLASS																Parse				D N		u2										
0	1	0	1	1	1	0	1	i	i	0	i	i	i	i	i	P	P	i	-	0	-	u	u	i	i	i	i	i	i	i	-	if (Pu) call #r15:2
0	1	0	1	1	1	0	1	i	i	1	i	i	i	i	i	P	P	i	-	0	-	u	u	i	i	i	i	i	i	i	-	if (!Pu) call #r15:2

Field name	Description
ICLASS	Instruction Class
DN	Dot-new
Parse	Packet/Loop parse bits
u2	Field to encode register u

Compare and jump

Compare two registers, or a register and immediate value, and write a predicate with the result. Then use the predicate result to conditionally jump to a PC-relative target address.

The registers available as operands are restricted to R0-R7 and R16-R23. The predicate destination is restricted to P0 and P1.

In assembly syntax, this instruction appears as two instructions in the packet: a compare and a separate conditional jump. The assembler may convert adjacent compare and jump instructions into compound compare-jump form.

Syntax	Behavior

`p[01]=cmp.eq(Rs,#-1); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs==-1) ? 0xff : 0x00 if ([!]P[01].new[0]) {`
	`  apply_extension(#r);`
	`  #r=#r & ~0x3;`
	`  PC=PC+#r;`
	`};`

`p[01]=cmp.eq(Rs,#U5); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs==#U) ? 0xff : 0x00 if ([!]P[01].new[0]) {`
	`  apply_extension(#r);`
	`  #r=#r & ~0x3;`
	`  PC=PC+#r;`
	`};`

`p[01]=cmp.eq(Rs,Rt); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs==Rt) ? 0xff : 0x00 if ([!]P[01].new[0]) {`
	`  apply_extension(#r);`
	`  #r=#r & ~0x3;`
	`  PC=PC+#r;`
	`};`

`p[01]=cmp.gt(Rs,#-1); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs>-1) ? 0xff : 0x00 if ([!]P[01].new[0]) {`
	`  apply_extension(#r);`
	`  #r=#r & ~0x3;`
	`  PC=PC+#r;`
	`};`

Syntax	Behavior
`p[01]=cmp.gt(Rs,#U5); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs>#U) ? 0xff : 0x00 if ([!]P[01].new[0]) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`
`p[01]=cmp.gt(Rs,Rt); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs>Rt) ? 0xff : 0x00 if ([!]P[01].new[0]) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`
`p[01]=cmp.gtu(Rs,#U5); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs.uw[0]>#U) ? 0xff : 0x00 if ([!]P[01].new[0]) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`
`p[01]=cmp.gtu(Rs,Rt); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs.uw[0]>Rt) ? 0xff : 0x00 if ([!]P[01].new[0]) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`
`p[01]=tstbit(Rs,#0); if ([!]p[01].new) jump:<hint> #r9:2`	`P[01]=(Rs & 1) ? 0xff : 0x00 if ([!]P[01].new[0]) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`

Class: J (slots 2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS												s4				Parse																
0	0	0	1	0	0	0	1	1	0	i	i	s	s	s	s	P	P	0	-	-	-	0	0	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#-1); if (p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	1	0	i	i	s	s	s	s	P	P	0	-	-	-	0	1	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#-1); if (p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	1	0	i	i	s	s	s	s	P	P	0	-	-	-	1	1	i	i	i	i	i	i	i	-	p0=tstbit(Rs,#0); if (p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	1	0	i	i	s	s	s	s	P	P	1	-	-	-	0	0	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#-1); if (p0.new) jump:t #r9:2
0	0	0	1	0	0	0	1	1	0	i	i	s	s	s	s	P	P	1	-	-	-	0	1	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#-1); if (p0.new) jump:t #r9:2
0	0	0	1	0	0	0	1	1	0	i	i	s	s	s	s	P	P	1	-	-	-	1	1	i	i	i	i	i	i	i	-	p0=tstbit(Rs,#0); if (p0.new) jump:t #r9:2
0	0	0	1	0	0	0	1	1	1	i	i	s	s	s	s	P	P	0	-	-	-	0	0	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#-1); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	1	1	i	i	s	s	s	s	P	P	0	-	-	-	0	1	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#-1); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	1	1	i	i	s	s	s	s	P	P	0	-	-	-	1	1	i	i	i	i	i	i	i	-	p0=tstbit(Rs,#0); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	1	1	i	i	s	s	s	s	P	P	1	-	-	-	0	0	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#-1); if (!p0.new) jump:t #r9:2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	0	0	0	1	1	1	i	i	s	s	s	s	P	P	1	-	-	-	0	1	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#-1); if (!p0.new) jump:t #r9:2
0	0	0	1	0	0	0	1	1	1	i	i	s	s	s	s	P	P	1	-	-	-	1	1	i	i	i	i	i	i	i	-	p0=tstbit(Rs,#0); if (!p0.new) jump:t #r9:2
0	0	0	1	0	0	0	0	0	0	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#U5); if (p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	0	0	0	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#U5); if (p0.new) jump:t #r9:2
0	0	0	1	0	0	0	0	0	1	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#U5); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	0	0	1	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.eq(Rs,#U5); if (!p0.new) jump:t #r9:2
0	0	0	1	0	0	0	0	1	0	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#U5); if (p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	0	1	0	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#U5); if (p0.new) jump:t #r9:2
0	0	0	1	0	0	0	0	1	1	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#U5); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	0	1	1	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gt(Rs,#U5); if (!p0.new) jump:t #r9:2
0	0	0	1	0	0	0	1	0	0	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gtu(Rs,#U5); if (p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	0	0	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gtu(Rs,#U5); if (p0.new) jump:t #r9:2
0	0	0	1	0	0	0	1	0	1	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gtu(Rs,#U5); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	0	0	1	0	1	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p0=cmp.gtu(Rs,#U5); if (!p0.new) jump:t #r9:2
0	0	0	1	0	0	1	1	1	0	i	i	s	s	s	s	P	P	0	-	-	-	0	0	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#-1); if (p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	1	0	i	i	s	s	s	s	P	P	0	-	-	-	0	1	i	i	i	i	i	i	i	-	p1=cmp.gt(Rs,#-1); if (p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	1	0	i	i	s	s	s	s	P	P	0	-	-	-	1	1	i	i	i	i	i	i	i	-	p1=tstbit(Rs,#0); if (p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	1	0	i	i	s	s	s	s	P	P	1	-	-	-	0	0	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#-1); if (p1.new) jump:t #r9:2
0	0	0	1	0	0	1	1	1	0	i	i	s	s	s	s	P	P	1	-	-	-	0	1	i	i	i	i	i	i	i	-	p1=cmp.gt(Rs,#-1); if (p1.new) jump:t #r9:2
0	0	0	1	0	0	1	1	1	0	i	i	s	s	s	s	P	P	1	-	-	-	1	1	i	i	i	i	i	i	i	-	p1=tstbit(Rs,#0); if (p1.new) jump:t #r9:2
0	0	0	1	0	0	1	1	1	1	i	i	s	s	s	s	P	P	0	-	-	-	0	0	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#-1); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	1	1	i	i	s	s	s	s	P	P	0	-	-	-	0	1	i	i	i	i	i	i	i	-	p1=cmp.gt(Rs,#-1); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	1	1	i	i	s	s	s	s	P	P	0	-	-	-	1	1	i	i	i	i	i	i	i	-	p1=tstbit(Rs,#0); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	1	1	i	i	s	s	s	s	P	P	1	-	-	-	0	0	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#-1); if (!p1.new) jump:t #r9:2
0	0	0	1	0	0	1	1	1	1	i	i	s	s	s	s	P	P	1	-	-	-	0	1	i	i	i	i	i	i	i	-	p1=cmp.gt(Rs,#-1); if (!p1.new) jump:t #r9:2
0	0	0	1	0	0	1	1	1	1	i	i	s	s	s	s	P	P	1	-	-	-	1	1	i	i	i	i	i	i	i	-	p1=tstbit(Rs,#0); if (!p1.new) jump:t #r9:2
0	0	0	1	0	0	1	0	0	0	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#U5); if (p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	0	0	0	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#U5); if (p1.new) jump:t #r9:2
0	0	0	1	0	0	1	0	0	1	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#U5); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	0	0	1	i	i	s	s	s	s	P	P	1	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.eq(Rs,#U5); if (!p1.new) jump:t #r9:2
0	0	0	1	0	0	1	0	1	0	i	i	s	s	s	s	P	P	0	I	I	I	I	I	i	i	i	i	i	i	i	-	p1=cmp.gt(Rs,#U5); if (p1.new) jump:nt #r9:2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	0	1	0	0	1	0	1	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gt(Rs,#U5); if (p1.new) jump:t #r9:2
0	0	0	1	0	0	1	0	1	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gt(Rs,#U5); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	0	1	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gt(Rs,#U5); if (!p1.new) jump:t #r9:2
0	0	0	1	0	0	1	1	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gtu(Rs,#U5); if (p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gtu(Rs,#U5); if (p1.new) jump:t #r9:2
0	0	0	1	0	0	1	1	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gtu(Rs,#U5); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	0	1	1	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gtu(Rs,#U5); if (!p1.new) jump:t #r9:2
ICLASS												s4				Parse				t4												
0	0	0	1	0	1	0	0	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p0=cmp.eq(Rs,Rt); if (p0.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.eq(Rs,Rt); if (p1.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p0=cmp.eq(Rs,Rt); if (p0.new) jump:t #r9:2
0	0	0	1	0	1	0	0	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.eq(Rs,Rt); if (p1.new) jump:t #r9:2
0	0	0	1	0	1	0	0	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p0=cmp.eq(Rs,Rt); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.eq(Rs,Rt); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p0=cmp.eq(Rs,Rt); if (!p0.new) jump:t #r9:2
0	0	0	1	0	1	0	0	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.eq(Rs,Rt); if (!p1.new) jump:t #r9:2
0	0	0	1	0	1	0	0	1	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p0=cmp.gt(Rs,Rt); if (p0.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	1	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gt(Rs,Rt); if (p1.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	1	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p0=cmp.gt(Rs,Rt); if (p0.new) jump:t #r9:2
0	0	0	1	0	1	0	0	1	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gt(Rs,Rt); if (p1.new) jump:t #r9:2
0	0	0	1	0	1	0	0	1	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p0=cmp.gt(Rs,Rt); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	1	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gt(Rs,Rt); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	1	0	0	1	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p0=cmp.gt(Rs,Rt); if (!p0.new) jump:t #r9:2
0	0	0	1	0	1	0	0	1	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gt(Rs,Rt); if (!p1.new) jump:t #r9:2
0	0	0	1	0	1	0	1	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p0=cmp.gtu(Rs,Rt); if (p0.new) jump:nt #r9:2
0	0	0	1	0	1	0	1	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gtu(Rs,Rt); if (p1.new) jump:nt #r9:2
0	0	0	1	0	1	0	1	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p0=cmp.gtu(Rs,Rt); if (p0.new) jump:t #r9:2
0	0	0	1	0	1	0	1	0	0	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gtu(Rs,Rt); if (p1.new) jump:t #r9:2
0	0	0	1	0	1	0	1	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p0=cmp.gtu(Rs,Rt); if (!p0.new) jump:nt #r9:2
0	0	0	1	0	1	0	1	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gtu(Rs,Rt); if (!p1.new) jump:nt #r9:2
0	0	0	1	0	1	0	1	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	0	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p0=cmp.gtu(Rs,Rt); if (!p0.new) jump:t #r9:2
0	0	0	1	0	1	0	1	0	1	**i**	**i**	**s**	**s**	**s**	**s**	**P**	**P**	1	1	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	-	p1=cmp.gtu(Rs,Rt); if (!p1.new) jump:t #r9:2

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s4	Field to encode register s
t4	Field to encode register t

Jump to address

Change the program flow to a target address. This instruction changes the Program Counter to a target address which is relative to the PC address. The offset from the current PC address is contained in the instruction encoding.

A speculated jump instruction includes a hint ("taken" or "not taken") which specifies the expected value of the conditional expression. If the actual generated value of the predicate differs from this expected value, the jump instruction incurs a performance penalty.

This instruction can appear in slots 2 or 3.

Syntax	Behavior
`if ([!]Pu) jump #r15:2`	`Assembler mapped to: "if ([!]Pu) ""jump"":nt ""#r15:2"`
`if ([!]Pu) jump:<hint> #r15:2`	`; if ([!]Pu[0]) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; ; };`
`jump #r22:2`	`apply_extension(#r); #r=#r & ~0x3; PC=PC+#r;`

Class: J (slots 2,3)

Notes

- This instruction can be conditionally executed based on the value of a predicate register. If the instruction is preceded by 'if Pn', then the instruction only executes if the least-significant bit of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', then the instruction is executed only if the least-significant bit of Pn is 0.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse																
0	1	0	1	1	0	0	i	i	i	i	i	i	i	i	i	P	P	i	i	i	i	i	i	i	i	i	i	i	i	i	-	jump #r22:2
ICLASS																Parse			PT	D N		u2										
0	1	0	1	1	1	0	0	i	i	0	i	i	i	i	i	P	P	i	0	0	-	u	u	i	i	i	i	i	i	i	-	if (Pu) jump:nt #r15:2
0	1	0	1	1	1	0	0	i	i	0	i	i	i	i	i	P	P	i	1	0	-	u	u	i	i	i	i	i	i	i	-	if (Pu) jump:t #r15:2
0	1	0	1	1	1	0	0	i	i	1	i	i	i	i	i	P	P	i	0	0	-	u	u	i	i	i	i	i	i	i	-	if (!Pu) jump:nt #r15:2
0	1	0	1	1	1	0	0	i	i	1	i	i	i	i	i	P	P	i	1	0	-	u	u	i	i	i	i	i	i	i	-	if (!Pu) jump:t #r15:2

Field name	Description
ICLASS	Instruction Class
DN	Dot-new
PT	Predict-taken
Parse	Packet/Loop parse bits
u2	Field to encode register u

Jump to address conditioned on new predicate

Perform speculated jump.

Jump if the LSB of the newly-generated predicate is true. The predicate must be generated in the same packet as the speculated jump instruction.

A speculated jump instruction includes a hint ("taken" or "not taken") which specifies the expected value of the conditional expression. If the actual generated value of the predicate differs from this expected value, the jump instruction incurs a performance penalty.

This instruction can appear in slots 2 or 3.

Syntax	Behavior
`if ([!]Pu.new) jump:<hint> #r15:2`	`}; { if([!]Pu.new[0]){ apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; ; };`

Class: J (slots 2,3)

Notes

- This instruction can be conditionally executed based on the value of a predicate register. If the instruction is preceded by 'if Pn', then the instruction only executes if the least-significant bit of the predicate register is 1. Similarly, if the instruction is preceded by 'if !Pn', then the instruction is executed only if the least-significant bit of Pn is 0.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse			PT	D N		u2										
0	1	0	1	1	1	0	0	i	i	0	i	i	i	i	i	P	P	i	0	1	-	u	u	i	i	i	i	i	i	i	-	if (Pu.new) jump:nt #r15:2
0	1	0	1	1	1	0	0	i	i	0	i	i	i	i	i	P	P	i	1	1	-	u	u	i	i	i	i	i	i	i	-	if (Pu.new) jump:t #r15:2
0	1	0	1	1	1	0	0	i	i	1	i	i	i	i	i	P	P	i	0	1	-	u	u	i	i	i	i	i	i	i	-	if (!Pu.new) jump:nt #r15:2
0	1	0	1	1	1	0	0	i	i	1	i	i	i	i	i	P	P	i	1	1	-	u	u	i	i	i	i	i	i	i	-	if (!Pu.new) jump:t #r15:2

Field name	Description
ICLASS	Instruction Class
DN	Dot-new
PT	Predict-taken
Parse	Packet/Loop parse bits
u2	Field to encode register u

Jump to address condition on register value

Perform register-conditional jump.

Jump if the specified register expression is true.

A register-conditional jump includes a hint ("taken" or "not taken") which specifies the expected value of the register expression. If the actual generated value of the expression differs from this expected value, the jump instruction incurs a performance penalty.

This instruction can appear only in slot 3.

Syntax	Behavior

`if (Rs!=#0) jump:nt #r13:2`	`if (Rs != 0) {`
	`  PC=PC+#r;`
	`};`

`if (Rs!=#0) jump:t #r13:2`	`if (Rs != 0) {`
	`  PC=PC+#r;`
	`};`

`if (Rs<=#0) jump:nt #r13:2`	`if (Rs<=0) {`
	`  PC=PC+#r;`
	`};`

`if (Rs<=#0) jump:t #r13:2`	`if (Rs<=0) {`
	`  PC=PC+#r;`
	`};`

`if (Rs==#0) jump:nt #r13:2`	`if (Rs == 0) {`
	`  PC=PC+#r;`
	`};`

`if (Rs==#0) jump:t #r13:2`	`if (Rs == 0) {`
	`  PC=PC+#r;`
	`};`

`if (Rs>=#0) jump:nt #r13:2`	`if (Rs>=0) {`
	`  PC=PC+#r;`
	`};`

`if (Rs>=#0) jump:t #r13:2`	`if (Rs>=0) {`
	`  PC=PC+#r;`
	`};`

Class: J (slot 3)

Notes

- This instruction will be deprecated in a future version.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm						s5					Parse																	
0	1	1	0	0	0	0	1	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs!=#0) jump:nt #r13:2
0	1	1	0	0	0	0	1	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs!=#0) jump:t #r13:2
0	1	1	0	0	0	0	1	0	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs>=#0) jump:nt #r13:2
0	1	1	0	0	0	0	1	0	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs>=#0) jump:t #r13:2
0	1	1	0	0	0	0	1	1	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs==#0) jump:nt #r13:2
0	1	1	0	0	0	0	1	1	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs==#0) jump:t #r13:2
0	1	1	0	0	0	0	1	1	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs<=#0) jump:nt #r13:2
0	1	1	0	0	0	0	1	1	1	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (Rs<=#0) jump:t #r13:2

Field name	Description
sm	Supervisor mode only
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s

Transfer and jump

Move an unsigned immediate or register value into a destination register and unconditionally jump. In assembly syntax, this instruction appears as two instructions in the packet, a transfer and a separate jump. The assembler may convert adjacent transfer and jump instructions into compound transfer-jump form.

Syntax	Behavior

`Rd=#U6 ; jump #r9:2`	`apply_extension(#r);`
	`#r=#r & ~0x3;`
	`Rd=#U;`
	`PC=PC+#r;`

`Rd=Rs ; jump #r9:2`	`apply_extension(#r);`
	`#r=#r & ~0x3;`
	`Rd=Rs;`
	`PC=PC+#r;`

Class: J (slots 2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS												d4				Parse																
0	0	0	1	0	1	1	0	-	-	i	i	d	d	d	d	P	P	I	I	I	I	I	I	i	i	i	i	i	i	i	-	Rd=#U6 ; jump #r9:2
ICLASS												s4				Parse				d4												
0	0	0	1	0	1	1	1	-	-	i	i	s	s	s	s	P	P	-	-	d	d	d	d	i	i	i	i	i	i	i	-	Rd=Rs ; jump #r9:2

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d4	Field to encode register d
s4	Field to encode register s

11.7 LD

The LD instruction class includes load instructions, which are used to load values into registers.

LD instructions are executable on slot 0 and slot 1.

Load doubleword

Load a 64-bit doubleword from memory and place in a destination register pair.

Syntax	Behavior

`Rdd=memd(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`Rdd = *EA;`
	`Re=#U;`

`Rdd=memd(Rs+#s11:3)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`Rdd = *EA;`

`Rdd=memd(Rs+Rt<<#u2)`	`EA=Rs+(Rt<<#u);` `Rdd = *EA;`

`Rdd=memd(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`Rdd = *EA;`

`Rdd=memd(Rx++#s4:3)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`Rdd = *EA;`

`Rdd=memd(Rx++#s4:3:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`Rdd = *EA;`

`Rdd=memd(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<3,MuV);`
	`Rdd = *EA;`

`Rdd=memd(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`Rdd = *EA;`

`Rdd=memd(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`Rdd = *EA;`

`Rdd=memd(gp+#u16:3)`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`Rdd = *EA;`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	1	0	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rs+Rt<<#u2)
ICLASS								Type		U N						Parse											d5					
0	1	0	0	1	**i**	**i**	1	1	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(gp+#u16:3)
ICLASS				Amode			Type			U N	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rs+#s11:3)
ICLASS				Amode			Type			U N	x5					Parse		u1									d5					

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	1	1	0	0	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rx++#s4:3:circ(Mu))
1	0	0	1	1	0	0	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse											d5					
1	0	0	1	1	0	1	1	1	1	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											d5					
1	0	0	1	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rx++#s4:3)
ICLASS				Amode			Type			UN	t5					Parse											d5					
1	0	0	1	1	1	0	1	1	1	0	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	1	0	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rx++Mu)
1	0	0	1	1	1	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=memd(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x

Load doubleword conditionally

Load a 64-bit doubleword from memory and place in a destination register pair.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pt[.new]) Rdd=memd(#u6)`	`if ([!]Pt[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=#u;`
	`  Rdd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pt[.new])`
`Rdd=memd(Rs+#u6:3)`
	`if ([!]Pt[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=Rs+#u;`
	`  Rdd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pt[.new])`
`Rdd=memd(Rx++#s4:3)`
	`if([!]Pt[.new][0]){`
	`  EA=Rx;`
	`  Rx=Rx+#s;`
	`  Rdd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`Rdd=memd(Rs+Rt<<#u2)`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+(Rt<<#u);`
	`  Rdd = *EA;`
	`} else {`
	`  NOP;`
	`};`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (Pv) Rdd=memd(Rs+Rt<<#u2)
0	0	1	1	0	0	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (!Pv) Rdd=memd(Rs+Rt<<#u2)
0	0	1	1	0	0	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (Pv.new) Rdd=memd(Rs+Rt<<#u2)
0	0	1	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (!Pv.new) Rdd=memd(Rs+Rt<<#u2)
ICLASS					Sense	Pred New		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rdd=memd(Rs+#u6:3)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rdd=memd(Rs+#u6:3)
0	1	0	0	0	1	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rdd=memd(Rs+#u6:3)
0	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rdd=memd(Rs+#u6:3)
ICLASS				Amode			Type			U N	x5					Parse					t2						d5						
1	0	0	1	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rdd=memd(Rx++#s4:3)
1	0	0	1	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rdd=memd(Rx++#s4:3)
1	0	0	1	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rdd=memd(Rx++#s4:3)
1	0	0	1	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rdd=memd(Rx++#s4:3)
ICLASS				Amode			Type			U N						Parse					t2						d5						
1	0	0	1	1	1	1	1	1	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rdd=memd(#u6)
1	0	0	1	1	1	1	1	1	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rdd=memd(#u6)
1	0	0	1	1	1	1	1	1	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rdd=memd(#u6)
1	0	0	1	1	1	1	1	1	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rdd=memd(#u6)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x

Load byte

Load a signed byte from memory. The byte at the effective address in memory is placed in the least-significant 8 bits of the destination register. The destination register is then sign-extended from 8 bits to 32.

Syntax	Behavior

`Rd=memb(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`Rd = *EA;`
	`Re=#U;`

`Rd=memb(Rs+#s11:0)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`Rd = *EA;`

`Rd=memb(Rs+Rt<<#u2)`	`EA=Rs+(Rt<<#u);` `Rd = *EA;`

`Rd=memb(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`Rd = *EA;`

`Rd=memb(Rx++#s4:0)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`Rd = *EA;`

`Rd=memb(Rx++#s4:0:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`Rd = *EA;`

`Rd=memb(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<0,MuV);`
	`Rd = *EA;`

`Rd=memb(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memb(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memb(gp+#u16:0)`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`Rd = *EA;`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	1	0	1	0	0	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	-	-	d	d	d	d	d	Rd=memb(Rs+Rt<<#u2)
ICLASS								Type		U N						Parse											d5					
0	1	0	0	1	i	i	1	0	0	0	i	i	i	i	i	P	P	i	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=memb(gp+#u16:0)
ICLASS				Amode			Type			U N	s5					Parse											d5					
1	0	0	1	0	i	i	1	0	0	0	s	s	s	s	s	P	P	i	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=memb(Rs+#s11:0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			U N	x5					Parse		u1										d5					
1	0	0	1	1	0	0	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++#s4:0:circ(Mu))
1	0	0	1	1	0	0	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++I:circ(Mu))
ICLASS				Amode			Type			U N	e5					Parse											d5					
1	0	0	1	1	0	1	1	0	0	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Re=#U6)
ICLASS				Amode			Type			U N	x5					Parse											d5					
1	0	0	1	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++#s4:0)
ICLASS				Amode			Type			U N	t5					Parse											d5					
1	0	0	1	1	1	0	1	0	0	0	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rt<<#u2+#U6)
ICLASS				Amode			Type			U N	x5					Parse		u1									d5					
1	0	0	1	1	1	0	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++Mu)
1	0	0	1	1	1	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memb(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x

Load byte conditionally

Load a signed byte from memory. The byte at the effective address in memory is placed in the least-significant 8 bits of the destination register. The destination register is then sign-extended from 8 bits to 32.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pt[.new]) Rd=memb(#u6)`	`if ([!]Pt[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=#u;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pt[.new])`
`Rd=memb(Rs+#u6:0)`
	`if ([!]Pt[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=Rs+#u;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pt[.new])`
`Rd=memb(Rx++#s4:0)`
	`if([!]Pt[.new][0]){`
	`  EA=Rx;`
	`  Rx=Rx+#s;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`Rd=memb(Rs+Rt<<#u2)`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+(Rt<<#u);`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

Class: LD (slots 0,1)

Encoding

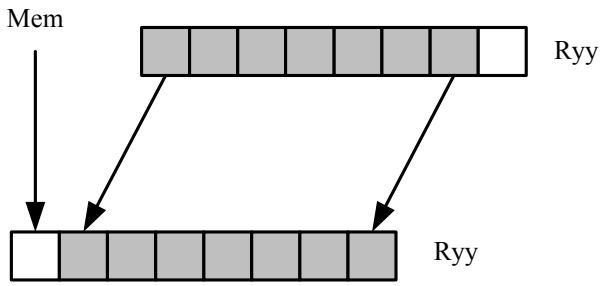
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5									d5					
0	0	1	1	0	0	0	0	0	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (Pv) Rd=memb(Rs+Rt<<#u2)
0	0	1	1	0	0	0	1	0	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (!Pv) Rd=memb(Rs+Rt<<#u2)
0	0	1	1	0	0	1	0	0	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (Pv.new) Rd=memb(Rs+Rt<<#u2)
0	0	1	1	0	0	1	1	0	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (!Pv.new) Rd=memb(Rs+Rt<<#u2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memb(Rs+#u6:0)
0	1	0	0	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memb(Rs+#u6:0)
0	1	0	0	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memb(Rs+#u6:0)
0	1	0	0	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memb(Rs+#u6:0)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memb(Rx++#s4:0)
1	0	0	1	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memb(Rx++#s4:0)
1	0	0	1	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memb(Rx++#s4:0)
1	0	0	1	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memb(Rx++#s4:0)
ICLASS				Amode			Type			UN						Parse					t2						d5					
1	0	0	1	1	1	1	1	0	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memb(#u6)
1	0	0	1	1	1	1	1	0	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memb(#u6)
1	0	0	1	1	1	1	1	0	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memb(#u6)
1	0	0	1	1	1	1	1	0	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memb(#u6)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x

Load byte into shifted vector

Shift a 64-bit vector right by one byte. Insert a byte from memory into the vacated upper byte of the vector.



Syntax	Behavior

`Ryy=memb_fifo(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`};`
	`;`
	`Re=#U;`

`Ryy=memb_fifo(Rs)`	`Assembler mapped to: "Ryy=memb_fifo""(Rs+#0)"`

`Ryy=memb_fifo(Rs+#s11:0)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`};`
	`;`

`Ryy=memb_fifo(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`};`
	`;`

Syntax	Behavior

`Ryy=memb_fifo(Rx++#s4:0)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`};`
	`;`

`Ryy=memb_fifo(Rx++#s4:0:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`};`
	`;`

`Ryy=memb_fifo(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<0,MuV);`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`};`
	`;`

`Ryy=memb_fifo(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`};`
	`;`

`Ryy=memb_fifo(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>8)|(tmpV<<56);`
	`};`
	`;`

Class: LD (slots 0,1)

Encoding

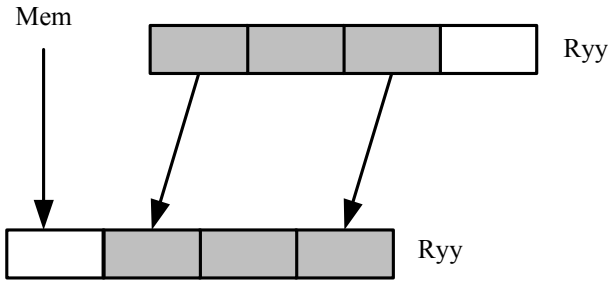
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse												y5					
1	0	0	1	0	**i**	**i**	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Rs+#s11:0)
ICLASS				Amode			Type			UN	x5					Parse	u1										y5					
1	0	0	1	1	0	0	0	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Rx++#s4:0:circ(Mu))
1	0	0	1	1	0	0	0	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse											y5					
1	0	0	1	1	0	1	0	1	0	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											y5					

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	1	1	0	1	0	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Rx++#s4:0)
ICLASS				Amode			Type			UN	t5					Parse												y5					
1	0	0	1	1	1	0	0	1	0	0	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**l**	**l**	**l**	**l**	**i**	**l**	**l**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1										y5					
1	0	0	1	1	1	0	0	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Rx++Mu)
1	0	0	1	1	1	1	0	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**y**	**y**	**y**	**y**	**y**	Ryy=memb_fifo(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x
y5	Field to encode register y

Load half into shifted vector

Shift a 64-bit vector right by one halfword. Insert a halfword from memory into the vacated upper halfword of the vector.



Syntax	Behavior

`Ryy=memh_fifo(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`};`
	`;`
	`Re=#U;`

`Ryy=memh_fifo(Rs)`	`Assembler mapped to: "Ryy=memh_fifo""(Rs+#0)"`

`Ryy=memh_fifo(Rs+#s11:1)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`};`
	`;`

`Ryy=memh_fifo(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`};`
	`;`

Syntax	Behavior

`Ryy=memh_fifo(Rx++#s4:1)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`};`
	`;`

`Ryy=memh_fifo(Rx++#s4:1:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`};`
	`;`

`Ryy=memh_fifo(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`};`
	`;`

`Ryy=memh_fifo(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`};`
	`;`

`Ryy=memh_fifo(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  Ryy = (((size8u_t)Ryy)>>16)|(tmpV<<48);`
	`};`
	`;`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse											y5					
1	0	0	1	0	**i**	**i**	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Rs+#s11:1)
ICLASS				Amode			Type			UN	x5					Parse		u1									y5					
1	0	0	1	1	0	0	0	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Rx++#s4:1:circ(Mu))
1	0	0	1	1	0	0	0	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse											y5					
1	0	0	1	1	0	1	0	0	1	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											y5					

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	1	1	0	1	0	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Rx++#s4:1)
ICLASS				Amode			Type			UN	t5					Parse											y5						
1	0	0	1	1	1	0	0	0	1	0	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									y5						
1	0	0	1	1	1	0	0	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Rx++Mu)
1	0	0	1	1	1	1	0	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**y**	**y**	**y**	**y**	**y**	Ryy=memh_fifo(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x
y5	Field to encode register y

Load halfword

Load a signed halfword from memory. The 16-bit halfword at the effective address in memory is placed in the least-significant 16 bits of the destination register. The destination register is then sign-extended from 16 bits to 32.

Syntax	Behavior

`Rd=memh(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`Rd = *EA;`
	`Re=#U;`

`Rd=memh(Rs+#s11:1)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`Rd = *EA;`

`Rd=memh(Rs+Rt<<#u2)`	`EA=Rs+(Rt<<#u);` `Rd = *EA;`

`Rd=memh(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`Rd = *EA;`

`Rd=memh(Rx++#s4:1)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`Rd = *EA;`

`Rd=memh(Rx++#s4:1:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`Rd = *EA;`

`Rd=memh(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`Rd = *EA;`

`Rd=memh(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memh(gp+#u16:1)`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`Rd = *EA;`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	1	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rs+Rt<<#u2)
ICLASS								Type		U N						Parse											d5					
0	1	0	0	1	**i**	**i**	1	0	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memh(gp+#u16:1)
ICLASS				Amode			Type			U N	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rs+#s11:1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	0	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++#s4:1:circ(Mu))
1	0	0	1	1	0	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse											d5					
1	0	0	1	1	0	1	1	0	1	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											d5					
1	0	0	1	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++#s4:1)
ICLASS				Amode			Type			UN	t5					Parse											d5					
1	0	0	1	1	1	0	1	0	1	0	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	1	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++Mu)
1	0	0	1	1	1	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memh(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x

Load halfword conditionally

Load a signed halfword from memory. The 16-bit halfword at the effective address in memory is placed in the least-significant 16 bits of the destination register. The destination register is then sign-extended from 16 bits to 32.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pt[.new]) Rd=memh(#u6)`	`if ([!]Pt[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=#u;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pt[.new])`
`Rd=memh(Rs+#u6:1)`
	`if ([!]Pt[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=Rs+#u;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pt[.new])`
`Rd=memh(Rx++#s4:1)`
	`if([!]Pt[.new][0]){`
	`  EA=Rx;`
	`  Rx=Rx+#s;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`Rd=memh(Rs+Rt<<#u2)`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+(Rt<<#u);`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5									d5					
0	0	1	1	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (Pv) Rd=memh(Rs+Rt<<#u2)
0	0	1	1	0	0	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (!Pv) Rd=memh(Rs+Rt<<#u2)
0	0	1	1	0	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (Pv.new) Rd=memh(Rs+Rt<<#u2)
0	0	1	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (!Pv.new) Rd=memh(Rs+Rt<<#u2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memh(Rs+#u6:1)
0	1	0	0	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memh(Rs+#u6:1)
0	1	0	0	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memh(Rs+#u6:1)
0	1	0	0	0	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memh(Rs+#u6:1)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memh(Rx++#s4:1)
1	0	0	1	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memh(Rx++#s4:1)
1	0	0	1	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memh(Rx++#s4:1)
1	0	0	1	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memh(Rx++#s4:1)
ICLASS				Amode			Type			UN						Parse					t2						d5					
1	0	0	1	1	1	1	1	0	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**i**	1	-	-	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memh(#u6)
1	0	0	1	1	1	1	1	0	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**i**	1	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memh(#u6)
1	0	0	1	1	1	1	1	0	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**i**	1	-	-	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memh(#u6)
1	0	0	1	1	1	1	1	0	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	1	**t**	**t**	**i**	1	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memh(#u6)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x

Load unsigned byte

Load an unsigned byte from memory. The byte at the effective address in memory is placed in the least-significant 8 bits of the destination register. The destination register is then zero-extended from 8 bits to 32.

Syntax	Behavior

`Rd=memub(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`Rd = *EA;`
	`Re=#U;`

`Rd=memub(Rs+#s11:0)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`Rd = *EA;`

`Rd=memub(Rs+Rt<<#u2)`	`EA=Rs+(Rt<<#u);` `Rd = *EA;`

`Rd=memub(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`Rd = *EA;`

`Rd=memub(Rx++#s4:0)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`Rd = *EA;`

`Rd=memub(Rx++#s4:0:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`Rd = *EA;`

`Rd=memub(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<0,MuV);`
	`Rd = *EA;`

`Rd=memub(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memub(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memub(gp+#u16:0)`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`Rd = *EA;`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5									d5					
0	0	1	1	1	0	1	0	0	0	1	s	s	s	s	s	P	P	i	t	t	t	t	t	i	-	-	d	d	d	d	d	Rd=memub(Rs+Rt<<#u2)
ICLASS								Type		U N						Parse												d5					
0	1	0	0	1	i	i	1	0	0	1	i	i	i	i	i	P	P	i	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=memub(gp+#u16:0)
ICLASS				Amode			Type			U N	s5					Parse												d5					
1	0	0	1	0	i	i	1	0	0	1	s	s	s	s	s	P	P	i	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=memub(Rs+#s11:0)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	0	0	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++#s4:0:circ(Mu))
1	0	0	1	1	0	0	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse											d5					
1	0	0	1	1	0	1	1	0	0	1	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											d5					
1	0	0	1	1	0	1	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++#s4:0)
ICLASS				Amode			Type			UN	t5					Parse											d5					
1	0	0	1	1	1	0	1	0	0	1	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	1	0	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++Mu)
1	0	0	1	1	1	1	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memub(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x

Load unsigned byte conditionally

Load an unsigned byte from memory. The byte at the effective address in memory is placed in the least-significant 8 bits of the destination register. The destination register is then zero-extended from 8 bits to 32.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pt[.new]) Rd=memub(#u6)`	`if ([!]Pt[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=#u;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pt[.new])`
`Rd=memub(Rs+#u6:0)`
	`if ([!]Pt[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=Rs+#u;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pt[.new])`
`Rd=memub(Rx++#s4:0)`
	`if([!]Pt[.new][0]){`
	`  EA=Rx;`
	`  Rx=Rx+#s;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`Rd=memub(Rs+Rt<<#u2)`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+(Rt<<#u);`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5									d5					
0	0	1	1	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (Pv) Rd=memub(Rs+Rt<<#u2)
0	0	1	1	0	0	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (!Pv) Rd=memub(Rs+Rt<<#u2)
0	0	1	1	0	0	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (Pv.new) Rd=memub(Rs+Rt<<#u2)
0	0	1	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (!Pv.new) Rd=memub(Rs+Rt<<#u2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memub(Rs+#u6:0)
0	1	0	0	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memub(Rs+#u6:0)
0	1	0	0	0	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memub(Rs+#u6:0)
0	1	0	0	0	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memub(Rs+#u6:0)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memub(Rx++#s4:0)
1	0	0	1	1	0	1	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memub(Rx++#s4:0)
1	0	0	1	1	0	1	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memub(Rx++#s4:0)
1	0	0	1	1	0	1	1	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memub(Rx++#s4:0)
ICLASS				Amode			Type			UN						Parse					t2						d5					
1	0	0	1	1	1	1	1	0	0	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memub(#u6)
1	0	0	1	1	1	1	1	0	0	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memub(#u6)
1	0	0	1	1	1	1	1	0	0	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memub(#u6)
1	0	0	1	1	1	1	1	0	0	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memub(#u6)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x

Load unsigned halfword

Load an unsigned halfword from memory. The 16-bit halfword at the effective address in memory is placed in the least-significant 16 bits of the destination register. The destination register is zero-extended from 16 bits to 32.

Syntax	Behavior

`Rd=memuh(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`Rd = *EA;`
	`Re=#U;`

`Rd=memuh(Rs+#s11:1)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`Rd = *EA;`

`Rd=memuh(Rs+Rt<<#u2)`	`EA=Rs+(Rt<<#u);` `Rd = *EA;`

`Rd=memuh(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`Rd = *EA;`

`Rd=memuh(Rx++#s4:1)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`Rd = *EA;`

`Rd=memuh(Rx++#s4:1:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`Rd = *EA;`

`Rd=memuh(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`Rd = *EA;`

`Rd=memuh(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memuh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memuh(gp+#u16:1)`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`Rd = *EA;`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5									d5					
0	0	1	1	1	0	1	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rs+Rt<<#u2)
ICLASS								Type		U N						Parse											d5					
0	1	0	0	1	**i**	**i**	1	0	1	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(gp+#u16:1)
ICLASS				Amode			Type			U N	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rs+#s11:1)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	x5					Parse		u1										d5					
1	0	0	1	1	0	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++#s4:1:circ(Mu))
1	0	0	1	1	0	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse											d5					
1	0	0	1	1	0	1	1	0	1	1	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											d5					
1	0	0	1	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++#s4:1)
ICLASS				Amode			Type			UN	t5					Parse											d5					
1	0	0	1	1	1	0	1	0	1	1	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	1	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++Mu)
1	0	0	1	1	1	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memuh(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x

Load unsigned halfword conditionally

Load an unsigned halfword from memory. The 16-bit halfword at the effective address in memory is placed in the least-significant 16 bits of the destination register. The destination register is zero-extended from 16 bits to 32.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pt[.new]) Rd=memuh(#u6)`	`if ([!]Pt[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=#u;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pt[.new])`
`Rd=memuh(Rs+#u6:1)`
	`if ([!]Pt[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=Rs+#u;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pt[.new])`
`Rd=memuh(Rx++#s4:1)`
	`if([!]Pt[.new][0]){`
	`  EA=Rx;`
	`  Rx=Rx+#s;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`Rd=memuh(Rs+Rt<<#u2)`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+(Rt<<#u);`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5									d5					
0	0	1	1	0	0	0	0	0	1	1	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (Pv) Rd=memuh(Rs+Rt<<#u2)
0	0	1	1	0	0	0	1	0	1	1	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (!Pv) Rd=memuh(Rs+Rt<<#u2)
0	0	1	1	0	0	1	0	0	1	1	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (Pv.new) Rd=memuh(Rs+Rt<<#u2)
0	0	1	1	0	0	1	1	0	1	1	s	s	s	s	s	P	P	i	t	t	t	t	t	i	v	v	d	d	d	d	d	if (!Pv.new) Rd=memuh(Rs+Rt<<#u2)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memuh(Rs+#u6:1)
0	1	0	0	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memuh(Rs+#u6:1)
0	1	0	0	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memuh(Rs+#u6:1)
0	1	0	0	0	1	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memuh(Rs+#u6:1)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5					
1	0	0	1	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memuh(Rx++#s4:1)
1	0	0	1	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memuh(Rx++#s4:1)
1	0	0	1	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memuh(Rx++#s4:1)
1	0	0	1	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memuh(Rx++#s4:1)
ICLASS				Amode			Type			UN						Parse					t2						d5					
1	0	0	1	1	1	1	1	0	1	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memuh(#u6)
1	0	0	1	1	1	1	1	0	1	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memuh(#u6)
1	0	0	1	1	1	1	1	0	1	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memuh(#u6)
1	0	0	1	1	1	1	1	0	1	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	1	**t**	**t**	**i**	1	**-**	**-**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memuh(#u6)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x

Load word

Load a 32-bit word from memory and place in a destination register.

Syntax	Behavior

`Rd=memw(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`Rd = *EA;`
	`Re=#U;`

`Rd=memw(Rs+#s11:2)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`Rd = *EA;`

`Rd=memw(Rs+Rt<<#u2)`	`EA=Rs+(Rt<<#u);` `Rd = *EA;`

`Rd=memw(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`Rd = *EA;`

`Rd=memw(Rx++#s4:2)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`Rd = *EA;`

`Rd=memw(Rx++#s4:2:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`Rd = *EA;`

`Rd=memw(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<2,MuV);`
	`Rd = *EA;`

`Rd=memw(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memw(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`Rd = *EA;`

`Rd=memw(gp+#u16:2)`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`Rd = *EA;`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5					
0	0	1	1	1	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rs+Rt<<#u2)
ICLASS								Type		U N						Parse											d5					
0	1	0	0	1	**i**	**i**	1	1	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memw(gp+#u16:2)
ICLASS				Amode			Type			U N	s5					Parse											d5					
1	0	0	1	0	**i**	**i**	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rs+#s11:2)
ICLASS				Amode			Type			U N	x5					Parse		u1									d5					

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	1	1	0	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	0	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++#s4:2:circ(Mu))
1	0	0	1	1	0	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	1	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse											d5					
1	0	0	1	1	0	1	1	1	0	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	1	**I**	**I**	**I**	**I**	**-**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											d5					
1	0	0	1	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	0	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++#s4:2)
ICLASS				Amode			Type			UN	t5					Parse											d5					
1	0	0	1	1	1	0	1	1	0	0	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**I**	**I**	**I**	**I**	**i**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	1	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++Mu)
1	0	0	1	1	1	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	**-**	**-**	**-**	**-**	0	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rd=memw(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x

Load word conditionally

Load a 32-bit word from memory and place in a destination register.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pt[.new]) Rd=memw(#u6)`	`if ([!]Pt[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=#u;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pt[.new])`
`Rd=memw(Rs+#u6:2)`
	`if ([!]Pt[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=Rs+#u;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pt[.new])`
`Rd=memw(Rx++#s4:2)`
	`if([!]Pt[.new][0]){`
	`  EA=Rx;`
	`  Rx=Rx+#s;`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`Rd=memw(Rs+Rt<<#u2)`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+(Rt<<#u);`
	`  Rd = *EA;`
	`} else {`
	`  NOP;`
	`};`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			t5								d5						
0	0	1	1	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (Pv) Rd=memw(Rs+Rt<<#u2)
0	0	1	1	0	0	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (!Pv) Rd=memw(Rs+Rt<<#u2)
0	0	1	1	0	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (Pv.new) Rd=memw(Rs+Rt<<#u2)
0	0	1	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**v**	**v**	**d**	**d**	**d**	**d**	**d**	if (!Pv.new) Rd=memw(Rs+Rt<<#u2)
ICLASS					Sense	PredNew		Type		UN	s5					Parse			t2								d5					
0	1	0	0	0	0	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memw(Rs+#u6:2)

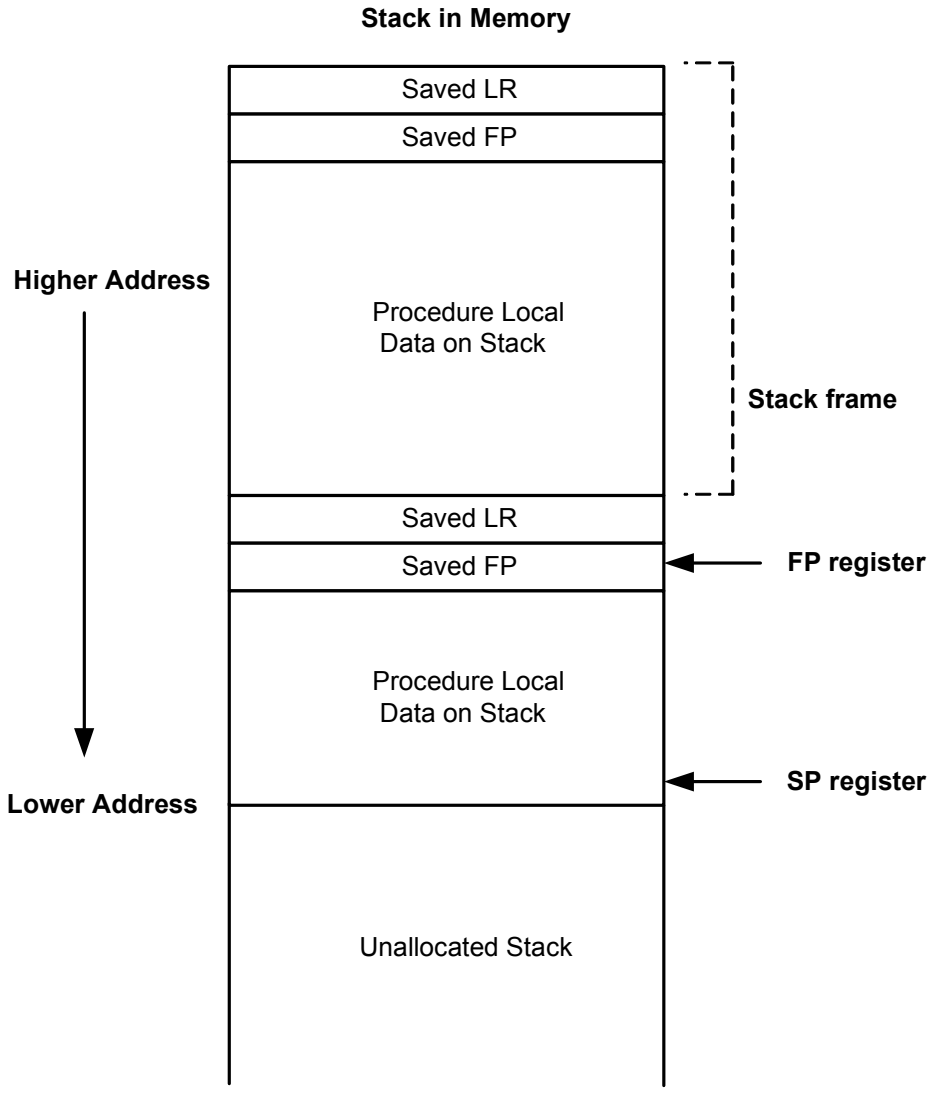
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memw(Rs+#u6:2)
0	1	0	0	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memw(Rs+#u6:2)
0	1	0	0	0	1	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memw(Rs+#u6:2)
ICLASS				Amode			Type			UN	x5					Parse					t2						d5						
1	0	0	1	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memw(Rx++#s4:2)
1	0	0	1	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memw(Rx++#s4:2)
1	0	0	1	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memw(Rx++#s4:2)
1	0	0	1	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	1	**t**	**t**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memw(Rx++#s4:2)
ICLASS				Amode			Type			UN						Parse					t2						d5						
1	0	0	1	1	1	1	1	1	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**i**	1	-	-	**d**	**d**	**d**	**d**	**d**	if (Pt) Rd=memw(#u6)
1	0	0	1	1	1	1	1	1	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**i**	1	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pt) Rd=memw(#u6)
1	0	0	1	1	1	1	1	1	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**i**	1	-	-	**d**	**d**	**d**	**d**	**d**	if (Pt.new) Rd=memw(#u6)
1	0	0	1	1	1	1	1	1	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	1	1	1	**t**	**t**	**i**	1	-	-	**d**	**d**	**d**	**d**	**d**	if (!Pt.new) Rd=memw(#u6)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Type	Type
UN	Unsigned
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t2	Field to encode register t
t5	Field to encode register t
v2	Field to encode register v
x5	Field to encode register x

Deallocate stack frame

Deallocate a stack frame from the call stack. The instruction first loads the saved FP and saved LR values from the address at FP. It then points SP back to the previous frame.

The stack layout is seen in the following figure.



Syntax	Behavior

`deallocframe`	`EA=FP;`
	`tmp = *EA;`
	`tmp = frame_unscramble(tmp);`
	`LR=tmp.w[1];`
	`FP=tmp.w[0];`
	`SP=EA+8;`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN						Parse																	
1	0	0	1	0	0	0	0	0	0	0	1	1	1	1	0	**P**	**P**	0	-	-	-	-	-	-	-	-	1	1	1	1	0	deallocframe

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits

Deallocate frame and return

Return from a function with a stack frame. This instruction is equivalent to deallocframe followed by jumpr R31.

Syntax	Behavior

`dealloc_return`	`EA=FP;`
	`tmp = *EA;`
	`tmp = frame_unscramble(tmp);`
	`LR=tmp.w[1];`
	`FP=tmp.w[0];`
	`SP=EA+8;`
	`PC=tmp.w[1];`

`if ([!]Ps) dealloc_return`	`;`
	`;`
	`if ([!]Ps[0]) {`
	`  EA=FP;`
	`  tmp = *EA;`
	`  tmp = frame_unscramble(tmp);`
	`  LR=tmp.w[1];`
	`  FP=tmp.w[0];`
	`  SP=EA+8;`
	`  PC=tmp.w[1];`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Ps.new) dealloc_return:nt`	`;`
	`if ([!]Ps.new[0]) {`
	`  EA=FP;`
	`  tmp = *EA;`
	`  tmp = frame_unscramble(tmp);`
	`  LR=tmp.w[1];`
	`  FP=tmp.w[0];`
	`  SP=EA+8;`
	`  PC=tmp.w[1];`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Ps.new) dealloc_return:t`	`;`
	`if ([!]Ps.new[0]) {`
	`  EA=FP;`
	`  tmp = *EA;`
	`  tmp = frame_unscramble(tmp);`
	`  LR=tmp.w[1];`
	`  FP=tmp.w[0];`
	`  SP=EA+8;`
	`  PC=tmp.w[1];`
	`} else {`
	`  NOP;`
	`};`

Class: LD (slots 0)

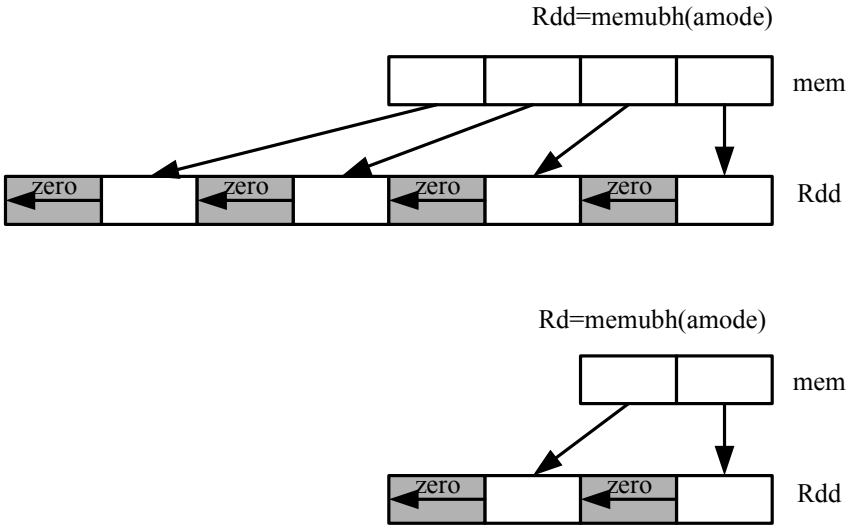
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN						Parse																	
1	0	0	1	0	1	1	0	0	0	0	1	1	1	1	0	**P**	**P**	0	0	0	0	-	-	-	-	-	1	1	1	1	0	dealloc_return
ICLASS				Amode			Type			UN						Parse						s2										
1	0	0	1	0	1	1	0	0	0	0	1	1	1	1	0	**P**	**P**	0	0	1	0	**s**	**s**	-	-	-	1	1	1	1	0	if (Ps.new) dealloc_return:nt
1	0	0	1	0	1	1	0	0	0	0	1	1	1	1	0	**P**	**P**	0	1	0	0	**s**	**s**	-	-	-	1	1	1	1	0	if (Ps) dealloc_return
1	0	0	1	0	1	1	0	0	0	0	1	1	1	1	0	**P**	**P**	0	1	1	0	**s**	**s**	-	-	-	1	1	1	1	0	if (Ps.new) dealloc_return:t
1	0	0	1	0	1	1	0	0	0	0	1	1	1	1	0	**P**	**P**	1	0	1	0	**s**	**s**	-	-	-	1	1	1	1	0	if (!Ps.new) dealloc_return:nt
1	0	0	1	0	1	1	0	0	0	0	1	1	1	1	0	**P**	**P**	1	1	0	0	**s**	**s**	-	-	-	1	1	1	1	0	if (!Ps) dealloc_return
1	0	0	1	0	1	1	0	0	0	0	1	1	1	1	0	**P**	**P**	1	1	1	0	**s**	**s**	-	-	-	1	1	1	1	0	if (!Ps.new) dealloc_return:t

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
s2	Field to encode register s

Load and unpack bytes to halfwords

Load contiguous bytes from memory and vector unpack them into halfwords.


Rdd=memubh(amode)
mem
zero
zero
zero
zero
Rdd
Rd=memubh(amode)
mem
zero
zero
Rdd

Syntax	Behavior

`Rd=membh(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  };`
	`};`
	`;`
	`Re=#U;`

`Rd=membh(Rs)`	`Assembler mapped to: "Rd=membh""(Rs+#0)"`

`Rd=membh(Rs+#s11:1)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  };`
	`};`
	`;`

`Rd=membh(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  };`
	`};`
	`;`

`Rd=membh(Rx++#s4:1)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  };`
	`};`
	`;`

`Rd=membh(Rx++#s4:1:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  };`
	`};`
	`;`

`Rd=membh(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  };`
	`};`
	`;`

Syntax	Behavior

`Rd=membh(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  };`
	`};`
	`;`

`Rd=membh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.b[i];`
	`  };`
	`};`
	`;`

`Rd=memubh(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`
	`Re=#U;`

`Rd=memubh(Rs+#s11:1)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`

`Rd=memubh(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`

`Rd=memubh(Rx++#s4:1)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<2;i++) {`
	`  Rd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`

Syntax	Behavior

Rd=memubh(Rx++#s4:1:circ(Mu))	EA=Rx;
	Rx=Rx=circ_add(Rx,#s,MuV);
	{
	  tmpV = *EA;
	  for (i=0;i<2;i++) {
	  Rd.h[i]=tmpV.ub[i];
	  };
	};
	;

Rd=memubh(Rx++I:circ(Mu))	EA=Rx;
	Rx=Rx=circ_add(Rx,I<<1,MuV);
	{
	  tmpV = *EA;
	  for (i=0;i<2;i++) {
	  Rd.h[i]=tmpV.ub[i];
	  };
	};
	;

Rd=memubh(Rx++Mu)	EA=Rx;
	Rx=Rx+MuV;
	{
	  tmpV = *EA;
	  for (i=0;i<2;i++) {
	  Rd.h[i]=tmpV.ub[i];
	  };
	};
	;

Rd=memubh(Rx++Mu:brev)	EA=Rx.h[1] | brev(Rx.h[0]);
	Rx=Rx+MuV;
	{
	  tmpV = *EA;
	  for (i=0;i<2;i++) {
	  Rd.h[i]=tmpV.ub[i];
	  };
	};
	;

Rdd=membh(Re=#U6)	apply_extension(#U);
	EA=#U;
	{
	  tmpV = *EA;
	  for (i=0;i<4;i++) {
	  Rdd.h[i]=tmpV.b[i];
	  };
	};
	;
	Re=#U;

Rdd=membh(Rs)	Assembler mapped to: "Rdd=membh""(Rs+#0)"

Rdd=membh(Rs+#s11:2)	apply_extension(#s);
	EA=Rs+#s;
	{
	  tmpV = *EA;
	  for (i=0;i<4;i++) {
	  Rdd.h[i]=tmpV.b[i];
	  };
	};
	;

Syntax	Behavior

`Rdd=membh(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.b[i];`
	`  };`
	`};`
	`;`

`Rdd=membh(Rx++#s4:2)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.b[i];`
	`  };`
	`};`
	`;`

`Rdd=membh(Rx++#s4:2:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.b[i];`
	`  };`
	`};`
	`;`

`Rdd=membh(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<2,MuV);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.b[i];`
	`  };`
	`};`
	`;`

`Rdd=membh(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.b[i];`
	`  };`
	`};`
	`;`

`Rdd=membh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.b[i];`
	`  };`
	`};`
	`;`

Syntax	Behavior

`Rdd=memubh(Re=#U6)`	`apply_extension(#U);`
	`EA=#U;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`
	`Re=#U;`

`Rdd=memubh(Rs+#s11:2)`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`

`Rdd=memubh(Rt<<#u2+#U6)`	`apply_extension(#U);`
	`EA=#U+(Rt<<#u);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`

Syntax	Behavior

`Rdd=memubh(Rx++#s4:2)`	`EA=Rx;`
	`Rx=Rx+#s;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`

`Rdd=memubh(Rx++#s4:2:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`

`Rdd=memubh(Rx++I:circ(Mu))`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<2,MuV);`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`

`Rdd=memubh(Rx++Mu)`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`

`Rdd=memubh(Rx++Mu:brev)`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`{`
	`  tmpV = *EA;`
	`  for (i=0;i<4;i++) {`
	`  Rdd.h[i]=tmpV.ub[i];`
	`  };`
	`};`
	`;`

Class: LD (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	i	i	0	0	0	1	s	s	s	s	s	P	P	i	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=membh(Rs+#s11:1)
1	0	0	1	0	i	i	0	0	1	1	s	s	s	s	s	P	P	i	i	i	i	i	i	i	i	i	d	d	d	d	d	Rd=memubh(Rs+#s11:1)
1	0	0	1	0	i	i	0	1	0	1	s	s	s	s	s	P	P	i	i	i	i	i	i	i	i	i	d	d	d	d	d	Rdd=memubh(Rs+#s11:2)
1	0	0	1	0	i	i	0	1	1	1	s	s	s	s	s	P	P	i	i	i	i	i	i	i	i	i	d	d	d	d	d	Rdd=membh(Rs+#s11:2)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	0	0	0	0	0	1	x	x	x	x	x	P	P	u	0	-	-	0	i	i	i	i	d	d	d	d	d	Rd=membh(Rx++#s4:1:circ(Mu))
1	0	0	1	1	0	0	0	0	0	1	x	x	x	x	x	P	P	u	0	-	-	1	-	0	-	-	d	d	d	d	d	Rd=membh(Rx++I:circ(Mu))
1	0	0	1	1	0	0	0	0	1	1	x	x	x	x	x	P	P	u	0	-	-	0	i	i	i	i	d	d	d	d	d	Rd=memubh(Rx++#s4:1:circ(Mu))
1	0	0	1	1	0	0	0	0	1	1	x	x	x	x	x	P	P	u	0	-	-	1	-	0	-	-	d	d	d	d	d	Rd=memubh(Rx++I:circ(Mu))
1	0	0	1	1	0	0	0	1	0	1	x	x	x	x	x	P	P	u	0	-	-	0	i	i	i	i	d	d	d	d	d	Rdd=memubh(Rx++#s4:2:circ(Mu))
1	0	0	1	1	0	0	0	1	0	1	x	x	x	x	x	P	P	u	0	-	-	1	-	0	-	-	d	d	d	d	d	Rdd=memubh(Rx++I:circ(Mu))
1	0	0	1	1	0	0	0	1	1	1	x	x	x	x	x	P	P	u	0	-	-	0	i	i	i	i	d	d	d	d	d	Rdd=membh(Rx++#s4:2:circ(Mu))
1	0	0	1	1	0	0	0	1	1	1	x	x	x	x	x	P	P	u	0	-	-	1	-	0	-	-	d	d	d	d	d	Rdd=membh(Rx++I:circ(Mu))
ICLASS				Amode			Type			UN	e5					Parse											d5					
1	0	0	1	1	0	1	0	0	0	1	e	e	e	e	e	P	P	0	1	l	l	l	l	-	l	l	d	d	d	d	d	Rd=membh(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											d5					
1	0	0	1	1	0	1	0	0	0	1	x	x	x	x	x	P	P	0	0	-	-	-	i	i	i	i	d	d	d	d	d	Rd=membh(Rx++#s4:1)
ICLASS				Amode			Type			UN	e5					Parse											d5					
1	0	0	1	1	0	1	0	0	1	1	e	e	e	e	e	P	P	0	1	l	l	l	l	-	l	l	d	d	d	d	d	Rd=memubh(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											d5					
1	0	0	1	1	0	1	0	0	1	1	x	x	x	x	x	P	P	0	0	-	-	-	i	i	i	i	d	d	d	d	d	Rd=memubh(Rx++#s4:1)
ICLASS				Amode			Type			UN	e5					Parse											d5					
1	0	0	1	1	0	1	0	1	0	1	e	e	e	e	e	P	P	0	1	l	l	l	l	-	l	l	d	d	d	d	d	Rdd=memubh(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											d5					
1	0	0	1	1	0	1	0	1	0	1	x	x	x	x	x	P	P	0	0	-	-	-	i	i	i	i	d	d	d	d	d	Rdd=memubh(Rx++#s4:2)
ICLASS				Amode			Type			UN	e5					Parse											d5					
1	0	0	1	1	0	1	0	1	1	1	e	e	e	e	e	P	P	0	1	l	l	l	l	-	l	l	d	d	d	d	d	Rdd=membh(Re=#U6)
ICLASS				Amode			Type			UN	x5					Parse											d5					
1	0	0	1	1	0	1	0	1	1	1	x	x	x	x	x	P	P	0	0	-	-	-	i	i	i	i	d	d	d	d	d	Rdd=membh(Rx++#s4:2)
ICLASS				Amode			Type			UN	t5					Parse											d5					
1	0	0	1	1	1	0	0	0	0	1	t	t	t	t	t	P	P	i	1	l	l	l	l	i	l	l	d	d	d	d	d	Rd=membh(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	1	1	1	0	0	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	-	-	-	-	0	-	-	**d**	**d**	**d**	**d**	**d**	Rd=membh(Rx++Mu)
ICLASS				Amode			Type			UN	t5					Parse											d5					
1	0	0	1	1	1	0	0	0	1	1	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**l**	**l**	**l**	**l**	**i**	**l**	**l**	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	1	0	0	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	-	-	-	-	0	-	-	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rx++Mu)
ICLASS				Amode			Type			UN	t5					Parse											d5					
1	0	0	1	1	1	0	0	1	0	1	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**l**	**l**	**l**	**l**	**i**	**l**	**l**	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	1	0	0	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	-	-	-	-	0	-	-	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rx++Mu)
ICLASS				Amode			Type			UN	t5					Parse											d5					
1	0	0	1	1	1	0	0	1	1	1	**t**	**t**	**t**	**t**	**t**	**P**	**P**	**i**	1	**l**	**l**	**l**	**l**	**i**	**l**	**l**	**d**	**d**	**d**	**d**	**d**	Rdd=membh(Rt<<#u2+#U6)
ICLASS				Amode			Type			UN	x5					Parse		u1									d5					
1	0	0	1	1	1	0	0	1	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	-	-	-	-	0	-	-	**d**	**d**	**d**	**d**	**d**	Rdd=membh(Rx++Mu)
1	0	0	1	1	1	1	0	0	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	-	-	-	-	0	-	-	**d**	**d**	**d**	**d**	**d**	Rd=membh(Rx++Mu:brev)
1	0	0	1	1	1	1	0	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	-	-	-	-	0	-	-	**d**	**d**	**d**	**d**	**d**	Rd=memubh(Rx++Mu:brev)
1	0	0	1	1	1	1	0	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	-	-	-	-	0	-	-	**d**	**d**	**d**	**d**	**d**	Rdd=memubh(Rx++Mu:brev)
1	0	0	1	1	1	1	0	1	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	0	-	-	-	-	0	-	-	**d**	**d**	**d**	**d**	**d**	Rdd=membh(Rx++Mu:brev)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
x5	Field to encode register x

11.8 MEMOP

The MEMOP instruction class includes simple operations on values in memory.

MEMOP instructions are executable on slot 0.

Operation on memory byte

Perform ALU or bit operation on the memory byte at the effective address.

Syntax	Behavior

`memb(Rs+#u6:0)=clrbit(#U5)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp &= (~(1<<#U));`
	`*EA = tmp;`

`memb(Rs+#u6:0)=setbit(#U5)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp |= (1<<#U);`
	`*EA = tmp;`

`memb(Rs+#u6:0)[+-]=#U5`	`apply_extension(#u);`
	`EA=Rs[+-]#u;`
	`tmp = *EA;`
	`tmp [+-]= #U;`
	`*EA = tmp;`

`memb(Rs+#u6:0)[+-|&]=Rt`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp [+-|&]= Rt;`
	`*EA = tmp;`

Class: MEMOP (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse											t5					
0	0	1	1	1	1	1	0	-	0	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	0	0	t	t	t	t	t	memb(Rs+#u6:0)+=Rt
0	0	1	1	1	1	1	0	-	0	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	0	1	t	t	t	t	t	memb(Rs+#u6:0)-=Rt
0	0	1	1	1	1	1	0	-	0	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	1	0	t	t	t	t	t	memb(Rs+#u6:0)&=Rt
0	0	1	1	1	1	1	0	-	0	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	1	1	t	t	t	t	t	memb(Rs+#u6:0)|=Rt
ICLASS											s5					Parse																
0	0	1	1	1	1	1	1	-	0	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	0	0	I	I	I	I	I	memb(Rs+#u6:0)+=#U5
0	0	1	1	1	1	1	1	-	0	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	0	1	I	I	I	I	I	memb(Rs+#u6:0)-=#U5
0	0	1	1	1	1	1	1	-	0	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	1	0	I	I	I	I	I	memb(Rs+#u6:0)=clrbit(#U5)
0	0	1	1	1	1	1	1	-	0	0	s	s	s	s	s	P	P	0	i	i	i	i	i	i	1	1	I	I	I	I	I	memb(Rs+#u6:0)=setbit(#U5)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t

Operation on memory halfword

Perform ALU or bit operation on the memory halfword at the effective address.

Syntax	Behavior

`memh(Rs+#u6:1)=clrbit(#U5)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp &= (~(1<<#U));`
	`*EA = tmp;`

`memh(Rs+#u6:1)=setbit(#U5)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp |= (1<<#U);`
	`*EA = tmp;`

`memh(Rs+#u6:1)[+-]=#U5`	`apply_extension(#u);`
	`EA=Rs[+-]#u;`
	`tmp = *EA;`
	`tmp [+-]= #U;`
	`*EA = tmp;`

`memh(Rs+#u6:1)[+-|&]=Rt`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp [+-|&]= Rt;`
	`*EA = tmp;`

Class: MEMOP (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse											t5					
0	0	1	1	1	1	1	0	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	0	0	**t**	**t**	**t**	**t**	**t**	memh(Rs+#u6:1)+=Rt
0	0	1	1	1	1	1	0	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	0	1	**t**	**t**	**t**	**t**	**t**	memh(Rs+#u6:1)-=Rt
0	0	1	1	1	1	1	0	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	1	0	**t**	**t**	**t**	**t**	**t**	memh(Rs+#u6:1)&=Rt
0	0	1	1	1	1	1	0	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	1	1	**t**	**t**	**t**	**t**	**t**	memh(Rs+#u6:1)|=Rt
ICLASS											s5					Parse																
0	0	1	1	1	1	1	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	0	0	**I**	**I**	**I**	**I**	**I**	memh(Rs+#u6:1)+=#U5
0	0	1	1	1	1	1	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	0	1	**I**	**I**	**I**	**I**	**I**	memh(Rs+#u6:1)-=#U5
0	0	1	1	1	1	1	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	1	0	**I**	**I**	**I**	**I**	**I**	memh(Rs+#u6:1)=clrbit(#U5)
0	0	1	1	1	1	1	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	1	1	**I**	**I**	**I**	**I**	**I**	memh(Rs+#u6:1)=setbit(#U5)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t

Operation on memory word

Perform ALU or bit operation on the memory word at the effective address.

Syntax	Behavior

`memw(Rs+#u6:2)=clrbit(#U5)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp &= (~(1<<#U));`
	`*EA = tmp;`

`memw(Rs+#u6:2)=setbit(#U5)`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp |= (1<<#U);`
	`*EA = tmp;`

`memw(Rs+#u6:2)[+-]=#U5`	`apply_extension(#u);`
	`EA=Rs[+-]#u;`
	`tmp = *EA;`
	`tmp [+-]= #U;`
	`*EA = tmp;`

`memw(Rs+#u6:2)[+-|&]=Rt`	`apply_extension(#u);`
	`EA=Rs+#u;`
	`tmp = *EA;`
	`tmp [+-|&]= Rt;`
	`*EA = tmp;`

Class: MEMOP (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse											t5					
0	0	1	1	1	1	1	0	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	0	0	**t**	**t**	**t**	**t**	**t**	memw(Rs+#u6:2)+=Rt
0	0	1	1	1	1	1	0	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	0	1	**t**	**t**	**t**	**t**	**t**	memw(Rs+#u6:2)-=Rt
0	0	1	1	1	1	1	0	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	1	0	**t**	**t**	**t**	**t**	**t**	memw(Rs+#u6:2)&=Rt
0	0	1	1	1	1	1	0	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	1	1	**t**	**t**	**t**	**t**	**t**	memw(Rs+#u6:2)|=Rt
ICLASS											s5					Parse																
0	0	1	1	1	1	1	1	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	0	0	**I**	**I**	**I**	**I**	**I**	memw(Rs+#u6:2)+=#U5
0	0	1	1	1	1	1	1	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	0	1	**I**	**I**	**I**	**I**	**I**	memw(Rs+#u6:2)-=#U5
0	0	1	1	1	1	1	1	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	1	0	**I**	**I**	**I**	**I**	**I**	memw(Rs+#u6:2)=clrbit(#U5)
0	0	1	1	1	1	1	1	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	1	1	**I**	**I**	**I**	**I**	**I**	memw(Rs+#u6:2)=setbit(#U5)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t

11.9 NV

The NV instruction class includes instructions which take the register source operand from another instruction in the same packet.

NV instructions are executable on slot 0.

11.9.1 NV/J

The NV/J instruction subclass includes jump instructions which take the register source operand from another instruction in the same packet.

Jump to address condition on new register value

Compare a register or constant against the value produced by a slot 1 instruction. If the comparison is true, the program counter is changed to a target address, relative to the current PC.

This instruction is executable only on slot 0.

Syntax	Behavior
`if ([!]cmp.eq(Ns.new,#-1)) jump:<hint> #r9:2`	`; if ((Ns.new[!]=(-1))) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`
`if ([!]cmp.eq(Ns.new,#U5)) jump:<hint> #r9:2`	`; if ((Ns.new[!]=(#U))) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`
`if ([!]cmp.eq(Ns.new,Rt)) jump:<hint> #r9:2`	`; if ((Ns.new[!]=Rt)) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`
`if ([!]cmp.gt(Ns.new,#-1)) jump:<hint> #r9:2`	`; if ([!](Ns.new>(-1))) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`
`if ([!]cmp.gt(Ns.new,#U5)) jump:<hint> #r9:2`	`; if ([!](Ns.new>(#U))) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`
`if ([!]cmp.gt(Ns.new,Rt)) jump:<hint> #r9:2`	`; if ([!](Ns.new>Rt)) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`

Syntax	Behavior
`if ([!]cmp.gt(Rt,Ns.new)) jump:<hint> #r9:2`	`; if ([!](Rt>Ns.new)) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`
`if ([!]cmp.gtu(Ns.new,#U5)) jump:<hint> #r9:2`	`; if ([!](Ns.new.uw[0]>(#U))) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`
`if ([!]cmp.gtu(Ns.new,Rt)) jump:<hint> #r9:2`	`; if ([!](Ns.new.uw[0]>Rt.uw[0])) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`
`if ([!]cmp.gtu(Rt,Ns.new)) jump:<hint> #r9:2`	`; if ([!](Rt.uw[0]>Ns.new.uw[0])) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`
`if ([!]tstbit(Ns.new,#0)) jump:<hint> #r9:2`	`; if ([!]((Ns.new) & 1)) { apply_extension(#r); #r=#r & ~0x3; PC=PC+#r; };`

Class: NV (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS													s3			Parse			t5														
0	0	1	0	0	0	0	0	0	0	i	i	-	s	s	s	P	P	0	t	t	t	t	t	i	i	i	i	i	i	i	-	if (cmp.eq(Ns.new,Rt)) jump:nt #r9:2
0	0	1	0	0	0	0	0	0	0	i	i	-	s	s	s	P	P	1	t	t	t	t	t	i	i	i	i	i	i	i	-	if (cmp.eq(Ns.new,Rt)) jump:t #r9:2
0	0	1	0	0	0	0	0	0	1	i	i	-	s	s	s	P	P	0	t	t	t	t	t	i	i	i	i	i	i	i	-	if (!cmp.eq(Ns.new,Rt)) jump:nt #r9:2
0	0	1	0	0	0	0	0	0	1	i	i	-	s	s	s	P	P	1	t	t	t	t	t	i	i	i	i	i	i	i	-	if (!cmp.eq(Ns.new,Rt)) jump:t #r9:2
0	0	1	0	0	0	0	0	1	0	i	i	-	s	s	s	P	P	0	t	t	t	t	t	i	i	i	i	i	i	i	-	if (cmp.gt(Ns.new,Rt)) jump:nt #r9:2
0	0	1	0	0	0	0	0	1	0	i	i	-	s	s	s	P	P	1	t	t	t	t	t	i	i	i	i	i	i	i	-	if (cmp.gt(Ns.new,Rt)) jump:t #r9:2
0	0	1	0	0	0	0	0	1	1	i	i	-	s	s	s	P	P	0	t	t	t	t	t	i	i	i	i	i	i	i	-	if (!cmp.gt(Ns.new,Rt)) jump:nt #r9:2
0	0	1	0	0	0	0	0	1	1	i	i	-	s	s	s	P	P	1	t	t	t	t	t	i	i	i	i	i	i	i	-	if (!cmp.gt(Ns.new,Rt)) jump:t #r9:2
0	0	1	0	0	0	0	1	0	0	i	i	-	s	s	s	P	P	0	t	t	t	t	t	i	i	i	i	i	i	i	-	if (cmp.gtu(Ns.new,Rt)) jump:nt #r9:2
0	0	1	0	0	0	0	1	0	0	i	i	-	s	s	s	P	P	1	t	t	t	t	t	i	i	i	i	i	i	i	-	if (cmp.gtu(Ns.new,Rt)) jump:t #r9:2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	0	0	0	1	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gtu(Ns.new,Rt)) jump:nt #r9:2
0	0	1	0	0	0	0	1	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gtu(Ns.new,Rt)) jump:t #r9:2
0	0	1	0	0	0	0	1	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gt(Rt,Ns.new)) jump:nt #r9:2
0	0	1	0	0	0	0	1	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gt(Rt,Ns.new)) jump:t #r9:2
0	0	1	0	0	0	0	1	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gt(Rt,Ns.new)) jump:nt #r9:2
0	0	1	0	0	0	0	1	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gt(Rt,Ns.new)) jump:t #r9:2
0	0	1	0	0	0	1	0	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gtu(Rt,Ns.new)) jump:nt #r9:2
0	0	1	0	0	0	1	0	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gtu(Rt,Ns.new)) jump:t #r9:2
0	0	1	0	0	0	1	0	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gtu(Rt,Ns.new)) jump:nt #r9:2
0	0	1	0	0	0	1	0	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gtu(Rt,Ns.new)) jump:t #r9:2
ICLASS													s3			Parse																
0	0	1	0	0	1	0	0	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.eq(Ns.new,#U5)) jump:nt #r9:2
0	0	1	0	0	1	0	0	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.eq(Ns.new,#U5)) jump:t #r9:2
0	0	1	0	0	1	0	0	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.eq(Ns.new,#U5)) jump:nt #r9:2
0	0	1	0	0	1	0	0	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.eq(Ns.new,#U5)) jump:t #r9:2
0	0	1	0	0	1	0	0	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gt(Ns.new,#U5)) jump:nt #r9:2
0	0	1	0	0	1	0	0	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gt(Ns.new,#U5)) jump:t #r9:2
0	0	1	0	0	1	0	0	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gt(Ns.new,#U5)) jump:nt #r9:2
0	0	1	0	0	1	0	0	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gt(Ns.new,#U5)) jump:t #r9:2
0	0	1	0	0	1	0	1	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gtu(Ns.new,#U5)) jump:nt #r9:2
0	0	1	0	0	1	0	1	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gtu(Ns.new,#U5)) jump:t #r9:2
0	0	1	0	0	1	0	1	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gtu(Ns.new,#U5)) jump:nt #r9:2
0	0	1	0	0	1	0	1	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**I**	**I**	**I**	**I**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gtu(Ns.new,#U5)) jump:t #r9:2
0	0	1	0	0	1	0	1	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (tstbit(Ns.new,#0)) jump:nt #r9:2
0	0	1	0	0	1	0	1	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (tstbit(Ns.new,#0)) jump:t #r9:2
0	0	1	0	0	1	0	1	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!tstbit(Ns.new,#0)) jump:nt #r9:2
0	0	1	0	0	1	0	1	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!tstbit(Ns.new,#0)) jump:t #r9:2
0	0	1	0	0	1	1	0	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.eq(Ns.new,#-1)) jump:nt #r9:2
0	0	1	0	0	1	1	0	0	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.eq(Ns.new,#-1)) jump:t #r9:2
0	0	1	0	0	1	1	0	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.eq(Ns.new,#-1)) jump:nt #r9:2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	0	1	0	0	1	1	0	0	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.eq(Ns.new,#-1)) jump:t #r9:2
0	0	1	0	0	1	1	0	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gt(Ns.new,#-1)) jump:nt #r9:2
0	0	1	0	0	1	1	0	1	0	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (cmp.gt(Ns.new,#-1)) jump:t #r9:2
0	0	1	0	0	1	1	0	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	0	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gt(Ns.new,#-1)) jump:nt #r9:2
0	0	1	0	0	1	1	0	1	1	**i**	**i**	**-**	**s**	**s**	**s**	**P**	**P**	1	**-**	**-**	**-**	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**-**	if (!cmp.gt(Ns.new,#-1)) jump:t #r9:2

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s3	Field to encode register s
t5	Field to encode register t

11.9.2 NV/ST

The NV/ST instruction subclass includes store instructions which take the register source operand from another instruction in the same packet.

Store new-value byte

Store the least-significant byte in a source register in memory at the effective address.

Syntax	Behavior

`memb(Re=#U6)=Nt.new`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Nt.new&0xff;`
	`Re=#U;`

`memb(Rs+#s11:0)=Nt.new`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Nt.new&0xff;`

`memb(Rs+Ru<<#u2)=Nt.new`	`EA=Rs+(Ru<<#u);` `*EA = Nt.new&0xff;`

`memb(Ru<<#u2+#U6)=Nt.new`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Nt.new&0xff;`

`memb(Rx++#s4:0)=Nt.new`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Nt.new&0xff;`

`memb(Rx++#s4:0:circ(Mu))=Nt.new`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Nt.new&0xff;`

`memb(Rx++I:circ(Mu))=Nt.new`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<0,MuV);`
	`*EA = Nt.new&0xff;`

`memb(Rx++Mu)=Nt.new`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Nt.new&0xff;`

`memb(Rx++Mu:brev)=Nt.new`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Nt.new&0xff;`

`memb(gp+#u16:0)=Nt.new`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Nt.new&0xff;`

Class: NV (slots 0)

Notes

- Forms of this instruction which use a new-value operand produced in the packet must execute on slot 0.
- This instruction can execute only in slot 0, even though it is an ST instruction.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5										t3			
0	0	1	1	1	0	1	1	1	0	1	s	s	s	s	s	P	P	i	u	u	u	u	u	i	-	-	0	0	t	t	t	memb(Rs+Ru<<#u2)=Nt.new
ICLASS								Type								Parse					t3											
0	1	0	0	1	i	i	0	1	0	1	i	i	i	i	i	P	P	i	0	0	t	t	t	i	i	i	i	i	i	i	i	memb(gp+#u16:0)=Nt.new
ICLASS				Amode			Type			U N	s5					Parse					t3											
1	0	1	0	0	i	i	1	1	0	1	s	s	s	s	s	P	P	i	0	0	t	t	t	i	i	i	i	i	i	i	i	memb(Rs+#s11:0)=Nt.new
ICLASS				Amode			Type			U N	x5					Parse		u1			t3											
1	0	1	0	1	0	0	1	1	0	1	x	x	x	x	x	P	P	u	0	0	t	t	t	0	-	-	-	-	-	1	-	memb(Rx++I:circ(Mu))=Nt.new
1	0	1	0	1	0	0	1	1	0	1	x	x	x	x	x	P	P	u	0	0	t	t	t	0	i	i	i	i	-	0	-	memb(Rx++#s4:0:circ(Mu))=Nt.new
ICLASS				Amode			Type			U N	e5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	e	e	e	e	e	P	P	0	0	0	t	t	t	1	-	l	l	l	l	l	l	memb(Re=#U6)=Nt.new
ICLASS				Amode			Type			U N	x5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	x	x	x	x	x	P	P	0	0	0	t	t	t	0	i	i	i	i	-	0	-	memb(Rx++#s4:0)=Nt.new
ICLASS				Amode			Type			U N	u5					Parse					t3											
1	0	1	0	1	1	0	1	1	0	1	u	u	u	u	u	P	P	i	0	0	t	t	t	1	i	l	l	l	l	l	l	memb(Ru<<#u2+#U6)=Nt.new
ICLASS				Amode			Type			U N	x5					Parse		u1			t3											
1	0	1	0	1	1	0	1	1	0	1	x	x	x	x	x	P	P	u	0	0	t	t	t	0	-	-	-	-	-	-	-	memb(Rx++Mu)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	x	x	x	x	x	P	P	u	0	0	t	t	t	0	-	-	-	-	-	-	-	memb(Rx++Mu:brev)=Nt.new

Field name	Description
ICLASS	Instruction Class
Type	Type
Parse	Packet/Loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t3	Field to encode register t
u1	Field to encode register u
u5	Field to encode register u
x5	Field to encode register x

Field name	Description
Amode	Amode
Type	Type
UN	Unsigned

Store new-value byte conditionally

Store the least-significant byte in a source register in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior
`if ([!]Pv[.new]) memb(#u6)=Nt.new`	`if ([!]Pv[.new][0]) { apply_extension(#u); EA=#u; *EA = Nt[.new]&0xff; } else { NOP; };`
`if ([!]Pv[.new]) memb(Rs+#u6:0)=Nt.new`	`apply_extension(#u); if ([!]Pv[.new][0]) { EA=Rs+#u; *EA = Nt[.new]&0xff; } else { NOP; };`
`if ([!]Pv[.new]) memb(Rs+Ru<<#u2)=Nt.new`	`if ([!]Pv[.new][0]) { EA=Rs+(Ru<<#u); *EA = Nt[.new]&0xff; } else { NOP; };`
`if ([!]Pv[.new]) memb(Rx++#s4:0)=Nt.new`	`if ([!]Pv[.new][0]){ EA=Rx; Rx=Rx+#s; *EA = Nt[.new]&0xff; } else { NOP; };`

Class: NV (slots 0)

Notes

- Forms of this instruction which use a new-value operand produced in the packet must execute on slot 0.
- This instruction can execute only in slot 0, even though it is an ST instruction.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5									t3			
0	0	1	1	0	1	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	0	**t**	**t**	**t**	if (Pv) memb(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	0	**t**	**t**	**t**	if (!Pv) memb(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	0	**t**	**t**	**t**	if (Pv.new) memb(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	0	**t**	**t**	**t**	if (!Pv.new) memb(Rs+Ru<<#u2)=Nt.new
ICLASS					Sense	PredNew		Type			s5					Parse					t3											
0	1	0	0	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memb(Rs+#u6:0)=Nt.new
0	1	0	0	0	0	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memb(Rs+#u6:0)=Nt.new
0	1	0	0	0	1	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memb(Rs+#u6:0)=Nt.new
0	1	0	0	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memb(Rs+#u6:0)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memb(Rx++#s4:0)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memb(Rx++#s4:0)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memb(Rx++#s4:0)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memb(Rx++#s4:0)=Nt.new
ICLASS				Amode			Type			UN						Parse					t3											
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	0	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memb(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	0	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memb(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memb(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	0	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memb(#u6)=Nt.new

Field name	Description
ICLASS	Instruction Class
Type	Type
PredNew	PredNew

Field name	Description
Sense	Sense
Parse	Packet/Loop parse bits
s5	Field to encode register s
t3	Field to encode register t
u5	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store new-value halfword

Store the upper or lower 16-bits of a source register in memory at the effective address.

Syntax	Behavior

`memh(Re=#U6)=Nt.new`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Nt.new.h[0];`
	`Re=#U;`

`memh(Rs+#s11:1)=Nt.new`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Nt.new.h[0];`

`memh(Rs+Ru<<#u2)=Nt.new`	`EA=Rs+(Ru<<#u);` `*EA = Nt.new.h[0];`

`memh(Ru<<#u2+#U6)=Nt.new`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Nt.new.h[0];`

`memh(Rx++#s4:1)=Nt.new`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Nt.new.h[0];`

`memh(Rx++#s4:1:circ(Mu))=Nt.new`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Nt.new.h[0];`

`memh(Rx++I:circ(Mu))=Nt.new`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`*EA = Nt.new.h[0];`

`memh(Rx++Mu)=Nt.new`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Nt.new.h[0];`

`memh(Rx++Mu:brev)=Nt.new`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Nt.new.h[0];`

`memh(gp+#u16:1)=Nt.new`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Nt.new.h[0];`

Class: NV (slots 0)

Notes

- Forms of this instruction which use a new-value operand produced in the packet must execute on slot 0.
- This instruction can execute only in slot 0, even though it is an ST instruction.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5										t3			
0	0	1	1	1	0	1	1	1	0	1	s	s	s	s	s	P	P	i	u	u	u	u	u	i	-	-	0	1	t	t	t	memh(Rs+Ru<<#u2)=Nt.new
ICLASS								Type								Parse					t3											
0	1	0	0	1	i	i	0	1	0	1	i	i	i	i	i	P	P	i	0	1	t	t	t	i	i	i	i	i	i	i	i	memh(gp+#u16:1)=Nt.new
ICLASS				Amode			Type			U N	s5					Parse					t3											
1	0	1	0	0	i	i	1	1	0	1	s	s	s	s	s	P	P	i	0	1	t	t	t	i	i	i	i	i	i	i	i	memh(Rs+#s11:1)=Nt.new
ICLASS				Amode			Type			U N	x5					Parse	u1				t3											
1	0	1	0	1	0	0	1	1	0	1	x	x	x	x	x	P	P	u	0	1	t	t	t	0	-	-	-	-	-	1	-	memh(Rx++I:circ(Mu))=Nt.new
1	0	1	0	1	0	0	1	1	0	1	x	x	x	x	x	P	P	u	0	1	t	t	t	0	i	i	i	i	-	0	-	memh(Rx++#s4:1:circ(Mu))=Nt.new
ICLASS				Amode			Type			U N	e5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	e	e	e	e	e	P	P	0	0	1	t	t	t	1	-	l	l	l	l	l	l	memh(Re=#U6)=Nt.new
ICLASS				Amode			Type			U N	x5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	x	x	x	x	x	P	P	0	0	1	t	t	t	0	i	i	i	i	-	0	-	memh(Rx++#s4:1)=Nt.new
ICLASS				Amode			Type			U N	u5					Parse					t3											
1	0	1	0	1	1	0	1	1	0	1	u	u	u	u	u	P	P	i	0	1	t	t	t	1	i	l	l	l	l	l	l	memh(Ru<<#u2+#U6)=Nt.new
ICLASS				Amode			Type			U N	x5					Parse	u1				t3											
1	0	1	0	1	1	0	1	1	0	1	x	x	x	x	x	P	P	u	0	1	t	t	t	0	-	-	-	-	-	-	-	memh(Rx++Mu)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	x	x	x	x	x	P	P	u	0	1	t	t	t	0	-	-	-	-	-	-	-	memh(Rx++Mu:brev)=Nt.new

Field name	Description
ICLASS	Instruction Class
Type	Type
Parse	Packet/Loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t3	Field to encode register t
u1	Field to encode register u
u5	Field to encode register u
x5	Field to encode register x

Field name	Description
Amode	Amode
Type	Type
UN	Unsigned

Store new-value halfword conditionally

Store the upper or lower 16-bits of a source register in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv[.new]) memh(#u6)=Nt.new`	`if ([!]Pv[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=#u;`
	`  *EA = Nt[.new].h[0];`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new]) memh(Rs+#u6:1)=Nt.new`	`apply_extension(#u);`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+#u;`
	`  *EA = Nt[.new].h[0];`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new]) memh(Rs+Ru<<#u2)=Nt.new`	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+(Ru<<#u);`
	`  *EA = Nt[.new].h[0];`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new]) memh(Rx++#s4:1)=Nt.new`	`if ([!]Pv[.new][0]){`
	`  EA=Rx;`
	`  Rx=Rx+#s;`
	`  *EA = Nt[.new].h[0];`
	`} else {`
	`  NOP;`
	`};`

Class: NV (slots 0)

Notes

- Forms of this instruction which use a new-value operand produced in the packet must execute on slot 0.
- This instruction can execute only in slot 0, even though it is an ST instruction.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5									t3			
0	0	1	1	0	1	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	1	**t**	**t**	**t**	if (Pv) memh(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	1	**t**	**t**	**t**	if (!Pv) memh(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	1	**t**	**t**	**t**	if (Pv.new) memh(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	0	1	**t**	**t**	**t**	if (!Pv.new) memh(Rs+Ru<<#u2)=Nt.new
ICLASS					Sense	PredNew			Type		s5					Parse					t3											
0	1	0	0	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	1	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(Rs+#u6:1)=Nt.new
0	1	0	0	0	0	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	1	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(Rs+#u6:1)=Nt.new
0	1	0	0	0	1	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	1	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memh(Rs+#u6:1)=Nt.new
0	1	0	0	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	0	1	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memh(Rs+#u6:1)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(Rx++#s4:1)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memh(Rx++#s4:1)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(Rx++#s4:1)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	0	1	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memh(Rx++#s4:1)=Nt.new
ICLASS				Amode			Type			UN						Parse					t3											
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	0	1	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	0	1	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memh(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	0	1	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memh(#u6)=Nt.new

Field name	Description
ICLASS	Instruction Class
Type	Type
PredNew	PredNew

Field name	Description
Sense	Sense
Parse	Packet/Loop parse bits
s5	Field to encode register s
t3	Field to encode register t
u5	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store new-value word

Store a 32-bit register in memory at the effective address.

Syntax	Behavior

`memw(Re=#U6)=Nt.new`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Nt.new;`
	`Re=#U;`

`memw(Rs+#s11:2)=Nt.new`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Nt.new;`

`memw(Rs+Ru<<#u2)=Nt.new`	`EA=Rs+(Ru<<#u);` `*EA = Nt.new;`

`memw(Ru<<#u2+#U6)=Nt.new`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Nt.new;`

`memw(Rx++#s4:2)=Nt.new`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Nt.new;`

`memw(Rx++#s4:2:circ(Mu))=Nt.new`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Nt.new;`

`memw(Rx++I:circ(Mu))=Nt.new`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<2,MuV);`
	`*EA = Nt.new;`

`memw(Rx++Mu)=Nt.new`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Nt.new;`

`memw(Rx++Mu:brev)=Nt.new`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Nt.new;`

`memw(gp+#u16:2)=Nt.new`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Nt.new;`

Class: NV (slots 0)

Notes

- Forms of this instruction which use a new-value operand produced in the packet must execute on slot 0.
- This instruction can execute only in slot 0, even though it is an ST instruction.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5										t3			
0	0	1	1	1	0	1	1	1	0	1	s	s	s	s	s	P	P	i	u	u	u	u	u	i	-	-	1	0	t	t	t	memw(Rs+Ru<<#u2)=Nt.new
ICLASS								Type								Parse					t3											
0	1	0	0	1	i	i	0	1	0	1	i	i	i	i	i	P	P	i	1	0	t	t	t	i	i	i	i	i	i	i	i	memw(gp+#u16:2)=Nt.new
ICLASS				Amode			Type			UN	s5					Parse					t3											
1	0	1	0	0	i	i	1	1	0	1	s	s	s	s	s	P	P	i	1	0	t	t	t	i	i	i	i	i	i	i	i	memw(Rs+#s11:2)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse		u1			t3											
1	0	1	0	1	0	0	1	1	0	1	x	x	x	x	x	P	P	u	1	0	t	t	t	0	-	-	-	-	-	1	-	memw(Rx++I:circ(Mu))=Nt.new
1	0	1	0	1	0	0	1	1	0	1	x	x	x	x	x	P	P	u	1	0	t	t	t	0	i	i	i	i	-	0	-	memw(Rx++#s4:2:circ(Mu))=Nt.new
ICLASS				Amode			Type			UN	e5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	e	e	e	e	e	P	P	0	1	0	t	t	t	1	-	l	l	l	l	l	l	memw(Re=#U6)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	x	x	x	x	x	P	P	0	1	0	t	t	t	0	i	i	i	i	-	0	-	memw(Rx++#s4:2)=Nt.new
ICLASS				Amode			Type			UN	u5					Parse					t3											
1	0	1	0	1	1	0	1	1	0	1	u	u	u	u	u	P	P	i	1	0	t	t	t	1	i	l	l	l	l	l	l	memw(Ru<<#u2+#U6)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse		u1			t3											
1	0	1	0	1	1	0	1	1	0	1	x	x	x	x	x	P	P	u	1	0	t	t	t	0	-	-	-	-	-	-	-	memw(Rx++Mu)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	x	x	x	x	x	P	P	u	1	0	t	t	t	0	-	-	-	-	-	-	-	memw(Rx++Mu:brev)=Nt.new

Field name	Description
ICLASS	Instruction Class
Type	Type
Parse	Packet/Loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t3	Field to encode register t
u1	Field to encode register u
u5	Field to encode register u
x5	Field to encode register x

Field name	Description
Amode	Amode
Type	Type
UN	Unsigned

Store new-value word conditionally

Store a 32-bit register in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv[.new]) memw(#u6)=Nt.new`	`if ([!]Pv[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=#u;`
	`  *EA = Nt[.new];`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new]) memw(Rs+#u6:2)=Nt.new`	`apply_extension(#u);`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+#u;`
	`  *EA = Nt[.new];`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new]) memw(Rs+Ru<<#u2)=Nt.new`	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+(Ru<<#u);`
	`  *EA = Nt[.new];`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new]) memw(Rx++#s4:2)=Nt.new`	`if ([!]Pv[.new][0]){`
	`  EA=Rx;`
	`  Rx=Rx+#s;`
	`  *EA = Nt[.new];`
	`} else {`
	`  NOP;`
	`};`

Class: NV (slots 0)

Notes

- Forms of this instruction which use a new-value operand produced in the packet must execute on slot 0.
- This instruction can execute only in slot 0, even though it is an ST instruction.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5									t3			
0	0	1	1	0	1	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	1	0	**t**	**t**	**t**	if (Pv) memw(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	1	0	**t**	**t**	**t**	if (!Pv) memw(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	1	0	**t**	**t**	**t**	if (Pv.new) memw(Rs+Ru<<#u2)=Nt.new
0	0	1	1	0	1	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	1	0	**t**	**t**	**t**	if (!Pv.new) memw(Rs+Ru<<#u2)=Nt.new
ICLASS					Sense	PredNew			Type		s5					Parse					t3											
0	1	0	0	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memw(Rs+#u6:2)=Nt.new
0	1	0	0	0	0	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memw(Rs+#u6:2)=Nt.new
0	1	0	0	0	1	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memw(Rs+#u6:2)=Nt.new
0	1	0	0	0	1	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	1	0	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memw(Rs+#u6:2)=Nt.new
ICLASS				Amode			Type			UN	x5					Parse					t3											
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memw(Rx++#s4:2)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memw(Rx++#s4:2)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memw(Rx++#s4:2)=Nt.new
1	0	1	0	1	0	1	1	1	0	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	1	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memw(Rx++#s4:2)=Nt.new
ICLASS				Amode			Type			UN						Parse					t3											
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	1	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memw(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	1	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memw(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memw(#u6)=Nt.new
1	0	1	0	1	1	1	1	1	0	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	1	0	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memw(#u6)=Nt.new

Field name	Description
ICLASS	Instruction Class
Type	Type
PredNew	PredNew

Field name	Description
Sense	Sense
Parse	Packet/Loop parse bits
s5	Field to encode register s
t3	Field to encode register t
u5	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

11.10 ST

The ST instruction class includes store instructions, used to store values in memory.

ST instructions are executable on slot 0 and slot 1.

Store doubleword

Store a 64-bit register pair in memory at the effective address.

Syntax	Behavior

`memd(Re=#U6)=Rtt`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Rtt;`
	`Re=#U;`

`memd(Rs+#s11:3)=Rtt`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Rtt;`

`memd(Rs+Ru<<#u2)=Rtt`	`EA=Rs+(Ru<<#u);` `*EA = Rtt;`

`memd(Ru<<#u2+#U6)=Rtt`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Rtt;`

`memd(Rx++#s4:3)=Rtt`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Rtt;`

`memd(Rx++#s4:3:circ(Mu))=Rtt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Rtt;`

`memd(Rx++I:circ(Mu))=Rtt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<3,MuV);`
	`*EA = Rtt;`

`memd(Rx++Mu)=Rtt`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Rtt;`

`memd(Rx++Mu:brev)=Rtt`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Rtt;`

`memd(gp+#u16:3)=Rtt`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Rtt;`

Class: ST (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5								t5						
0	0	1	1	1	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	-	-	**t**	**t**	**t**	**t**	**t**	memd(Rs+Ru<<#u2)=Rtt
ICLASS								Type								Parse			t5													
0	1	0	0	1	**i**	**i**	0	1	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memd(gp+#u16:3)=Rtt
ICLASS				Amode			Type			U N	s5					Parse			t5													
1	0	1	0	0	**i**	**i**	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memd(Rs+#s11:3)=Rtt
ICLASS				Amode			Type			U N	x5					Parse		u1	t5													
1	0	1	0	1	0	0	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	-	-	-	-	-	1	-	memd(Rx++I:circ(Mu))=Rtt

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	0	1	0	0	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memd(Rx++#s4:3:circ(Mu))=Rtt
ICLASS				Amode			Type			UN	e5					Parse			t5														
1	0	1	0	1	0	1	1	1	1	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**-**	**I**	**I**	**I**	**I**	**I**	**I**	memd(Re=#U6)=Rtt
ICLASS				Amode			Type			UN	x5					Parse			t5														
1	0	1	0	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memd(Rx++#s4:3)=Rtt
ICLASS				Amode			Type			UN	u5					Parse			t5														
1	0	1	0	1	1	0	1	1	1	0	**u**	**u**	**u**	**u**	**u**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	1	**i**	**I**	**I**	**I**	**I**	**I**	**I**	memd(Ru<<#u2+#U6)=Rtt
ICLASS				Amode			Type			UN	x5					Parse		u1	t5														
1	0	1	0	1	1	0	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memd(Rx++Mu)=Rtt
1	0	1	0	1	1	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memd(Rx++Mu:brev)=Rtt

Field name	Description
ICLASS	Instruction Class
Type	Type
Parse	Packet/Loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
u5	Field to encode register u
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store doubleword conditionally

Store a 64-bit register pair in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv[.new]) memd(#u6)=Rtt`	`if ([!]Pv[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=#u;`
	`  *EA = Rtt;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`memd(Rs+#u6:3)=Rtt`
	`apply_extension(#u);`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+#u;`
	`  *EA = Rtt;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`memd(Rs+Ru<<#u2)=Rtt`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+(Ru<<#u);`
	`  *EA = Rtt;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`memd(Rx++#s4:3)=Rtt`
	`if ([!]Pv[.new][0]){`
	`  EA=Rx;`
	`  Rx=Rx+#s;`
	`  *EA = Rtt;`
	`} else {`
	`  NOP;`
	`};`

Class: ST (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5								t5						
0	0	1	1	0	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	**t**	**t**	**t**	**t**	**t**	if (Pv) memd(Rs+Ru<<#u2)=Rtt
0	0	1	1	0	1	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	**t**	**t**	**t**	**t**	**t**	if (!Pv) memd(Rs+Ru<<#u2)=Rtt
0	0	1	1	0	1	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	**t**	**t**	**t**	**t**	**t**	if (Pv.new) memd(Rs+Ru<<#u2)=Rtt
0	0	1	1	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	**t**	**t**	**t**	**t**	**t**	if (!Pv.new) memd(Rs+Ru<<#u2)=Rtt
ICLASS					Sense	PredNew		Type			s5					Parse			t5													
0	1	0	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memd(Rs+#u6:3)=Rtt

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0	1	0	0	0	0	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memd(Rs+#u6:3)=Rtt
0	1	0	0	0	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memd(Rs+#u6:3)=Rtt
0	1	0	0	0	1	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memd(Rs+#u6:3)=Rtt
ICLASS				Amode			Type			U N	x5					Parse			t5													
1	0	1	0	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memd(Rx++#s4:3)=Rtt
1	0	1	0	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memd(Rx++#s4:3)=Rtt
1	0	1	0	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memd(Rx++#s4:3)=Rtt
1	0	1	0	1	0	1	1	1	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memd(Rx++#s4:3)=Rtt
ICLASS				Amode			Type			U N						Parse			t5													
1	0	1	0	1	1	1	1	1	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memd(#u6)=Rtt
1	0	1	0	1	1	1	1	1	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memd(#u6)=Rtt
1	0	1	0	1	1	1	1	1	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memd(#u6)=Rtt
1	0	1	0	1	1	1	1	1	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memd(#u6)=Rtt

Field name	Description
ICLASS	Instruction Class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
u5	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store byte

Store the least-significant byte in a source register at the effective address.

Syntax	Behavior

`memb(Re=#U6)=Rt`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Rt&0xff;`
	`Re=#U;`

`memb(Rs+#s11:0)=Rt`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Rt&0xff;`

`memb(Rs+#u6:0)=#S8`	`EA=Rs+#u;`
	`apply_extension(#S);`
	`*EA = #S;`

`memb(Rs+Ru<<#u2)=Rt`	`EA=Rs+(Ru<<#u);` `*EA = Rt&0xff;`

`memb(Ru<<#u2+#U6)=Rt`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Rt&0xff;`

`memb(Rx++#s4:0)=Rt`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Rt&0xff;`

`memb(Rx++#s4:0:circ(Mu))=Rt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Rt&0xff;`

`memb(Rx++I:circ(Mu))=Rt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<0,MuV);`
	`*EA = Rt&0xff;`

`memb(Rx++Mu)=Rt`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Rt&0xff;`

`memb(Rx++Mu:brev)=Rt`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Rt&0xff;`

`memb(gp+#u16:0)=Rt`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Rt&0xff;`

Class: ST (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5								t5						
0	0	1	1	1	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**-**	**-**	**t**	**t**	**t**	**t**	**t**	memb(Rs+Ru<<#u2)=Rt
ICLASS											s5					Parse																
0	0	1	1	1	1	0	**-**	**-**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**I**	**I**	**I**	**I**	memb(Rs+#u6:0)=#S8
ICLASS								Type								Parse			t5													
0	1	0	0	1	**i**	**i**	0	0	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memb(gp+#u16:0)=Rt

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse			t5													
1	0	1	0	0	i	i	1	0	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	i	i	i	i	i	i	i	memb(Rs+#s11:0)=Rt
ICLASS				Amode			Type			UN	x5					Parse		u1	t5													
1	0	1	0	1	0	0	1	0	0	0	x	x	x	x	x	P	P	u	t	t	t	t	t	0	-	-	-	-	-	1	-	memb(Rx++I:circ(Mu))=Rt
1	0	1	0	1	0	0	1	0	0	0	x	x	x	x	x	P	P	u	t	t	t	t	t	0	i	i	i	i	-	0	-	memb(Rx++#s4:0:circ(Mu))=Rt
ICLASS				Amode			Type			UN	e5					Parse			t5													
1	0	1	0	1	0	1	1	0	0	0	e	e	e	e	e	P	P	0	t	t	t	t	t	1	-	l	l	l	l	l	l	memb(Re=#U6)=Rt
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	0	0	0	x	x	x	x	x	P	P	0	t	t	t	t	t	0	i	i	i	i	-	0	-	memb(Rx++#s4:0)=Rt
ICLASS				Amode			Type			UN	u5					Parse			t5													
1	0	1	0	1	1	0	1	0	0	0	u	u	u	u	u	P	P	i	t	t	t	t	t	1	i	l	l	l	l	l	l	memb(Ru<<#u2+#U6)=Rt
ICLASS				Amode			Type			UN	x5					Parse		u1	t5													
1	0	1	0	1	1	0	1	0	0	0	x	x	x	x	x	P	P	u	t	t	t	t	t	0	-	-	-	-	-	-	-	memb(Rx++Mu)=Rt
1	0	1	0	1	1	1	1	0	0	0	x	x	x	x	x	P	P	u	t	t	t	t	t	0	-	-	-	-	-	-	-	memb(Rx++Mu:brev)=Rt

Field name	Description
ICLASS	Instruction Class
Type	Type
Parse	Packet/Loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
u5	Field to encode register u
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store byte conditionally

Store the least-significant byte in a source register at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv[.new]) memb(#u6)=Rt`	`if ([!]Pv[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=#u;`
	`  *EA = Rt&0xff;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`memb(Rs+#u6:0)=#S6`
	`if ([!]Pv[.new][0]){`
	`  EA=Rs+#u;`
	`  apply_extension(#S);`
	`  *EA = #S;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`memb(Rs+#u6:0)=Rt`
	`apply_extension(#u);`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+#u;`
	`  *EA = Rt&0xff;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`memb(Rs+Ru<<#u2)=Rt`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+(Ru<<#u);`
	`  *EA = Rt&0xff;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`memb(Rx++#s4:0)=Rt`
	`if ([!]Pv[.new][0]){`
	`  EA=Rx;`
	`  Rx=Rx+#s;`
	`  *EA = Rt&0xff;`
	`} else {`
	`  NOP;`
	`};`

Class: ST (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5								t5						
0	0	1	1	0	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	**t**	**t**	**t**	**t**	**t**	if (Pv) memb(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	**t**	**t**	**t**	**t**	**t**	if (!Pv) memb(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	**t**	**t**	**t**	**t**	**t**	if (Pv.new) memb(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**v**	**v**	**t**	**t**	**t**	**t**	**t**	if (!Pv.new) memb(Rs+Ru<<#u2)=Rt
ICLASS											s5					Parse																
0	0	1	1	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (Pv) memb(Rs+#u6:0)=#S6
0	0	1	1	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (!Pv) memb(Rs+#u6:0)=#S6
0	0	1	1	1	0	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (Pv.new) memb(Rs+#u6:0)=#S6
0	0	1	1	1	0	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (!Pv.new) memb(Rs+#u6:0)=#S6
ICLASS					Sense	PredNew		Type			s5					Parse			t5													
0	1	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memb(Rs+#u6:0)=Rt
0	1	0	0	0	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memb(Rs+#u6:0)=Rt
0	1	0	0	0	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memb(Rs+#u6:0)=Rt
0	1	0	0	0	1	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memb(Rs+#u6:0)=Rt
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memb(Rx++#s4:0)=Rt
1	0	1	0	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memb(Rx++#s4:0)=Rt
1	0	1	0	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memb(Rx++#s4:0)=Rt
1	0	1	0	1	0	1	1	0	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memb(Rx++#s4:0)=Rt
ICLASS				Amode			Type			UN						Parse			t5													
1	0	1	0	1	1	1	1	0	0	0	-	-	-	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memb(#u6)=Rt
1	0	1	0	1	1	1	1	0	0	0	-	-	-	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memb(#u6)=Rt
1	0	1	0	1	1	1	1	0	0	0	-	-	-	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memb(#u6)=Rt
1	0	1	0	1	1	1	1	0	0	0	-	-	-	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memb(#u6)=Rt

Field name	Description
ICLASS	Instruction Class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits

Field name	Description
`s5`	Field to encode register s
`t5`	Field to encode register t
`u5`	Field to encode register u
`v2`	Field to encode register v
`x5`	Field to encode register x
`Amode`	Amode
`Type`	Type
`UN`	Unsigned

Store halfword

Store the upper or lower 16-bits of a source register at the effective address.

Syntax	Behavior

`memh(Re=#U6)=Rt.H`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Rt.h[1];`
	`Re=#U;`

`memh(Re=#U6)=Rt`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Rt.h[0];`
	`Re=#U;`

`memh(Rs+#s11:1)=Rt.H`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Rt.h[1];`

`memh(Rs+#s11:1)=Rt`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Rt.h[0];`

`memh(Rs+#u6:1)=#S8`	`EA=Rs+#u;`
	`apply_extension(#S);`
	`*EA = #S;`

`memh(Rs+Ru<<#u2)=Rt.H`	`EA=Rs+(Ru<<#u);` `*EA = Rt.h[1];`
`memh(Rs+Ru<<#u2)=Rt`	`EA=Rs+(Ru<<#u);` `*EA = Rt.h[0];`

`memh(Ru<<#u2+#U6)=Rt.H`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Rt.h[1];`

`memh(Ru<<#u2+#U6)=Rt`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Rt.h[0];`

`memh(Rx++#s4:1)=Rt.H`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Rt.h[1];`

`memh(Rx++#s4:1)=Rt`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Rt.h[0];`

`memh(Rx++#s4:1:circ(Mu))=Rt.H`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Rt.h[1];`

`memh(Rx++#s4:1:circ(Mu))=Rt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Rt.h[0];`

`memh(Rx++I:circ(Mu))=Rt.H`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`*EA = Rt.h[1];`

`memh(Rx++I:circ(Mu))=Rt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<1,MuV);`
	`*EA = Rt.h[0];`

Syntax	Behavior

`memh(Rx++Mu)=Rt.H`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Rt.h[1];`

`memh(Rx++Mu)=Rt`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Rt.h[0];`

`memh(Rx++Mu:brev)=Rt.H`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Rt.h[1];`

`memh(Rx++Mu:brev)=Rt`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Rt.h[0];`

`memh(gp+#u16:1)=Rt.H`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Rt.h[1];`

`memh(gp+#u16:1)=Rt`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Rt.h[0];`

Class: ST (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5									t5					
0	0	1	1	1	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**-**	**-**	**t**	**t**	**t**	**t**	**t**	memh(Rs+Ru<<#u2)=Rt
0	0	1	1	1	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**-**	**-**	**t**	**t**	**t**	**t**	**t**	memh(Rs+Ru<<#u2)=Rt.H
ICLASS											s5					Parse																
0	0	1	1	1	1	0	**-**	**-**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**l**	**i**	**i**	**i**	**i**	**i**	**i**	**l**	**l**	**l**	**l**	**l**	**l**	**l**	memh(Rs+#u6:1)=#S8
ICLASS								Type								Parse			t5													
0	1	0	0	1	**i**	**i**	0	0	1	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memh(gp+#u16:1)=Rt
0	1	0	0	1	**i**	**i**	0	0	1	1	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memh(gp+#u16:1)=Rt.H
ICLASS				Amode			Type			U N	s5					Parse			t5													
1	0	1	0	0	**i**	**i**	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memh(Rs+#s11:1)=Rt
1	0	1	0	0	**i**	**i**	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memh(Rs+#s11:1)=Rt.H
ICLASS				Amode			Type			U N	x5					Parse	u1		t5													
1	0	1	0	1	0	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	1	**-**	memh(Rx++I:circ(Mu))=Rt
1	0	1	0	1	0	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memh(Rx++#s4:1:circ(Mu))=Rt
1	0	1	0	1	0	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	1	**-**	memh(Rx++I:circ(Mu))=Rt.H
1	0	1	0	1	0	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memh(Rx++#s4:1:circ(Mu))=Rt.H
ICLASS				Amode			Type			U N	e5					Parse			t5													
1	0	1	0	1	0	1	1	0	1	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**-**	**l**	**l**	**l**	**l**	**l**	**l**	memh(Re=#U6)=Rt
ICLASS				Amode			Type			U N	x5					Parse			t5													
1	0	1	0	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memh(Rx++#s4:1)=Rt
ICLASS				Amode			Type			U N	e5					Parse			t5													

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	1	0	1	0	1	1	0	1	1	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**-**	**I**	**I**	**I**	**I**	**I**	**I**	memh(Re=#U6)=Rt.H
ICLASS				Amode			Type			UN	x5					Parse			t5														
1	0	1	0	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memh(Rx++#s4:1)=Rt.H
ICLASS				Amode			Type			UN	u5					Parse			t5														
1	0	1	0	1	1	0	1	0	1	0	**u**	**u**	**u**	**u**	**u**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	1	**i**	**I**	**I**	**I**	**I**	**I**	**I**	memh(Ru<<#u2+#U6)=Rt
ICLASS				Amode			Type			UN	x5					Parse		u1	t5														
1	0	1	0	1	1	0	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memh(Rx++Mu)=Rt
ICLASS				Amode			Type			UN	u5					Parse			t5														
1	0	1	0	1	1	0	1	0	1	1	**u**	**u**	**u**	**u**	**u**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	1	**i**	**I**	**I**	**I**	**I**	**I**	**I**	memh(Ru<<#u2+#U6)=Rt.H
ICLASS				Amode			Type			UN	x5					Parse		u1	t5														
1	0	1	0	1	1	0	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memh(Rx++Mu)=Rt.H
1	0	1	0	1	1	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memh(Rx++Mu:brev)=Rt
1	0	1	0	1	1	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memh(Rx++Mu:brev)=Rt.H

Field name	Description
ICLASS	Instruction Class
Type	Type
Parse	Packet/Loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
u5	Field to encode register u
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store halfword conditionally

Store the upper or lower 16-bits of a source register in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv[.new]) memh(#u6)=Rt.H`	`if ([!]Pv[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=#u;`
	`  *EA = Rt.h[1];`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new]) memh(#u6)=Rt`	`if ([!]Pv[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=#u;`
	`  *EA = Rt.h[0];`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`memh(Rs+#u6:1)=#S6`
	`if ([!]Pv[.new][0]){`
	`  EA=Rs+#u;`
	`  apply_extension(#S);`
	`  *EA = #S;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`memh(Rs+#u6:1)=Rt.H`
	`apply_extension(#u);`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+#u;`
	`  *EA = Rt.h[1];`
	`} else {`
	`  NOP;`
	`};`

Syntax	Behavior
`if ([!]Pv[.new]) memh(Rs+#u6:1)=Rt`	`apply_extension(#u); if ([!]Pv[.new][0]) { EA=Rs+#u; *EA = Rt.h[0]; } else { NOP; };`
`if ([!]Pv[.new]) memh(Rs+Ru<<#u2)=Rt.H`	`if ([!]Pv[.new][0]) { EA=Rs+(Ru<<#u); *EA = Rt.h[1]; } else { NOP; };`
`if ([!]Pv[.new]) memh(Rs+Ru<<#u2)=Rt`	`if ([!]Pv[.new][0]) { EA=Rs+(Ru<<#u); *EA = Rt.h[0]; } else { NOP; };`
`if ([!]Pv[.new]) memh(Rx++#s4:1)=Rt.H`	`if ([!]Pv[.new][0]){ EA=Rx; Rx=Rx+#s; *EA = Rt.h[1]; } else { NOP; };`
`if ([!]Pv[.new]) memh(Rx++#s4:1)=Rt`	`if ([!]Pv[.new][0]){ EA=Rx; Rx=Rx+#s; *EA = Rt.h[0]; } else { NOP; };`

Class: ST (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5								t5					
0	0	1	1	0	1	0	0	0	1	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (Pv) memh(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	0	0	0	1	1	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (Pv) memh(Rs+Ru<<#u2)=Rt.H
0	0	1	1	0	1	0	1	0	1	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (!Pv) memh(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	0	1	0	1	1	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (!Pv) memh(Rs+Ru<<#u2)=Rt.H
0	0	1	1	0	1	1	0	0	1	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (Pv.new) memh(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	1	0	0	1	1	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (Pv.new) memh(Rs+Ru<<#u2)=Rt.H
0	0	1	1	0	1	1	1	0	1	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (!Pv.new) memh(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	1	1	0	1	1	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (!Pv.new) memh(Rs+Ru<<#u2)=Rt.H

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse																
0	0	1	1	1	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (Pv) memh(Rs+#u6:1)=#S6
0	0	1	1	1	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (!Pv) memh(Rs+#u6:1)=#S6
0	0	1	1	1	0	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (Pv.new) memh(Rs+#u6:1)=#S6
0	0	1	1	1	0	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**v**	**v**	**I**	**I**	**I**	**I**	**I**	if (!Pv.new) memh(Rs+#u6:1)=#S6
ICLASS					Sense	PredNew		Type			s5					Parse			t5												
0	1	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(Rs+#u6:1)=Rt
0	1	0	0	0	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(Rs+#u6:1)=Rt.H
0	1	0	0	0	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(Rs+#u6:1)=Rt
0	1	0	0	0	0	1	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(Rs+#u6:1)=Rt.H
0	1	0	0	0	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memh(Rs+#u6:1)=Rt
0	1	0	0	0	1	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv) memh(Rs+#u6:1)=Rt.H
0	1	0	0	0	1	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memh(Rs+#u6:1)=Rt
0	1	0	0	0	1	1	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	0	**v**	**v**	if (!Pv.new) memh(Rs+#u6:1)=Rt.H
ICLASS				Amode			Type			UN	x5					Parse			t5												
1	0	1	0	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(Rx++#s4:1)=Rt
1	0	1	0	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memh(Rx++#s4:1)=Rt
1	0	1	0	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(Rx++#s4:1)=Rt
1	0	1	0	1	0	1	1	0	1	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memh(Rx++#s4:1)=Rt
1	0	1	0	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(Rx++#s4:1)=Rt.H
1	0	1	0	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memh(Rx++#s4:1)=Rt.H
1	0	1	0	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(Rx++#s4:1)=Rt.H
1	0	1	0	1	0	1	1	0	1	1	**x**	**x**	**x**	**x**	**x**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memh(Rx++#s4:1)=Rt.H
ICLASS				Amode			Type			UN						Parse			t5												
1	0	1	0	1	1	1	1	0	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(#u6)=Rt
1	0	1	0	1	1	1	1	0	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memh(#u6)=Rt
1	0	1	0	1	1	1	1	0	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(#u6)=Rt
1	0	1	0	1	1	1	1	0	1	0	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memh(#u6)=Rt
1	0	1	0	1	1	1	1	0	1	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv) memh(#u6)=Rt.H
1	0	1	0	1	1	1	1	0	1	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv) memh(#u6)=Rt.H
1	0	1	0	1	1	1	1	0	1	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	0	**v**	**v**	if (Pv.new) memh(#u6)=Rt.H
1	0	1	0	1	1	1	1	0	1	1	**-**	**-**	**-**	**i**	**i**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	**i**	**i**	**i**	**i**	1	**v**	**v**	if (!Pv.new) memh(#u6)=Rt.H

Field name	Description
ICLASS	Instruction Class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
u5	Field to encode register u
v2	Field to encode register v
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store word

Store a 32-bit register in memory at the effective address.

Syntax	Behavior

`memw(Re=#U6)=Rt`	`apply_extension(#U);`
	`EA=#U;`
	`*EA = Rt;`
	`Re=#U;`

`memw(Rs+#s11:2)=Rt`	`apply_extension(#s);`
	`EA=Rs+#s;`
	`*EA = Rt;`

`memw(Rs+#u6:2)=#S8`	`EA=Rs+#u;`
	`apply_extension(#S);`
	`*EA = #S;`

`memw(Rs+Ru<<#u2)=Rt`	`EA=Rs+(Ru<<#u);` `*EA = Rt;`

`memw(Ru<<#u2+#U6)=Rt`	`apply_extension(#U);`
	`EA=#U+(Ru<<#u);`
	`*EA = Rt;`

`memw(Rx++#s4:2)=Rt`	`EA=Rx;`
	`Rx=Rx+#s;`
	`*EA = Rt;`

`memw(Rx++#s4:2:circ(Mu))=Rt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,#s,MuV);`
	`*EA = Rt;`

`memw(Rx++I:circ(Mu))=Rt`	`EA=Rx;`
	`Rx=Rx=circ_add(Rx,I<<2,MuV);`
	`*EA = Rt;`

`memw(Rx++Mu)=Rt`	`EA=Rx;`
	`Rx=Rx+MuV;`
	`*EA = Rt;`

`memw(Rx++Mu:brev)=Rt`	`EA=Rx.h[1] | brev(Rx.h[0]);`
	`Rx=Rx+MuV;`
	`*EA = Rt;`

`memw(gp+#u16:2)=Rt`	`apply_extension(#u);`
	`EA=(Constant_extended ? (0) : GP)+#u;`
	`*EA = Rt;`

Class: ST (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5								t5						
0	0	1	1	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**u**	**u**	**u**	**u**	**u**	**i**	**-**	**-**	**t**	**t**	**t**	**t**	**t**	memw(Rs+Ru<<#u2)=Rt
ICLASS											s5					Parse																
0	0	1	1	1	1	0	**-**	**-**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**I**	**i**	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**I**	**I**	**I**	**I**	memw(Rs+#u6:2)=#S8
ICLASS								Type								Parse			t5													
0	1	0	0	1	**i**	**i**	0	1	0	0	**i**	**i**	**i**	**i**	**i**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memw(gp+#u16:2)=Rt

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse			t5													
1	0	1	0	0	**i**	**i**	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	memw(Rs+#s11:2)=Rt
ICLASS				Amode			Type			UN	x5					Parse		u1	t5													
1	0	1	0	1	0	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	1	**-**	memw(Rx++I:circ(Mu))=Rt
1	0	1	0	1	0	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memw(Rx++#s4:2:circ(Mu))=Rt
ICLASS				Amode			Type			UN	e5					Parse			t5													
1	0	1	0	1	0	1	1	1	0	0	**e**	**e**	**e**	**e**	**e**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**-**	**I**	**I**	**I**	**I**	**I**	**I**	memw(Re=#U6)=Rt
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**i**	**i**	**i**	**i**	**-**	0	**-**	memw(Rx++#s4:2)=Rt
ICLASS				Amode			Type			UN	u5					Parse			t5													
1	0	1	0	1	1	0	1	1	0	0	**u**	**u**	**u**	**u**	**u**	**P**	**P**	**i**	**t**	**t**	**t**	**t**	**t**	1	**i**	**I**	**I**	**I**	**I**	**I**	**I**	memw(Ru<<#u2+#U6)=Rt
ICLASS				Amode			Type			UN	x5					Parse		u1	t5													
1	0	1	0	1	1	0	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memw(Rx++Mu)=Rt
1	0	1	0	1	1	1	1	1	0	0	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**u**	**t**	**t**	**t**	**t**	**t**	0	**-**	**-**	**-**	**-**	**-**	**-**	**-**	memw(Rx++Mu:brev)=Rt

Field name	Description
ICLASS	Instruction Class
Type	Type
Parse	Packet/Loop parse bits
e5	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
u1	Field to encode register u
u5	Field to encode register u
x5	Field to encode register x
Amode	Amode
Type	Type
UN	Unsigned

Store word conditionally

Store a 32-bit register in memory at the effective address.

This instruction is conditional based on a predicate value. If the predicate is true, the instruction is performed, otherwise it is treated as a NOP.

Syntax	Behavior

`if ([!]Pv[.new]) memw(#u6)=Rt`	`if ([!]Pv[.new][0]) {`
	`  apply_extension(#u);`
	`  EA=#u;`
	`  *EA = Rt;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`memw(Rs+#u6:2)=#S6`
	`if ([!]Pv[.new][0]){`
	`  EA=Rs+#u;`
	`  apply_extension(#S);`
	`  *EA = #S;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`memw(Rs+#u6:2)=Rt`
	`apply_extension(#u);`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+#u;`
	`  *EA = Rt;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`memw(Rs+Ru<<#u2)=Rt`
	`if ([!]Pv[.new][0]) {`
	`  EA=Rs+(Ru<<#u);`
	`  *EA = Rt;`
	`} else {`
	`  NOP;`
	`};`

`if ([!]Pv[.new])`
`memw(Rx++#s4:2)=Rt`
	`if ([!]Pv[.new][0]){`
	`  EA=Rx;`
	`  Rx=Rx+#s;`
	`  *EA = Rt;`
	`} else {`
	`  NOP;`
	`};`

Class: ST (slots 0,1)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse			u5									t5					
0	0	1	1	0	1	0	0	1	0	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (Pv) memw(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	0	1	1	0	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (!Pv) memw(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	1	0	1	0	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (Pv.new) memw(Rs+Ru<<#u2)=Rt
0	0	1	1	0	1	1	1	1	0	0	s	s	s	s	s	P	P	i	u	u	u	u	u	i	v	v	t	t	t	t	t	if (!Pv.new) memw(Rs+Ru<<#u2)=Rt
ICLASS											s5					Parse																
0	0	1	1	1	0	0	0	0	1	0	s	s	s	s	s	P	P	I	i	i	i	i	i	i	v	v	I	I	I	I	I	if (Pv) memw(Rs+#u6:2)=#S6
0	0	1	1	1	0	0	0	1	1	0	s	s	s	s	s	P	P	I	i	i	i	i	i	i	v	v	I	I	I	I	I	if (!Pv) memw(Rs+#u6:2)=#S6
0	0	1	1	1	0	0	1	0	1	0	s	s	s	s	s	P	P	I	i	i	i	i	i	i	v	v	I	I	I	I	I	if (Pv.new) memw(Rs+#u6:2)=#S6
0	0	1	1	1	0	0	1	1	1	0	s	s	s	s	s	P	P	I	i	i	i	i	i	i	v	v	I	I	I	I	I	if (!Pv.new) memw(Rs+#u6:2)=#S6
ICLASS					Sense	PredNew		Type			s5					Parse			t5													
0	1	0	0	0	0	0	0	1	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	i	i	i	i	0	v	v	if (Pv) memw(Rs+#u6:2)=Rt
0	1	0	0	0	0	1	0	1	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	i	i	i	i	0	v	v	if (Pv.new) memw(Rs+#u6:2)=Rt
0	1	0	0	0	1	0	0	1	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	i	i	i	i	0	v	v	if (!Pv) memw(Rs+#u6:2)=Rt
0	1	0	0	0	1	1	0	1	0	0	s	s	s	s	s	P	P	i	t	t	t	t	t	i	i	i	i	i	0	v	v	if (!Pv.new) memw(Rs+#u6:2)=Rt
ICLASS				Amode			Type			UN	x5					Parse			t5													
1	0	1	0	1	0	1	1	1	0	0	x	x	x	x	x	P	P	1	t	t	t	t	t	0	i	i	i	i	0	v	v	if (Pv) memw(Rx++#s4:2)=Rt
1	0	1	0	1	0	1	1	1	0	0	x	x	x	x	x	P	P	1	t	t	t	t	t	0	i	i	i	i	1	v	v	if (!Pv) memw(Rx++#s4:2)=Rt
1	0	1	0	1	0	1	1	1	0	0	x	x	x	x	x	P	P	1	t	t	t	t	t	1	i	i	i	i	0	v	v	if (Pv.new) memw(Rx++#s4:2)=Rt
1	0	1	0	1	0	1	1	1	0	0	x	x	x	x	x	P	P	1	t	t	t	t	t	1	i	i	i	i	1	v	v	if (!Pv.new) memw(Rx++#s4:2)=Rt
ICLASS				Amode			Type			UN						Parse			t5													
1	0	1	0	1	1	1	1	1	0	0	-	-	-	i	i	P	P	0	t	t	t	t	t	1	i	i	i	i	0	v	v	if (Pv) memw(#u6)=Rt
1	0	1	0	1	1	1	1	1	0	0	-	-	-	i	i	P	P	0	t	t	t	t	t	1	i	i	i	i	1	v	v	if (!Pv) memw(#u6)=Rt
1	0	1	0	1	1	1	1	1	0	0	-	-	-	i	i	P	P	1	t	t	t	t	t	1	i	i	i	i	0	v	v	if (Pv.new) memw(#u6)=Rt
1	0	1	0	1	1	1	1	1	0	0	-	-	-	i	i	P	P	1	t	t	t	t	t	1	i	i	i	i	1	v	v	if (!Pv.new) memw(#u6)=Rt

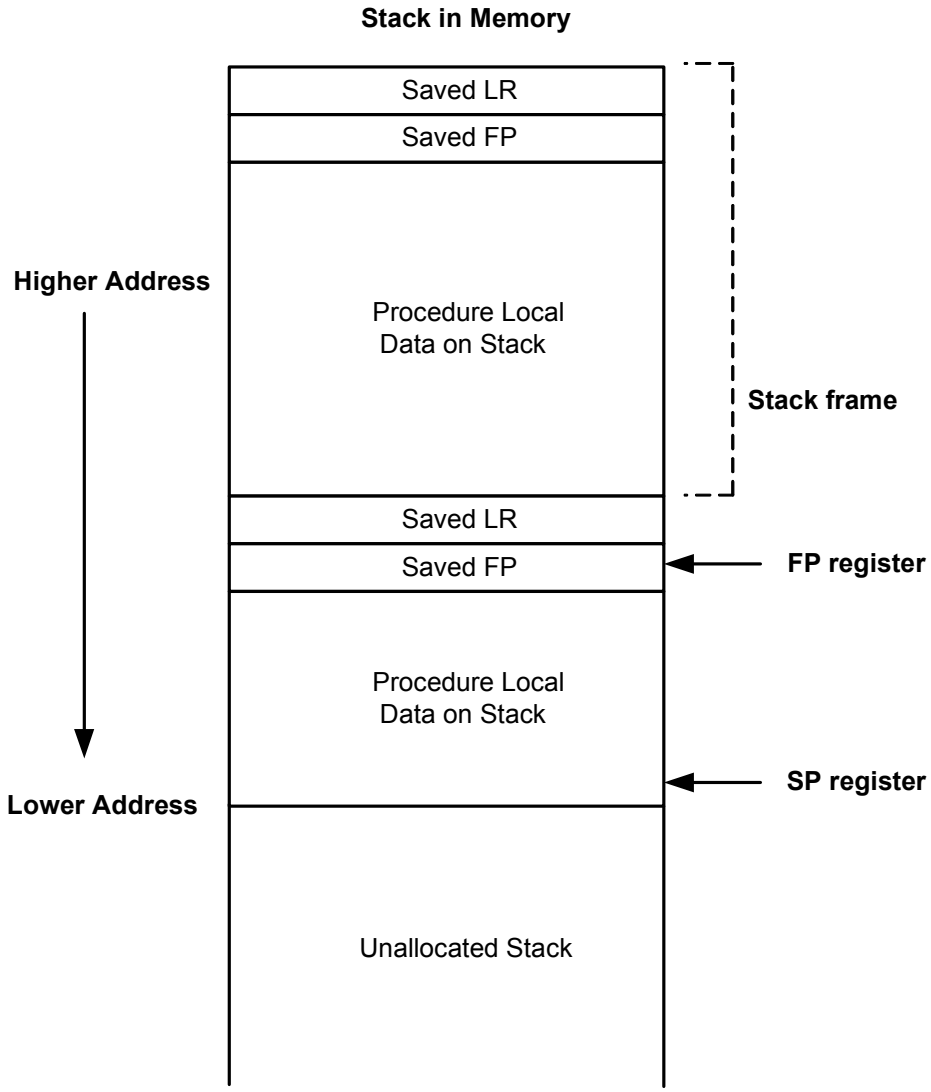
Field name	Description
ICLASS	Instruction Class
Type	Type
PredNew	PredNew
Sense	Sense
Parse	Packet/Loop parse bits

Field name	Description
`s5`	Field to encode register s
`t5`	Field to encode register t
`u5`	Field to encode register u
`v2`	Field to encode register v
`x5`	Field to encode register x
`Amode`	Amode
`Type`	Type
`UN`	Unsigned

Allocate stack frame

Allocate a stack frame on the call stack. This instruction first pushes LR and FP to the top of stack. It then subtracts an unsigned immediate from SP to allocate room for local variables. FP is set to the address of the old frame pointer on the stack.

The following figure shows the stack layout.



Syntax	Behavior

`allocframe(#u11:3)`	`EA=SP+-8;`
	`*EA = frame_scramble((LR << 32) | FP);`
	`FP=EA;`
	`frame_check_limit(EA-#u);`
	`SP=EA-#u;`

Class: ST (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN						Parse																	
1	0	1	0	0	0	0	0	1	0	0	1	1	1	0	1	**P**	**P**	0	0	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	allocframe(#u11:3)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
Amode	Amode
Type	Type
UN	Unsigned

11.11 SYSTEM

The SYSTEM instruction class includes instructions for managing system resources.

11.11.1 SYSTEM/USER

The SYSTEM/USER instruction subclass includes instructions which allow user access to system resources.

Load locked

Perform a word or double-word locked load. This returns the contents of the memory at address Rs and also reserves a lock reservation at that address. See the section on Atomic Operations for more information.

Syntax	Behavior
`Rd=memw_locked(Rs)`	`EA=Rs;` `Rd = *EA;`
`Rdd=memd_locked(Rs)`	`EA=Rs;` `Rdd = *EA;`

Class: SYSTEM (slots 0)

Notes

- This instruction may only be grouped with ALU32 or non floating-point XTYPE instructions.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse											d5					
1	0	0	1	0	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	-	-	-	-	-	-	0	**d**	**d**	**d**	**d**	**d**	Rd=memw_locked(Rs)
1	0	0	1	0	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	1	-	-	-	-	-	-	0	**d**	**d**	**d**	**d**	**d**	Rdd=memd_locked(Rs)

Field name	Description
ICLASS	Instruction Class
Amode	Amode
Type	Type
UN	Unsigned
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s

Store conditional

Perform a word or double-word conditional store operation. If the address reservation is held by this thread and there have been no intervening accesses to the memory location, then the store is performed and the predicate is set to true. Otherwise, the store is not performed and the predicate returns false. See the section on Atomic Operations for more information.

Syntax	Behavior

`memd_locked(Rs,Pd)=Rtt`	`EA=Rs;`
	`if (lock_valid) {`
	`  *EA = Rtt;`
	`  Pd = 0xff;`
	`  lock_valid = 0;`
	`} else {`
	`  Pd = 0;`
	`};`

`memw_locked(Rs,Pd)=Rt`	`EA=Rs;`
	`if (lock_valid) {`
	`  *EA = Rt;`
	`  Pd = 0xff;`
	`  lock_valid = 0;`
	`} else {`
	`  Pd = 0;`
	`};`

Class: SYSTEM (slots 0)

Notes

- This instruction may only be grouped with ALU32 or non floating-point XTYPE instructions.
- The predicate generated by this instruction can not be used as a .new predicate, nor can it be automatically ANDed with another predicate.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse			t5											d2		
1	0	1	0	0	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	memw_locked(Rs,Pd)=Rt
1	0	1	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	memd_locked(Rs,Pd)=Rtt

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Field name	Description
Amode	Amode
Type	Type
UN	Unsigned

Zero a cache line

Clear the specified 32-byte block in memory.

If the specified cache line exists in the L1 or L2 cache, it is cleared and marked dirty.

If the line does not exist, it is allocated for any cache operating in write-back mode.

If the cache is configured as write-through, the cleared cache line is written to the next level of memory.

If the cache is configured as write-back, no memory write occurs. Instead, the cleared cache line is marked as dirty.

This instruction is useful for efficiently handling write-only data by preallocating lines in the cache.

The address should be 32-byte aligned. If not, an unaligned error exception is raised.

If this instruction appears in a packet, then slot 1 must be A-type or empty.

Syntax	Behavior
`dczeroa(Rs)`	`EA=Rs;` `dcache_zero_addr(EA);`

Class: SYSTEM (slots 0)

Notes

- A packet containing this instruction must have slot 1 either empty or executing an ALU32 instruction.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse																	
1	0	1	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	-	-	-	-	-	-	-	-	dczeroa(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
Amode	Amode
Type	Type
UN	Unsigned

Memory barrier

Establish a memory barrier to ensure proper ordering between accesses before the barrier instruction and accesses after the barrier instruction. All accesses before the barrier will be globally observable before any access after the barrier can be observed.

The use of this instruction is system-dependent.

Syntax	Behavior
`barrier`	`memory_barrier;`

Class: SYSTEM (slots 0)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN						Parse																	
1	0	1	0	1	0	0	0	0	0	0	-	-	-	-	-	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	barrier

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
Amode	Amode
Type	Type
UN	Unsigned

Breakpoint

Cause the program to enter debug mode if enabled by ISDB. Execution control is handed to ISDB and the program will not proceed until directed by the debugger.

If ISDB is disabled, this instruction is treated as a NOP.

Syntax	Behavior
`brkpt`	`Enter debug mode;`

Class: SYSTEM (slot 3)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm											Parse																
0	1	1	0	1	1	0	0	0	0	1	-	-	-	-	-	**P**	**P**	-	-	-	-	-	-	0	0	0	-	-	-	-	-	brkpt

Field name	Description
`sm`	Supervisor mode only
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits

Data cache prefetch

Prefetch the data at address Rs + unsigned immediate.

This instruction is a hint to the memory system, and is handled in an implementation-dependent manner.

Syntax	Behavior
`dcfetch(Rs)`	`Assembler mapped to: "dcfetch(Rs+#0)"`
`dcfetch(Rs+#u11:3)`	`dcache_fetch(Rs+#u);`

Class: SYSTEM (slots 0)

Intrinsics

`dcfetch(Rs)`	`void Q6_dcfetch_A(Address a)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			U N	s5					Parse																	
1	0	0	1	0	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**-**	**-**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	dcfetch(Rs+#u11:3)

Field name	Description
`ICLASS`	Instruction Class
`Amode`	Amode
`Type`	Type
`UN`	Unsigned
`Parse`	Packet/Loop parse bits
`s5`	Field to encode register s

Data cache maintenance user operations

Perform maintenance operations on the data cache.

dccleaninva looks up the data cache at address Rs. If this address is in the cache and has dirty data, then the data is written back out to memory and the line is then invalidated.

dccleana looks up the data cache at address Rs. If this address is in the cache and has dirty data, then the data is written back out to memory.

dcinva looks up the data cache at address Rs. If this address is in the cache, then the line containing the data is invalidated.

If these instruction appears in a packet, then slot 1 must be A-type or empty.

In implementations that support L2 cache, these instructions operate on both L1 data and L2 caches.

Syntax	Behavior
`dccleana(Rs)`	`EA=Rs;` `dcache_clean_addr(EA);`
`dccleaninva(Rs)`	`EA=Rs;` `dcache_cleaninv_addr(EA);`
`dcinva(Rs)`	`EA=Rs;` `dcache_cleaninv_addr(EA);`

Class: SYSTEM (slots 0)

Notes

- A packet containing this instruction must have slot 1 either empty or executing an ALU32 instruction.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN	s5					Parse																	
1	0	1	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	dccleana(Rs)
1	0	1	0	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	dcinva(Rs)
1	0	1	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	-	-	dccleaninva(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
Amode	Amode
Type	Type
UN	Unsigned

Instruction cache maintenance user operations

Look up the address in Rs in the instruction cache. If the address is found, invalidate the corresponding cache line. If the user does not have proper permissions to the page which is to be invalidated, the instruction is converted to a NOP.

Syntax	Behavior
`icinva(Rs)`	`EA=Rs;` `icache_inv_addr(EA);`

Class: SYSTEM (slot 2)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS											s5					Parse																
0	1	0	1	0	1	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	0	-	-	-	-	-	-	-	-	-	-	-	icinva(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`s5`	Field to encode register s

Instruction synchronization

Ensure that all prior instructions have committed before continuing to the next instruction.

This instruction should be executed after the following events (when subsequent instructions need to observe the results of the event).

After modifying the TLB with a tlbw instruction

After modifying the SSR register

After modifying the SYSCFG register

After any instruction cache maintenance operation

After modifying the TID register

Syntax	Behavior
`isync`	`instruction_sync;`

Class: SYSTEM (slot 2)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse																
0	1	0	1	0	1	1	1	1	1	0	0	0	0	0	0	**P**	**P**	0	-	-	-	0	0	0	0	0	0	0	0	1	0	isync

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits

L2 cache prefetch

This instruction initiates background prefetching into the L2 cache.

Rs specifies the 32-bit virtual start address.

There are two forms of this instruction. In the first form, the dimensions of the area to prefetch are encoded in source register Rt as follows:

Rt[15:8] = Width of a fetch block in bytes.

Rt[7:0] = Height: the number of Width-sized blocks to fetch.

Rt[31:16] = Stride: an unsigned byte offset which is used to increment the pointer after each Width-sized block is fetched.

In the second form, the operands are encoded in register pair Rtt as follows:

Rtt[31:16] = Width of a fetch block in bytes.

Rtt[15:0] = Height: the number of Width-sized blocks to fetch.

Rtt[47:32] = Stride: an unsigned byte offset which is used to increment the pointer after each Width-sized block is fetched.

Rtt[48] = Direction. If clear, the prefetches should be performed in row major form meaning all cache lines in a row should be fetched before proceeding to the next row. If the bit is set, prefetch should be done in column major form meaning all cache lines in a column are fetched before proceeding to the next column.

The following figure shows two examples of using the l2fetch instruction. In the box prefetch, a 2-D range of memory is defined within a larger frame. The second example shows prefetch for a large linear area of memory which has size Lines * 128.

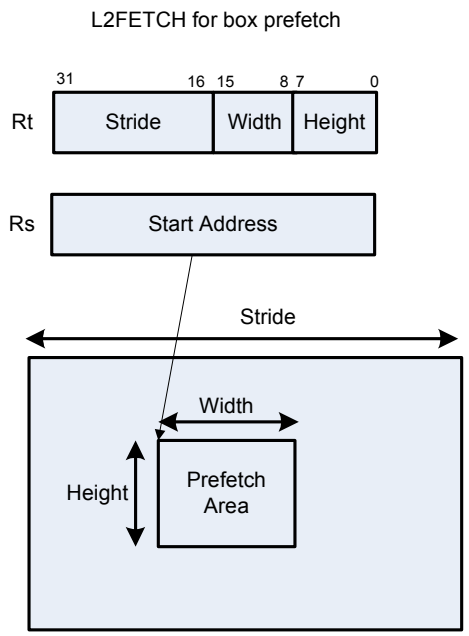
l2fetch is non-blocking. After the instruction is initiated, the program will continue on to the next instruction while the prefetching is performed in the background. l2fetch can be used to bring in either code or data to the L2 cache. If the lines of interest are already in the L2, no action is performed. If the lines are missing from the L2$, the hardware attempts to fetch them from the system memory.

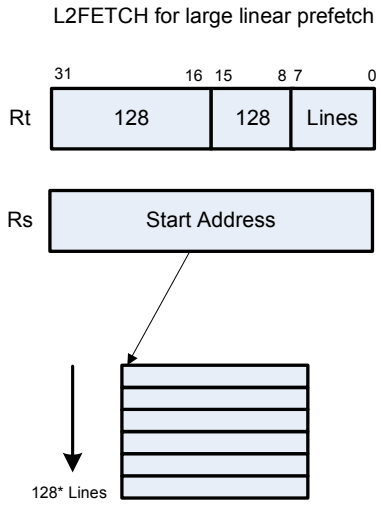
The hardware prefetch engine continues to request all lines in the programmed memory range. The prefetching hardware makes a best-effort to prefetch the requested data, and attempts to perform prefetching at a lower priority than demand fetches. This prevents prefetch from adding traffic while the system is under heavy load.

If a program initiates a new l2fetch while an older l2fetch operation is still pending, the new request is queued, up to 3 deep. If 3 l2fetch operations are already pending, the oldest request is dropped. During the time an L2 prefetch is active for a thread, the USR:PFA status bit is set to indicate that prefetches are in-progress. This bit can be used by the programmer to decide whether or not to start a new l2fetch before the previous one completes.

Executing an l2fetch instruction with any subfield programmed as zero cancels all pending prefetches by the calling thread.

The implementation is free to drop prefetches when needed.





Syntax	Behavior
`l2fetch(Rs,Rt)`	`l2fetch(Rs,INFO);`
`l2fetch(Rs,Rtt)`	`l2fetch(Rs,INFO);`

Class: SYSTEM (slots 0)

Notes

- This instruction may only be grouped with ALU32 or non floating-point XTYPE instructions.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			U N	s5					Parse			t5														
1	0	1	0	0	1	1	0	0	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	-	-	-	-	-	l2fetch(Rs,Rt)
1	0	1	0	0	1	1	0	1	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	-	-	-	-	-	-	-	-	l2fetch(Rs,Rtt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t

Field name	Description
Amode	Amode
Type	Type
UN	Unsigned

Pause

Pause execution for a specified period of time.

During the pause duration, the program enters a low-power state and will not fetch and execute instructions. The instruction provides a short immediate which indicates the pause duration. The program will pause for at most the number of cycles specified in the immediate plus 8. The minimum pause is 0 cycles, and the maximum pause is implementation-defined.

An interrupt to the program exits the paused state.

System events, such as HW or DMA completion, can trigger exits from pause mode.

An implementation is free to pause for durations shorter than (immediate+8), but not longer.

This instruction is useful for implementing user-level low-power synchronization operations, such as spin locks or wait-for-event signaling.

Syntax	Behavior
`pause(#u8)`	`Pause for #u cycles;`

Class: SYSTEM (slot 2)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse																
0	1	0	1	0	1	0	0	0	1	-	-	-	-	-	-	P	P	-	i	i	i	i	i	-	-	-	i	i	i	-	-	pause(#u8)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits

Memory thread synchronization

Synchronize memory.

All outstanding memory operations, including cached and uncached loads and stores, are completed before the processor continues to the next instruction. This ensures that certain memory operations are performed in the desired order (for example, when accessing I/O devices).

After performing a syncht operation, the processor ceases fetching and executing instructions from the program until all outstanding memory operations of that program are completed.

In multithreaded cores, syncht ensures that outstanding memory operations from all threads are complete before committing the syncht instruction.

The use of this instruction is system-dependent.

Syntax	Behavior
`syncht`	`memory_synch;`

Class: SYSTEM (slots 0)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				Amode			Type			UN						Parse																	
1	0	1	0	1	0	0	0	0	1	0	-	-	-	-	-	P	P	-	-	-	-	-	-	-	-	-	-	-	-	-	-	syncht

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
Amode	Amode
Type	Type
UN	Unsigned

Send value to ETM trace

Take the value of register Rs and emit it to the ETM trace.

The ETM block must be enabled, and the thread must have permissions to perform tracing. The contents of Rs are user-defined.

Syntax	Behavior
`trace(Rs)`	`Send value to ETM trace;`

Class: SYSTEM (slot 3)

Notes

- This instruction may only be grouped with ALU32 or non floating-point XTYPE instructions.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS					sm						s5					Parse																
0	1	1	0	0	0	1	0	0	1	0	s	s	s	s	s	P	P	-	-	-	-	-	-	-	-	-	-	-	-	-	-	trace(Rs)

Field name	Description
`sm`	Supervisor mode only
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`s5`	Field to encode register s

Trap

Cause a precise exception.

Executing a trap instruction sets the EX bit in SSR to 1, which disables interrupts and enables supervisor mode. The program then jumps to the vector location (either trap0 location or trap1 location). The instruction specifies an 8-bit immediate field. This field is copied into the system status register cause field.

Upon returning from the service routine with an rte instruction, execution resumes at the packet after the trap instruction.

These instructions are generally intended for user code to request services from the operating system. Two trap instructions are provided so the OS can optimize for fast service routines and slower service routines.

Syntax	Behavior
`trap0(#u8)`	`SSR.CAUSE = #u;` `TRAP "0";`
`trap1(#u8)`	`SSR.CAUSE = #u;` `TRAP "1";`

Class: SYSTEM (slot 2)

Notes

- This is a solo instruction. It must not be grouped with other instructions in a packet.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS																Parse																
0	1	0	1	0	1	0	0	0	0	-	-	-	-	-	-	P	P	-	i	i	i	i	i	-	-	-	i	i	i	-	-	trap0(#u8)
0	1	0	1	0	1	0	0	1	0	-	-	-	-	-	-	P	P	-	i	i	i	i	i	-	-	-	i	i	i	-	-	trap1(#u8)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits

11.12 XTYPE

The XTYPE instruction class includes instructions which perform most of the data processing done by the Hexagon processor.

XTYPE instructions are executable on slot 2 or slot 3.

11.12.1 XTYPE/ALU

The XTYPE/ALU instruction subclass includes instructions which perform arithmetic and logical operations.

Absolute value doubleword

Take the absolute value of the 64-bit source register and place it in the destination register.

Syntax	Behavior
`Rdd=abs(Rss)`	`Rdd = ABS(Rss);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=abs(Rss)`	`Word64 Q6_P_abs_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=abs(Rss)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Absolute value word

Take the absolute value of the source register and place it in the destination register.

The 32-bit absolute value is available with optional saturation. The single case of saturation is if the source register is equal to 0x8000_0000, then the destination saturates to 0x7fff_ffff.

Syntax	Behavior
`Rd=abs(Rs)[:sat]`	`Rd = [sat_32](ABS(sxt`$_{32->64}$`(Rs)));`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=abs(Rs)`	`Word32 Q6_R_abs_R(Word32 Rs)`
`Rd=abs(Rs):sat`	`Word32 Q6_R_abs_R_sat(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	1	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=abs(Rs)
1	0	0	0	1	1	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=abs(Rs):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Add and accumulate

Add Rs and Rt or a signed immediate, then add or subtract the resulting value. The result is saved in Rx.

Syntax	Behavior
`Rd=add(Rs,add(Ru,#s6))`	`Rd = Rs + Ru + apply_extension(#s);`
`Rd=add(Rs,sub(#s6,Ru))`	`Rd = Rs - Ru + apply_extension(#s);`
`Rx+=add(Rs,#s8)`	`apply_extension(#s);` `Rx=Rx + Rs + #s;`
`Rx+=add(Rs,Rt)`	`Rx=Rx + Rs + Rt;`
`Rx-=add(Rs,#s8)`	`apply_extension(#s);` `Rx=Rx - (Rs + #s);`
`Rx-=add(Rs,Rt)`	`Rx=Rx - (Rs + Rt);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=add(Rs,add(Ru,#s6))`	`Word32 Q6_R_add_add_RRI(Word32 Rs, Word32 Ru, Word32 Is6)`
`Rd=add(Rs,sub(#s6,Ru))`	`Word32 Q6_R_add_sub_RIR(Word32 Rs, Word32 Is6, Word32 Ru)`
`Rx+=add(Rs,#s8)`	`Word32 Q6_R_addacc_RI(Word32 Rx, Word32 Rs, Word32 Is8)`
`Rx+=add(Rs,Rt)`	`Word32 Q6_R_addacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=add(Rs,#s8)`	`Word32 Q6_R_addnac_RI(Word32 Rx, Word32 Rs, Word32 Is8)`
`Rx-=add(Rs,Rt)`	`Word32 Q6_R_addnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			d5								u5					
1	1	0	1	1	0	1	1	0	**i**	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**d**	**d**	**d**	**d**	**d**	**i**	**i**	**i**	**u**	**u**	**u**	**u**	**u**	Rd=add(Rs,add(Ru,#s6))
1	1	0	1	1	0	1	1	1	**i**	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**d**	**d**	**d**	**d**	**d**	**i**	**i**	**i**	**u**	**u**	**u**	**u**	**u**	Rd=add(Rs,sub(#s6,Ru))
ICLASS				RegType				MajOp			s5					Parse								MinOp			x5					
1	1	1	0	0	0	1	0	0	**-**	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	Rx+=add(Rs,#s8)
1	1	1	0	0	0	1	0	1	**-**	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	Rx-=add(Rs,#s8)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx+=add(Rs,Rt)
1	1	1	0	1	1	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx-=add(Rs,Rt)

Field name	Description
RegType	Register Type
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u5	Field to encode register u
x5	Field to encode register x

Add doublewords

The first form of this instruction adds two 32-bit registers. If the result overflows 32 bits, then the result is saturated to 0x7FFF_FFFF for a positive result, or 0x8000_0000 for a negative result. Note that 32-bit non-saturating register add is a ALU32-class instruction and can be executed on any slot.

The second instruction form sign-extends a 32-bit register Rt to 64-bits and performs a 64-bit add with Rss. The result is stored in Rdd.

The third instruction form adds 64-bit registers Rss and Rtt and places the result in Rdd.

The final instruction form adds two 64-bit registers Rss and Rtt. If the result overflows 64 bits, then it is saturated to 0x7fff_ffff_ffff_ffff for a positive result, or 0x8000_0000_0000_0000 for a negative result.

Syntax	Behavior
`Rd=add(Rs,Rt):sat:deprecated`	`Rd=sat_32(Rs+Rt);`
`Rdd=add(Rs,Rtt)`	`if ("Rs & 1") { Assembler mapped to: "Rdd=add(Rss,Rtt):raw:hi"; } else { Assembler mapped to: "Rdd=add(Rss,Rtt):raw:lo"; };`
`Rdd=add(Rss,Rtt)`	`Rdd=Rss+Rtt;`
`Rdd=add(Rss,Rtt):raw:hi`	$Rdd=Rtt+sxt_{32->64}(Rss.w[1])$;
`Rdd=add(Rss,Rtt):raw:lo`	$Rdd=Rtt+sxt_{32->64}(Rss.w[0])$;
`Rdd=add(Rss,Rtt):sat`	`Rdd=sat64(Rss+Rtt);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=add(Rs,Rtt)`	`Word64 Q6_P_add_RP(Word32 Rs, Word64 Rtt)`
`Rdd=add(Rss,Rtt)`	`Word64 Q6_P_add_PP(Word64 Rss, Word64 Rtt)`
`Rdd=add(Rss,Rtt):sat`	`Word64 Q6_P_add_PP_sat(Word64 Rss, Word64 Rtt)`

Encoding

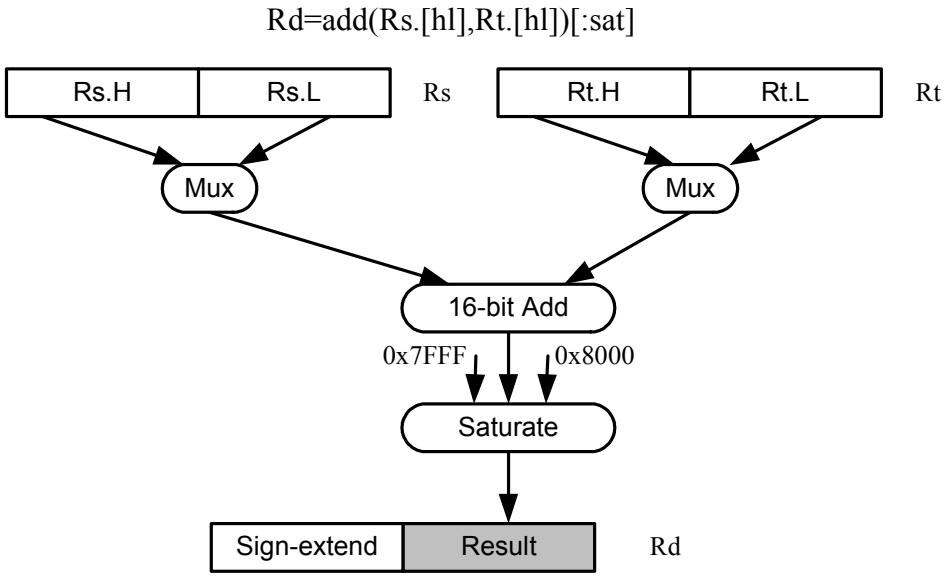
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=add(Rss,Rtt)
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=add(Rss,Rtt):sat
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=add(Rss,Rtt):raw:lo
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=add(Rss,Rtt):raw:hi
1	1	0	1	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	-	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rs,Rt):sat:deprecated

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Add halfword

Perform a 16-bit add with optional saturation, and place the result in either the upper or lower half of a register. If the result goes in the upper half, then the sources can be any high or low halfword of Rs and Rt. The lower 16 bits of the result are zeroed.

If the result is to be placed in the lower 16 bits of Rd, then the Rs source can be either high or low, but the other source must be the low halfword of Rt. In this case, the upper halfword of Rd is the sign-extension of the low halfword.



Syntax	Behavior
`Rd=add(Rt.L,Rs.[HL])[:sat]`	`Rd=[sat_16](Rt.h[0]+Rs.h[01]);`
`Rd=add(Rt.[HL],Rs.[HL])[:sat]:<<16`	`Rd=([sat_16](Rt.h[01]+Rs.h[01]))<<16;`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=add(Rt.H,Rs.H):<<16`	`Word32 Q6_R_add_RhRh_s16(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.H,Rs.H):sat:<<16`	`Word32 Q6_R_add_RhRh_sat_s16(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.H,Rs.L):<<16`	`Word32 Q6_R_add_RhRl_s16(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.H,Rs.L):sat:<<16`	`Word32 Q6_R_add_RhRl_sat_s16(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.H)`	`Word32 Q6_R_add_RlRh(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.H):<<16`	`Word32 Q6_R_add_RlRh_s16(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.H):sat`	`Word32 Q6_R_add_RlRh_sat(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.H):sat:<<16`	`Word32 Q6_R_add_RlRh_sat_s16(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.L)`	`Word32 Q6_R_add_RlRl(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.L):<<16`	`Word32 Q6_R_add_RlRl_s16(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.L):sat`	`Word32 Q6_R_add_RlRl_sat(Word32 Rt, Word32 Rs)`
`Rd=add(Rt.L,Rs.L):sat:<<16`	`Word32 Q6_R_add_RlRl_sat_s16(Word32 Rt, Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.L)
1	1	0	1	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.H)
1	1	0	1	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.L):sat
1	1	0	1	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.H):sat
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.L):<<16
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.H):<<16
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.H,Rs.L):<<16
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.H,Rs.H):<<16
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.L):sat:<<16
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.L,Rs.H):sat:<<16
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.H,Rs.L):sat:<<16
1	1	0	1	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=add(Rt.H,Rs.H):sat:<<16

Field name	Description
RegType	Register Type
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Add or subtract doublewords with carry

Add or subtract with carry. Predicate register Px is used as an extra input and output.

For adds, the LSB of the predicate is added to the sum of the two input pairs.

For subtracts, the predicate is considered a not-borrow. The LSB of the predicate is added to the first source register and the logical complement of the second argument.

The carry-out from the sum is saved in predicate Px.

These instructions allow efficient addition or subtraction of numbers larger than 64 bits.

Syntax	Behavior
`Rdd=add(Rss,Rtt,Px):carry`	`Rdd = Rss + Rtt + Px[0];` `Px = carry_from_add(Rss,Rtt,Px[0]) ? 0xff : 0x00;`
`Rdd=sub(Rss,Rtt,Px):carry`	`Rdd = Rss + ~Rtt + Px[0];` `Px = carry_from_add(Rss,~Rtt,Px[0]) ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Notes

- The predicate generated by this instruction can not be used as a .new predicate, nor can it be automatically ANDed with another predicate.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5						x2		d5						
1	1	0	0	0	0	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	**x**	**x**	**d**	**d**	**d**	**d**	**d**	Rdd=add(Rss,Rtt,Px):carry
1	1	0	0	0	0	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	**x**	**x**	**d**	**d**	**d**	**d**	**d**	Rdd=sub(Rss,Rtt,Px):carry

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x2	Field to encode register x
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Logical doublewords

Perform bitwise logical AND, OR, XOR, and NOT operations.

The source and destination registers are 64-bit.

For 32-bit logical operations, see the ALU32 logical instructions.

Syntax	Behavior
`Rdd=and(Rss,Rtt)`	`Rdd=Rss&Rtt;`
`Rdd=and(Rtt,~Rss)`	`Rdd = (Rtt & ~Rss);`
`Rdd=not(Rss)`	`Rdd=~Rss;`
`Rdd=or(Rss,Rtt)`	`Rdd=Rss|Rtt;`
`Rdd=or(Rtt,~Rss)`	`Rdd = (Rtt | ~Rss);`
`Rdd=xor(Rss,Rtt)`	`Rdd=Rss^Rtt;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=and(Rss,Rtt)`	`Word64 Q6_P_and_PP(Word64 Rss, Word64 Rtt)`
`Rdd=and(Rtt,~Rss)`	`Word64 Q6_P_and_PnP(Word64 Rtt, Word64 Rss)`
`Rdd=not(Rss)`	`Word64 Q6_P_not_P(Word64 Rss)`
`Rdd=or(Rss,Rtt)`	`Word64 Q6_P_or_PP(Word64 Rss, Word64 Rtt)`
`Rdd=or(Rtt,~Rss)`	`Word64 Q6_P_or_PnP(Word64 Rtt, Word64 Rss)`
`Rdd=xor(Rss,Rtt)`	`Word64 Q6_P_xor_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=not(Rss)
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=and(Rss,Rtt)
1	1	0	1	0	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=and(Rtt,~Rss)
1	1	0	1	0	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=or(Rss,Rtt)
1	1	0	1	0	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=or(Rtt,~Rss)
1	1	0	1	0	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=xor(Rss,Rtt)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits

Field name	Description
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Logical-logical doublewords

Perform a logical operation of the two source operands, then perform a second logical operation of the result with the destination register Rxx.

The source and destination registers are 64-bit.

Syntax	Behavior
`Rxx^=xor(Rss,Rtt)`	`Rxx^=Rss^Rtt;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rxx^=xor(Rss,Rtt)`	`Word64 Q6_P_xorxacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			x5					
1	1	0	0	1	0	1	0	1	0	-	s	s	s	s	s	P	P	0	t	t	t	t	t	0	0	0	x	x	x	x	x	Rxx^=xor(Rss,Rtt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Logical-logical words

Perform a logical operation of the two source operands, then perform a second logical operation of the result with the destination register Rx.

The source and destination registers are 32-bit.

Syntax	Behavior
`Rx=or(Ru,and(Rx,#s10))`	`Rx = Ru | (Rx & apply_extension(#s));`
`Rx[&|^]=and(Rs,Rt)`	`Rx [|&^]= (Rs [|&^] Rt);`
`Rx[&|^]=and(Rs,~Rt)`	`Rx [|&^]= (Rs [|&^] ~Rt);`
`Rx[&|^]=or(Rs,Rt)`	`Rx [|&^]= (Rs [|&^] Rt);`
`Rx[&|^]=xor(Rs,Rt)`	`Rx[|&^]=Rs[|&^]Rt;`
`Rx|=and(Rs,#s10)`	`Rx = Rx | (Rs & apply_extension(#s));`
`Rx|=or(Rs,#s10)`	`Rx = Rx | (Rs | apply_extension(#s));`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx&=and(Rs,Rt)`	`Word32 Q6_R_andand_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx&=and(Rs,~Rt)`	`Word32 Q6_R_andand_RnR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx&=or(Rs,Rt)`	`Word32 Q6_R_orand_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx&=xor(Rs,Rt)`	`Word32 Q6_R_xorand_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx=or(Ru,and(Rx,#s10))`	`Word32 Q6_R_or_and_RRI(Word32 Ru, Word32 Rx, Word32 Is10)`
`Rx^=and(Rs,Rt)`	`Word32 Q6_R_andxacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx^=and(Rs,~Rt)`	`Word32 Q6_R_andxacc_RnR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx^=or(Rs,Rt)`	`Word32 Q6_R_orxacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx^=xor(Rs,Rt)`	`Word32 Q6_R_xorxacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx|=and(Rs,#s10)`	`Word32 Q6_R_andor_RI(Word32 Rx, Word32 Rs, Word32 Is10)`

`Rx|=and(Rs,Rt)`	`Word32 Q6_R_andor_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx|=and(Rs,~Rt)`	`Word32 Q6_R_andor_RnR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx|=or(Rs,#s10)`	`Word32 Q6_R_oror_RI(Word32 Rx, Word32 Rs, Word32 Is10)`
`Rx|=or(Rs,Rt)`	`Word32 Q6_R_oror_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx|=xor(Rs,Rt)`	`Word32 Q6_R_xoror_RR(Word32 Rx, Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse											x5					
1	1	0	1	1	0	1	0	0	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	Rx|=and(Rs,#s10)
ICLASS				RegType							x5					Parse											u5					
1	1	0	1	1	0	1	0	0	1	**i**	**x**	**x**	**x**	**x**	**x**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**u**	**u**	**u**	**u**	**u**	Rx=or(Ru,and(Rx,#s10))
ICLASS				RegType							s5					Parse											x5					
1	1	0	1	1	0	1	0	1	0	**i**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	Rx|=or(Rs,#s10)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx|=and(Rs,~Rt)
1	1	1	0	1	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx&=and(Rs,~Rt)
1	1	1	0	1	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx^=and(Rs,~Rt)
1	1	1	0	1	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx&=and(Rs,Rt)
1	1	1	0	1	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx&=or(Rs,Rt)
1	1	1	0	1	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx&=xor(Rs,Rt)
1	1	1	0	1	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx|=and(Rs,Rt)
1	1	1	0	1	1	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx^=xor(Rs,Rt)
1	1	1	0	1	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx|=or(Rs,Rt)
1	1	1	0	1	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx|=xor(Rs,Rt)
1	1	1	0	1	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx^=and(Rs,Rt)
1	1	1	0	1	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx^=or(Rs,Rt)

Field name	Description
`RegType`	Register Type
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`s5`	Field to encode register s
`t5`	Field to encode register t
`u5`	Field to encode register u
`x5`	Field to encode register x

Maximum words

Select either the signed or unsigned maximum of two source registers and place in a destination register Rdd.

Syntax	Behavior
`Rd=max(Rs,Rt)`	`Rd = max(Rs,Rt);`
`Rd=maxu(Rs,Rt)`	`Rd = max(Rs.uw[0],Rt.uw[0]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=max(Rs,Rt)`	`Word32 Q6_R_max_RR(Word32 Rs, Word32 Rt)`
`Rd=maxu(Rs,Rt)`	`UWord32 Q6_R_maxu_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	1	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	-	-	**d**	**d**	**d**	**d**	**d**	Rd=max(Rs,Rt)
1	1	0	1	0	1	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	-	-	**d**	**d**	**d**	**d**	**d**	Rd=maxu(Rs,Rt)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Maximum doublewords

Select either the signed or unsigned maximum of two 64-bit source registers and place in a destination register.

Syntax	Behavior
`Rdd=max(Rss,Rtt)`	`Rdd = max(Rss,Rtt);`
`Rdd=maxu(Rss,Rtt)`	`Rdd = max(Rss.u64,Rtt.u64);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=max(Rss,Rtt)`	`Word64 Q6_P_max_PP(Word64 Rss, Word64 Rtt)`
`Rdd=maxu(Rss,Rtt)`	`UWord64 Q6_P_maxu_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=max(Rss,Rtt)
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=maxu(Rss,Rtt)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Minimum words

Select either the signed or unsigned minimum of two source registers and place in destination register Rd.

Syntax	Behavior
`Rd=min(Rt,Rs)`	`Rd = min(Rt,Rs);`
`Rd=minu(Rt,Rs)`	`Rd = min(Rt.uw[0],Rs.uw[0]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=min(Rt,Rs)`	`Word32 Q6_R_min_RR(Word32 Rt, Word32 Rs)`
`Rd=minu(Rt,Rs)`	`UWord32 Q6_R_minu_RR(Word32 Rt, Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	-	-	**d**	**d**	**d**	**d**	**d**	Rd=min(Rt,Rs)
1	1	0	1	0	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	-	-	**d**	**d**	**d**	**d**	**d**	Rd=minu(Rt,Rs)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Minimum doublewords

Select either the signed or unsigned minimum of two 64-bit source registers and place in the destination register Rdd.

Syntax	Behavior
`Rdd=min(Rtt,Rss)`	`Rdd = min(Rtt,Rss);`
`Rdd=minu(Rtt,Rss)`	`Rdd = min(Rtt.u64,Rss.u64);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=min(Rtt,Rss)`	`Word64 Q6_P_min_PP(Word64 Rtt, Word64 Rss)`
`Rdd=minu(Rtt,Rss)`	`UWord64 Q6_P_minu_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=min(Rtt,Rss)
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=minu(Rtt,Rss)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Modulo wrap

Wrap the Rs value into the modulo range from 0 to Rt.

If Rs is greater than or equal to Rt, wrap it to the bottom of the range by subtracting Rt.

If Rs is less than zero, wrap it to the top of the range by adding Rt.

Otherwise, when Rs fits within the range, no adjustment is necessary. The result is returned in register Rd.

Syntax	Behavior

`Rd=modwrap(Rs,Rt)`	`if (Rs < 0) {`
	`  Rd = Rs + Rt.uw[0];`
	`} else if (Rs.uw[0] >= Rt.uw[0]) {`
	`  Rd = Rs - Rt.uw[0];`
	`} else {`
	`  Rd = Rs;`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=modwrap(Rs,Rt)`	`Word32 Q6_R_modwrap_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=modwrap(Rs,Rt)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Negate

The first form of this instruction performs a negate on a 32-bit register with saturation. If the input is 0x80000000, the result is saturated to 0x7fffffff. Note that the non-saturating 32-bit register negate is a ALU32-class instruction and can be executed on any slot.

The second form of this instruction negates a 64-bit source register and place the result in destination Rdd.

Syntax	Behavior
`Rd=neg(Rs):sat`	`Rd = sat_32(-Rs.s64);`
`Rdd=neg(Rss)`	`Rdd = -Rss;`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=neg(Rs):sat`	`Word32 Q6_R_neg_R_sat(Word32 Rs)`
`Rdd=neg(Rss)`	`Word64 Q6_P_neg_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=neg(Rss)
1	0	0	0	1	1	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=neg(Rs):sat

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Round

Perform either arithmetic (.5 is rounded up) or convergent (.5 is rounded towards even) rounding to any bit location.

Arithmetic rounding has optional saturation. In this version, the result is saturated to a 32-bit number after adding the rounding constant. After the rounding and saturation have been performed, the final result is right shifted using a sign-extending shift.

Syntax	Behavior
`Rd=cround(Rs,#u5)`	`Rd = (#u==0)?Rs:convround(Rs,2**(#u-1))>>#u;`
`Rd=cround(Rs,Rt)`	`Rd = (zxt5->32(Rt)==0)?Rs:convround(Rs,2**(zxt5->32(Rt)-1))>>zxt5->32(Rt);`
`Rd=round(Rs,#u5)[:sat]`	`Rd = ([sat_32]((#u==0)?(Rs):round(Rs,2**(#u-1))))>>#u;`
`Rd=round(Rs,Rt)[:sat]`	`Rd = ([sat_32]((zxt5->32(Rt)==0)?(Rs):round(Rs,2**(zxt5->32(Rt)-1))))>>zxt5->32(Rt);`
`Rd=round(Rss):sat`	`tmp=sat64(Rss+0x080000000ULL);` `Rd = tmp.w[1];`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=cround(Rs,#u5)`	`Word32 Q6_R_cround_RI(Word32 Rs, Word32 Iu5)`
`Rd=cround(Rs,Rt)`	`Word32 Q6_R_cround_RR(Word32 Rs, Word32 Rt)`
`Rd=round(Rs,#u5)`	`Word32 Q6_R_round_RI(Word32 Rs, Word32 Iu5)`
`Rd=round(Rs,#u5):sat`	`Word32 Q6_R_round_RI_sat(Word32 Rs, Word32 Iu5)`
`Rd=round(Rs,Rt)`	`Word32 Q6_R_round_RR(Word32 Rs, Word32 Rt)`
`Rd=round(Rs,Rt):sat`	`Word32 Q6_R_round_RR_sat(Word32 Rs, Word32 Rt)`
`Rd=round(Rss):sat`	`Word32 Q6_R_round_P_sat(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**-**	**-**	**-**	**-**	**-**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=round(Rss):sat
1	0	0	0	1	1	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	**-**	**d**	**d**	**d**	**d**	**d**	Rd=cround(Rs,#u5)
1	0	0	0	1	1	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	1	0	**-**	**d**	**d**	**d**	**d**	**d**	Rd=round(Rs,#u5)
1	0	0	0	1	1	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	1	1	**-**	**d**	**d**	**d**	**d**	**d**	Rd=round(Rs,#u5):sat
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	1	1	0	1	1	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	0	**-**	**d**	**d**	**d**	**d**	**d**	Rd=cround(Rs,Rt)
1	1	0	0	0	1	1	0	1	1	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	0	**-**	**d**	**d**	**d**	**d**	**d**	Rd=round(Rs,Rt)
1	1	0	0	0	1	1	0	1	1	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	1	**-**	**d**	**d**	**d**	**d**	**d**	Rd=round(Rs,Rt):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
MinOp	Minor Opcode
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type
RegType	Register Type

Subtract doublewords

Subtract the 64-bit register Rss from register Rtt.

Syntax	Behavior
`Rd=sub(Rt,Rs):sat:deprecated`	`Rd=sat_32(Rt - Rs);`
`Rdd=sub(Rtt,Rss)`	`Rdd=Rtt-Rss;`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=sub(Rtt,Rss)`	`Word64 Q6_P_sub_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=sub(Rtt,Rss)
1	1	0	1	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	-	-	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt,Rs):sat:deprecated

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Subtract and accumulate words

Subtract Rs from Rt, then add the resulting value with Rx. The result is saved in Rx.

Syntax	Behavior
`Rx+=sub(Rt,Rs)`	`Rx=Rx + Rt - Rs;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx+=sub(Rt,Rs)`	`Word32 Q6_R_subacc_RR(Word32 Rx, Word32 Rt, Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx+=sub(Rt,Rs)

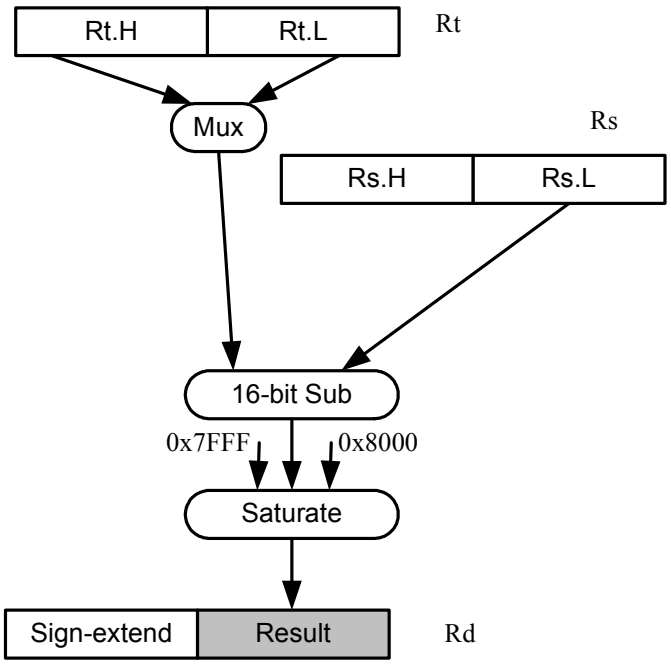
Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Subtract halfword

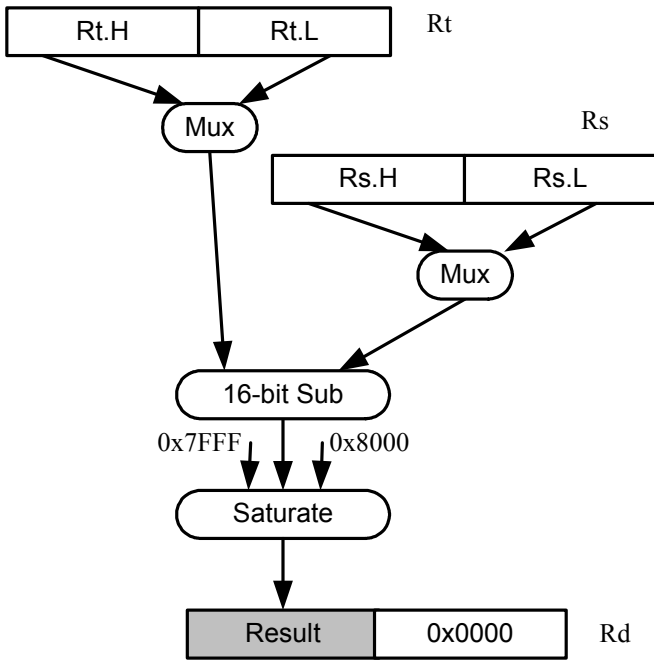
Perform a 16-bit subtract with optional saturation and place the result in either the upper or lower half of a register. If the result goes in the upper half, then the sources can be any high or low halfword of Rs and Rt. The lower 16 bits of the result are zeroed.

If the result is to be placed in the lower 16 bits of Rd, then the Rs source can be either high or low, but the other source must be the low halfword of Rt. In this case, the upper halfword of Rd is the sign-extension of the low halfword.

Rd=sub(Rt.[hl],Rs.l)[:sat]



Rd=sub(Rt.[hl],Rs.[hl])[:sat]:<<16



Syntax	Behavior
`Rd=sub(Rt.L,Rs.[HL])[:sat]`	`Rd=[sat_16](Rt.h[0]-Rs.h[01]);`
`Rd=sub(Rt.[HL],Rs.[HL])[:sat]:<<16`	`Rd=([sat_16](Rt.h[01]-Rs.h[01]))<<16;`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=sub(Rt.H,Rs.H):<<16`	`Word32 Q6_R_sub_RhRh_s16(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.H,Rs.H):sat:<<16`	`Word32 Q6_R_sub_RhRh_sat_s16(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.H,Rs.L):<<16`	`Word32 Q6_R_sub_RhRl_s16(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.H,Rs.L):sat:<<16`	`Word32 Q6_R_sub_RhRl_sat_s16(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.H)`	`Word32 Q6_R_sub_RlRh(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.H):<<16`	`Word32 Q6_R_sub_RlRh_s16(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.H):sat`	`Word32 Q6_R_sub_RlRh_sat(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.H):sat:<<16`	`Word32 Q6_R_sub_RlRh_sat_s16(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.L)`	`Word32 Q6_R_sub_RlRl(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.L):<<16`	`Word32 Q6_R_sub_RlRl_s16(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.L):sat`	`Word32 Q6_R_sub_RlRl_sat(Word32 Rt, Word32 Rs)`
`Rd=sub(Rt.L,Rs.L):sat:<<16`	`Word32 Q6_R_sub_RlRl_sat_s16(Word32 Rt, Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.L)
1	1	0	1	0	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.H)
1	1	0	1	0	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.L):sat
1	1	0	1	0	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.H):sat
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.L):<<16
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.H):<<16
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.H,Rs.L):<<16
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.H,Rs.H):<<16
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.L):sat:<<16
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.L,Rs.H):sat:<<16
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.H,Rs.L):sat:<<16
1	1	0	1	0	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=sub(Rt.H,Rs.H):sat:<<16

Field name	Description
RegType	Register Type
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Sign extend word to doubleword

Sign-extend a 32-bit word to a 64-bit doubleword.

Syntax	Behavior
`Rdd=sxtw(Rs)`	`Rdd = sxt32->64(Rs);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=sxtw(Rs)`	`Word64 Q6_P_sxtw_R(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	0	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=sxtw(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Vector absolute value halfwords

Take the absolute value of each of the four halfwords in the 64-bit source vector Rss. Place the result in Rdd.

Saturation is optionally available.

Syntax	Behavior

`Rdd=vabsh(Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=ABS(Rss.h[i]);`
	`};`

`Rdd=vabsh(Rss):sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=sat_16(ABS(Rss.h[i]));`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vabsh(Rss)`	`Word64 Q6_P_vabsh_P(Word64 Rss)`
`Rdd=vabsh(Rss):sat`	`Word64 Q6_P_vabsh_P_sat(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsh(Rss)
1	0	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vabsh(Rss):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Vector absolute value words

Take the absolute value of each of the two words in the 64-bit source vector Rss. Place the result in Rdd.

Saturation is optionally available.

Syntax	Behavior

`Rdd=vabsw(Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=ABS(Rss.w[i]);`
	`};`

`Rdd=vabsw(Rss):sat`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=sat_32(ABS(Rss.w[i]));`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vabsw(Rss)`	`Word64 Q6_P_vabsw_P(Word64 Rss)`
`Rdd=vabsw(Rss):sat`	`Word64 Q6_P_vabsw_P_sat(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsw(Rss)
1	0	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vabsw(Rss):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Vector absolute difference bytes

For each element in the source vector Rss, subtract the corresponding element in source vector Rtt. Take the absolute value of the results, and store into Rdd.

Syntax	Behavior

`Rdd=vabsdiffb(Rtt,Rss)`	`for (i=0;i<8;i++) {`
	`  Rdd.b[i]=ABS(Rtt.b[i] - Rss.b[i]);`
	`};`

`Rdd=vabsdiffub(Rtt,Rss)`	`for (i=0;i<8;i++) {`
	`  Rdd.b[i]=ABS(Rtt.ub[i] - Rss.ub[i]);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vabsdiffb(Rtt,Rss)`	`Word64 Q6_P_vabsdiffb_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vabsdiffub(Rtt,Rss)`	`Word64 Q6_P_vabsdiffub_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			d5					
1	1	1	0	1	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsdiffub(Rtt,Rss)
1	1	1	0	1	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsdiffb(Rtt,Rss)

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

NOTE `vabsdiffb` and `vabsdiffub` are supported only in processor versions V61 or greater.

Vector absolute difference halfwords

For each element in the source vector Rss, subtract the corresponding element in source vector Rtt. Take the absolute value of the results, and store into Rdd.

Syntax	Behavior

`Rdd=vabsdiffh(Rtt,Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=ABS(Rtt.h[i] - Rss.h[i]);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vabsdiffh(Rtt,Rss)`	`Word64 Q6_P_vabsdiffh_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsdiffh(Rtt,Rss)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector absolute difference words

For each element in the source vector Rss, subtract the corresponding element in source vector Rtt. Take the absolute value of the results, and store into Rdd.

Syntax	Behavior

`Rdd=vabsdiffw(Rtt,Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=ABS(Rtt.w[i] - Rss.w[i]);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vabsdiffw(Rtt,Rss)`	`Word64 Q6_P_vabsdiffw_PP(Word64 Rtt, Word64 Rss)`

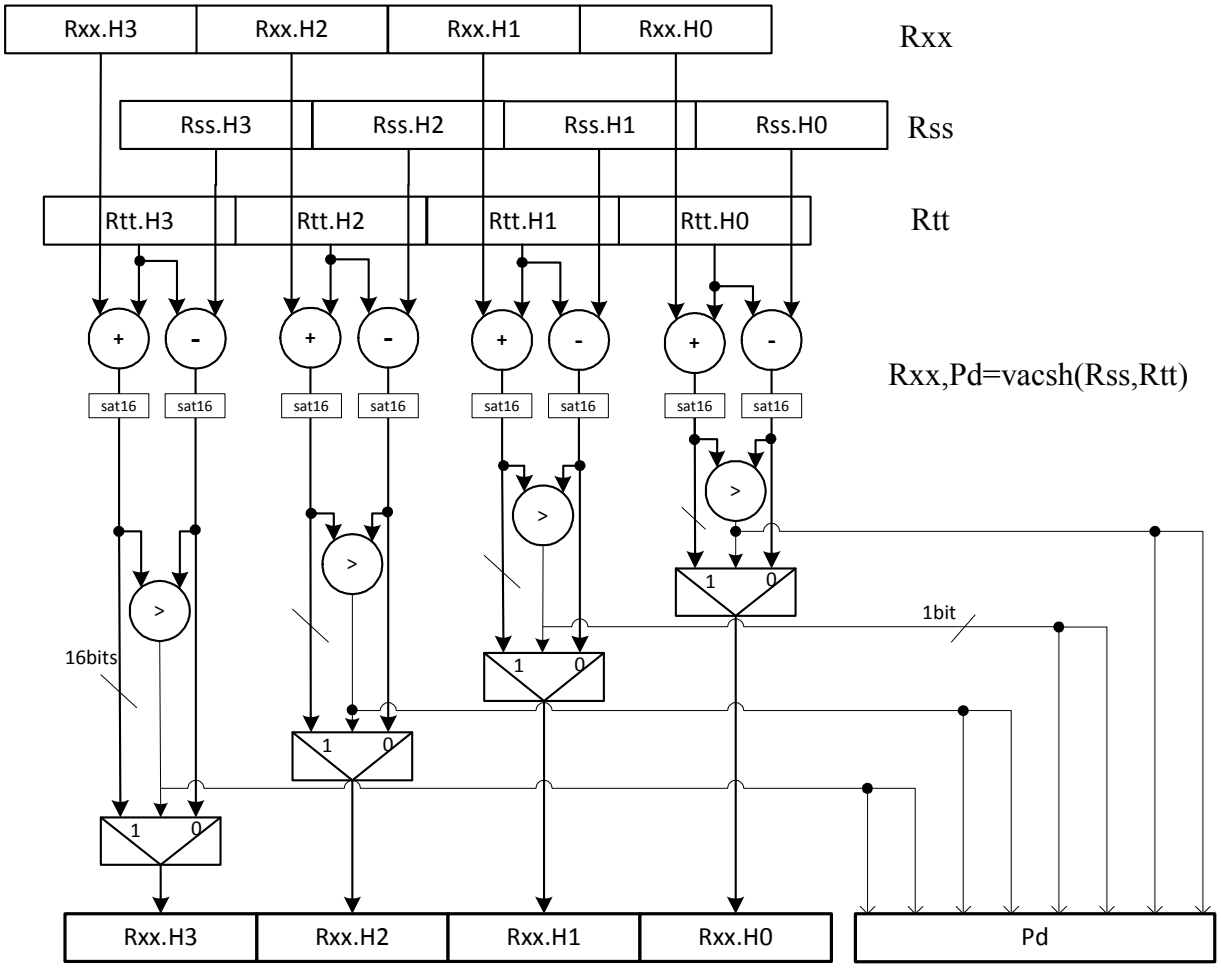
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vabsdiffw(Rtt,Rss)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector add compare and select maximum bytes

Add each byte element in Rxx and Rtt, and compare the resulting sums with the corresponding differences between Rss and Rtt. Store the maximum value of each compare in Rxx, and set the corresponding bits in a predicate destination to '1' if the compare result is greater, '0' if not. Each sum and difference is saturated to 8 bits before the compare, and the compare operation is a signed byte compare.

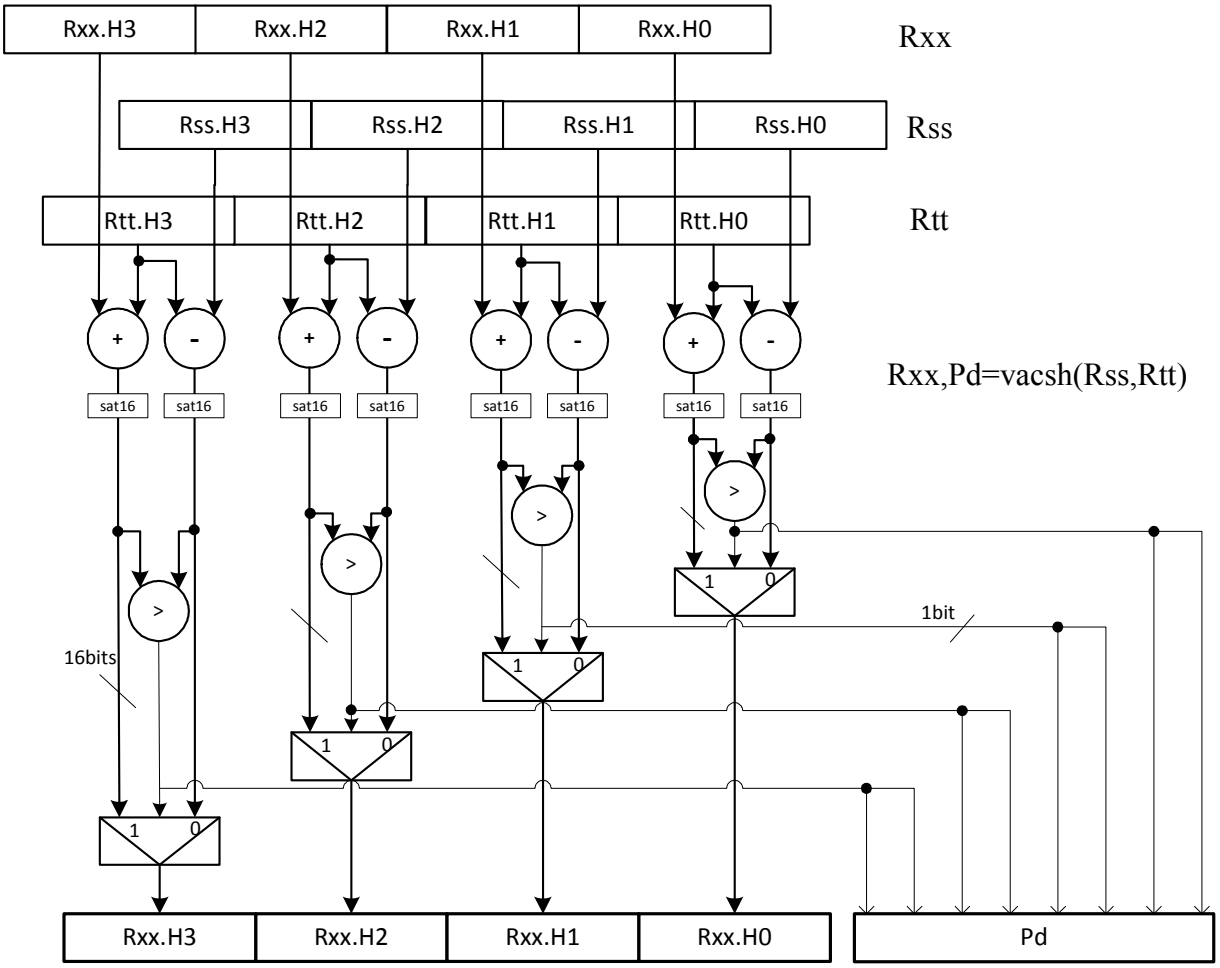


Syntax	Behavior

Class: N/A

Vector add compare and select maximum halfwords

Add each halfword element in Rxx and Rtt, and compare the resulting sums with the corresponding differences between Rss and Rtt. Store the maximum value of each compare in Rxx, and set the corresponding bits in a predicate destination to '11' if the compare result is greater, '00' if not. Each sum and difference is saturated to 16 bits before the compare, and the compare operation is a signed halfword compare.



Syntax

```
Rxx,Pe=vacsh(Rss,Rtt)
```

Behavior

```
for (i = 0; i < 4; i++) {
    xv = (int) Rxx.h[i];
    sv = (int) Rss.h[i];
    tv = (int) Rtt.h[i];
    xv = xv + tv;
    sv = sv - tv;
    Pe.i*2 = (xv > sv);
    Pe.i*2+1 = (xv > sv);
    Rxx.h[i]=sat_16(max(xv,sv));
};
```

Class: XTYPE (slots 2,3)

Notes

- The predicate generated by this instruction can not be used as a .new predicate, nor can it be automatically ANDed with another predicate.
- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						e2		x5						
1	1	1	0	1	0	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**e**	**e**	**x**	**x**	**x**	**x**	**x**	Rxx,Pe=vacsh(Rss,Rtt)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
e2	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector add halfwords

Add each of the four halfwords in 64-bit vector Rss to the corresponding halfword in vector Rtt.

Optionally saturate each 16-bit addition to either a signed or unsigned 16-bit value. Applying saturation to the vaddh instruction clamps the result to the signed range 0x8000 to 0x7fff, whereas applying saturation to the vadduh instruction ensures that the unsigned result falls within the range 0 to 0xffff. When saturation is not needed, the vaddh form should be used.

For the 32-bit version of this vector operation, see the ALU32 instructions.

Syntax	Behavior

`Rdd=vaddh(Rss,Rtt)[:sat]`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=[sat_16](Rss.h[i]+Rtt.h[i]);`
	`};`

`Rdd=vadduh(Rss,Rtt):sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=usat_16(Rss.uh[i]+Rtt.uh[i]);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vaddh(Rss,Rtt)`	`Word64 Q6_P_vaddh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vaddh(Rss,Rtt):sat`	`Word64 Q6_P_vaddh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vadduh(Rss,Rtt):sat`	`Word64 Q6_P_vadduh_PP_sat(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vaddh(Rss,Rtt)
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vaddh(Rss,Rtt):sat
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vadduh(Rss,Rtt):sat

Field name	Description
RegType	Register Type
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector add halfwords with saturate and pack to unsigned bytes

Add the four 16-bit halfwords of Rss to the four 16-bit halfwords of Rtt. The results are saturated to unsigned 8-bits and packed in destination register Rd.

Syntax	Behavior

`Rd=vaddhub(Rss,Rtt):sat`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=usat_8(Rss.h[i]+Rtt.h[i]);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vaddhub(Rss,Rtt):sat`	`Word32 Q6_R_vaddhub_PP_sat(Word64 Rss, Word64 Rtt)`

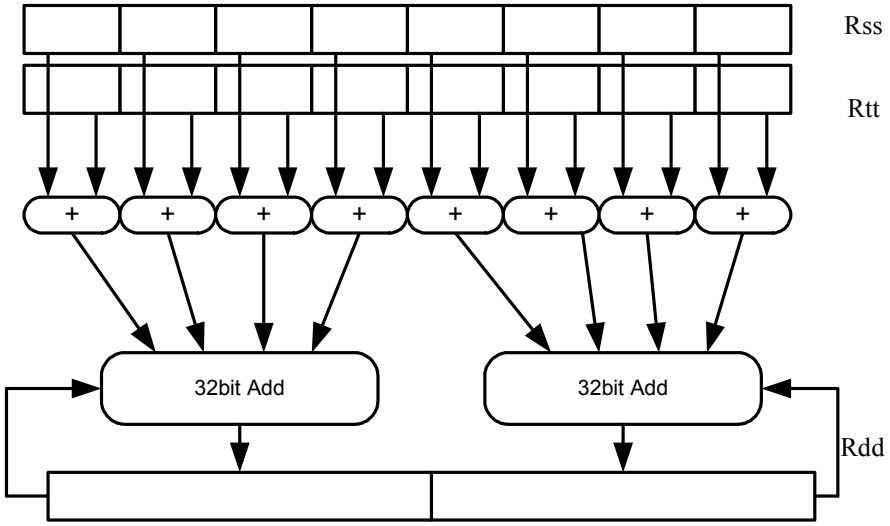
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	0	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	1	d	d	d	d	d	Rd=vaddhub(Rss,Rtt):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Vector reduce add unsigned bytes

For each byte in the source vector Rss, add the corresponding byte in the source vector Rtt. Add the four upper intermediate results and optionally the upper word of the destination. Add the four lower results and optionally the lower word of the destination.



Syntax	Behavior

`Rdd=vraddub(Rss,Rtt)`	`Rdd = 0;`
	`for (i=0;i<4;i++) {`
	`  Rdd.w[0]=(Rdd.w[0] + (Rss.ub[i]+Rtt.ub[i]));`
	`};`
	`for (i=4;i<8;i++) {`
	`  Rdd.w[1]=(Rdd.w[1] + (Rss.ub[i]+Rtt.ub[i]));`
	`};`

`Rxx+=vraddub(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Rxx.w[0]=(Rxx.w[0] + (Rss.ub[i]+Rtt.ub[i]));`
	`};`
	`for (i = 4; i < 8; i++) {`
	`  Rxx.w[1]=(Rxx.w[1] + (Rss.ub[i]+Rtt.ub[i]));`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vraddub(Rss,Rtt)`	`Word64 Q6_P_vraddub_PP(Word64 Rss, Word64 Rtt)`
`Rxx+=vraddub(Rss,Rtt)`	`Word64 Q6_P_vraddubacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

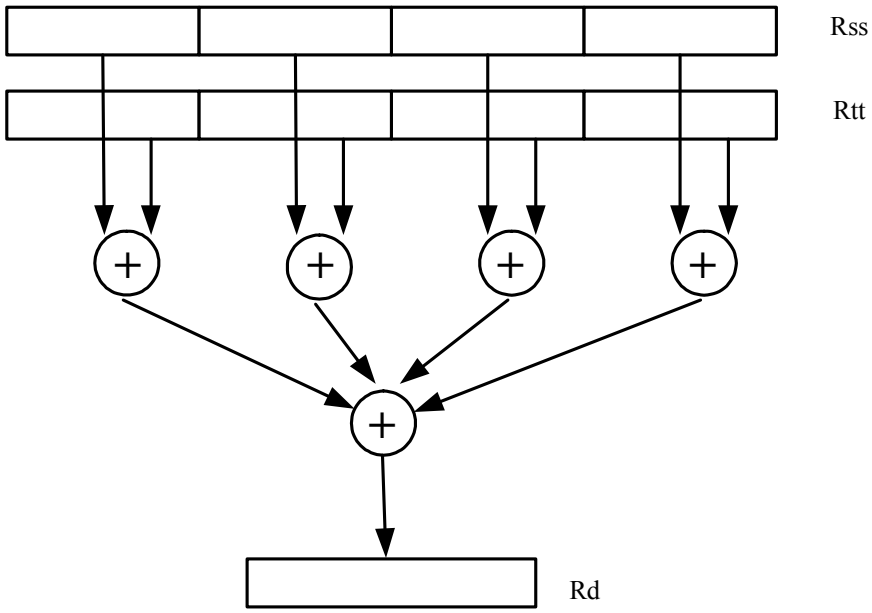
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			d5					
1	1	1	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vraddub(Rss,Rtt)
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			x5					
1	1	1	0	1	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vraddub(Rss,Rtt)

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Vector reduce add halfwords

For each halfword in the source vector Rss, add the corresponding halfword in the source vector Rtt. Add these intermediate results together, and place the result in Rd.



Syntax	Behavior

`Rd=vraddh(Rss,Rtt)`	`Rd = 0;`
	`for (i=0;i<4;i++) {`
	`  Rd += (Rss.h[i]+Rtt.h[i]);`
	`};`

`Rd=vradduh(Rss,Rtt)`	`Rd = 0;`
	`for (i=0;i<4;i++) {`
	`  Rd += (Rss.uh[i]+Rtt.uh[i]);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=vraddh(Rss,Rtt)`	`Word32 Q6_R_vraddh_PP(Word64 Rss, Word64 Rtt)`
`Rd=vradduh(Rss,Rtt)`	`Word32 Q6_R_vradduh_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	1	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	-	0	1	**d**	**d**	**d**	**d**	**d**	Rd=vradduh(Rss,Rtt)
1	1	1	0	1	0	0	1	0	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=vraddh(Rss,Rtt)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector add bytes

Add each of the eight bytes in 64-bit vector Rss to the corresponding byte in vector Rtt. Optionally, saturate each 8-bit addition to an unsigned value between 0 and 255. The eight results are stored in destination register Rdd.

Syntax	Behavior
`Rdd=vaddb(Rss,Rtt)`	`Assembler mapped to: "Rdd=vaddub(Rss,Rtt)"`

`Rdd=vaddub(Rss,Rtt)[:sat]`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=[usat_8](Rss.ub[i]+Rtt.ub[i]);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vaddb(Rss,Rtt)`	`Word64 Q6_P_vaddb_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vaddub(Rss,Rtt)`	`Word64 Q6_P_vaddub_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vaddub(Rss,Rtt):sat`	`Word64 Q6_P_vaddub_PP_sat(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vaddub(Rss,Rtt)
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vaddub(Rss,Rtt):sat

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector add words

Add each of the two words in 64-bit vector Rss to the corresponding word in vector Rtt. Optionally, saturate each 32-bit addition to a signed value between 0x80000000 and 0x7fffffff. The two word results are stored in destination register Rdd.

Syntax	Behavior

`Rdd=vaddw(Rss,Rtt)[:sat]`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=[sat_32](Rss.w[i]+Rtt.w[i]);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vaddw(Rss,Rtt)`	`Word64 Q6_P_vaddw_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vaddw(Rss,Rtt):sat`	`Word64 Q6_P_vaddw_PP_sat(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vaddw(Rss,Rtt)
1	1	0	1	0	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vaddw(Rss,Rtt):sat

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector average halfwords

Average each of the four halfwords in the 64-bit source vector Rss with the corresponding halfword in Rtt. The average operation performed on each halfword adds the two halfwords and shifts the result right by 1 bit. Unsigned average uses a logical right shift (shift in 0), whereas signed average uses an arithmetic right shift (shift in the sign bit). If the round option is used, then a 0x0001 is also added to each result before shifting. This operation does not overflow. In the case that a summation (before right shift by 1) causes an overflow of 32 bits, the value shifted in is the most-significant carry out.

The signed average and negative average halfwords is available with optional convergent rounding. In convergent rounding, if the two LSBs after the addition/subtraction are 11, then a rounding constant of 1 is added, otherwise a 0 is added. This result is then shifted right by one bit. Convergent rounding accumulates less error than arithmetic rounding.

Syntax	Behavior

`Rdd=vavgh(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.h[i]+Rtt.h[i])>>1;`
	`};`

`Rdd=vavgh(Rss,Rtt):crnd`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=convround(Rss.h[i]+Rtt.h[i])>>1;`
	`};`

`Rdd=vavgh(Rss,Rtt):rnd`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.h[i]+Rtt.h[i]+1)>>1;`
	`};`

`Rdd=vavguh(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.uh[i]+Rtt.uh[i])>>1;`
	`};`

`Rdd=vavguh(Rss,Rtt):rnd`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.uh[i]+Rtt.uh[i]+1)>>1;`
	`};`

`Rdd=vnavgh(Rtt,Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rtt.h[i]-Rss.h[i])>>1;`
	`};`

`Rdd=vnavgh(Rtt,Rss):crnd:sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=sat_16(convround(Rtt.h[i]-`
	`Rss.h[i])>>1);`
	`};`

`Rdd=vnavgh(Rtt,Rss):rnd:sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=sat_16((Rtt.h[i]-Rss.h[i]+1)>>1);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vavgh(Rss,Rtt)`	`Word64 Q6_P_vavgh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavgh(Rss,Rtt):crnd`	`Word64 Q6_P_vavgh_PP_crnd(Word64 Rss, Word64 Rtt)`
`Rdd=vavgh(Rss,Rtt):rnd`	`Word64 Q6_P_vavgh_PP_rnd(Word64 Rss, Word64 Rtt)`
`Rdd=vavguh(Rss,Rtt)`	`Word64 Q6_P_vavguh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavguh(Rss,Rtt):rnd`	`Word64 Q6_P_vavguh_PP_rnd(Word64 Rss, Word64 Rtt)`
`Rdd=vnavgh(Rtt,Rss)`	`Word64 Q6_P_vnavgh_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vnavgh(Rtt,Rss):crnd:sat`	`Word64 Q6_P_vnavgh_PP_crnd_sat(Word64 Rtt, Word64 Rss)`
`Rdd=vnavgh(Rtt,Rss):rnd:sat`	`Word64 Q6_P_vnavgh_PP_rnd_sat(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vavgh(Rss,Rtt)
1	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vavgh(Rss,Rtt):rnd
1	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vavgh(Rss,Rtt):crnd
1	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vavguh(Rss,Rtt)
1	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vavguh(Rss,Rtt):rnd
1	1	0	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vnavgh(Rtt,Rss)
1	1	0	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vnavgh(Rtt,Rss):rnd:sat
1	1	0	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vnavgh(Rtt,Rss):crnd:sat

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector average unsigned bytes

Average each of the eight unsigned bytes in the 64-bit source vector Rss with the corresponding byte in Rtt. The average operation performed on each byte is the sum of the two bytes shifted right by 1 bit. If the round option is used, then a 0x01 is also added to each result before shifting. This operation does not overflow. In the case that a summation (before right shift by 1) causes an overflow of 8 bits, the value shifted in is the most-significant carry out.

Syntax	Behavior

`Rdd=vavgub(Rss,Rtt)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=((Rss.ub[i] + Rtt.ub[i])>>1);`
	`};`

`Rdd=vavgub(Rss,Rtt):rnd`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=((Rss.ub[i]+Rtt.ub[i]+1)>>1);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vavgub(Rss,Rtt)`	`Word64 Q6_P_vavgub_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavgub(Rss,Rtt):rnd`	`Word64 Q6_P_vavgub_PP_rnd(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vavgub(Rss,Rtt)
1	1	0	1	0	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vavgub(Rss,Rtt):rnd

Field name	Description
RegType	Register Type
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector average words

Average each of the two words in the 64-bit source vector Rss with the corresponding word in Rtt. The average operation performed on each halfword adds the two words and shifts the result right by 1 bit. Unsigned average uses a logical right shift (shift in 0), whereas signed average uses an arithmetic right shift (shift in the sign bit). If the round option is used, then a 0x1 is also added to each result before shifting. This operation does not overflow. In the case that a summation (before right shift by 1) causes an overflow of 32 bits, the value shifted in is the most-significant carry out.

The signed average and negative average words is available with optional convergent rounding. In convergent rounding, if the two LSBs after the addition/subtraction are 11, then a rounding constant of 1 is added, otherwise a 0 is added. This result is then shifted right by one bit. Convergent rounding accumulates less error than arithmetic rounding.

Syntax	Behavior
`Rdd=vavguw(Rss,Rtt)[:rnd]`	`for (i=0;i<2;i++) { Rdd.w[i]=(zxt32->33(Rss.uw[i])+zxt32->33(Rtt.uw[i])+1)>>1; };`
`Rdd=vavgw(Rss,Rtt):crnd`	`for (i=0;i<2;i++) { Rdd.w[i]=(convround(sxt32->33(Rss.w[i])+sxt32->33(Rtt.w[i]))>>1); };`
`Rdd=vavgw(Rss,Rtt)[:rnd]`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt32->33(Rss.w[i])+sxt32->33(Rtt.w[i])+1)>>1; };`
`Rdd=vnavgw(Rtt,Rss)`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt32->33(Rtt.w[i])-sxt32->33(Rss.w[i]))>>1; };`
`Rdd=vnavgw(Rtt,Rss):crnd:sat`	`for (i=0;i<2;i++) { Rdd.w[i]=sat_32(convround(sxt32->33(Rtt.w[i])-sxt32->33(Rss.w[i]))>>1); };`
`Rdd=vnavgw(Rtt,Rss):rnd:sat`	`for (i=0;i<2;i++) { Rdd.w[i]=sat_32((sxt32->33(Rtt.w[i])-sxt32->33(Rss.w[i])+1)>>1); };`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vavguw(Rss,Rtt)`	`Word64 Q6_P_vavguw_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavguw(Rss,Rtt):rnd`	`Word64 Q6_P_vavguw_PP_rnd(Word64 Rss, Word64 Rtt)`
`Rdd=vavgw(Rss,Rtt)`	`Word64 Q6_P_vavgw_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vavgw(Rss,Rtt):crnd`	`Word64 Q6_P_vavgw_PP_crnd(Word64 Rss, Word64 Rtt)`
`Rdd=vavgw(Rss,Rtt):rnd`	`Word64 Q6_P_vavgw_PP_rnd(Word64 Rss, Word64 Rtt)`
`Rdd=vnavgw(Rtt,Rss)`	`Word64 Q6_P_vnavgw_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vnavgw(Rtt,Rss):crnd:sat`	`Word64 Q6_P_vnavgw_PP_crnd_sat(Word64 Rtt, Word64 Rss)`
`Rdd=vnavgw(Rtt,Rss):rnd:sat`	`Word64 Q6_P_vnavgw_PP_rnd_sat(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vavgw(Rss,Rtt)
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vavgw(Rss,Rtt):rnd
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vavgw(Rss,Rtt):crnd
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vavguw(Rss,Rtt)
1	1	0	1	0	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vavguw(Rss,Rtt):rnd
1	1	0	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vnavgw(Rtt,Rss)
1	1	0	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vnavgw(Rtt,Rss):rnd:sat
1	1	0	1	0	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vnavgw(Rtt,Rss):crnd:sat

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector conditional negate

Based on bits in Rt, conditionally negate halves in Rss.

Syntax	Behavior

`Rdd=vcnegh(Rss,Rt)`	`for (i = 0; i < 4; i++) {`
	`  if (Rt.i) {`
	`  Rdd.h[i]=sat_16(-Rss.h[i]);`
	`  } else {`
	`  Rdd.h[i]=Rss.h[i];`
	`  };`
	`};`

`Rxx+=vrcnegh(Rss,Rt)`	`for (i = 0; i < 4; i++) {`
	`  if (Rt.i) {`
	`  Rxx += -Rss.h[i];`
	`  } else {`
	`  Rxx += Rss.h[i];`
	`  };`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vcnegh(Rss,Rt)`	`Word64 Q6_P_vcnegh_PR(Word64 Rss, Word32 Rt)`
`Rxx+=vrcnegh(Rss,Rt)`	`Word64 Q6_P_vrcneghacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vcnegh(Rss,Rt)
ICLASS				RegType				Maj			s5					Parse			t5					Min			x5					
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vrcnegh(Rss,Rt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Field name	Description
x5	Field to encode register x
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Vector maximum bytes

Compare each of the eight unsigned bytes in the 64-bit source vector Rss to the corresponding byte in Rtt. For each comparison, select the maximum of the two bytes and place that byte in the corresponding location in Rdd.

Syntax	Behavior

`Rdd=vmaxb(Rtt,Rss)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=max(Rtt.b[i],Rss.b[i]);`
	`};`

`Rdd=vmaxub(Rtt,Rss)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=max(Rtt.ub[i],Rss.ub[i]);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vmaxb(Rtt,Rss)`	`Word64 Q6_P_vmaxb_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vmaxub(Rtt,Rss)`	`Word64 Q6_P_vmaxub_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxub(Rtt,Rss)
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxb(Rtt,Rss)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector maximum halfwords

Compare each of the four halfwords in the 64-bit source vector Rss to the corresponding halfword in Rtt. For each comparison, select the maximum of the two halfwords and place that halfword in the corresponding location in Rdd. Comparisons are available in both signed and unsigned form.

Syntax	Behavior

`Rdd=vmaxh(Rtt,Rss)`	`for (i = 0; i < 4; i++) {`
	`  Rdd.h[i]=max(Rtt.h[i],Rss.h[i]);`
	`};`

`Rdd=vmaxuh(Rtt,Rss)`	`for (i = 0; i < 4; i++) {`
	`  Rdd.h[i]=max(Rtt.uh[i],Rss.uh[i]);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vmaxh(Rtt,Rss)`	`Word64 Q6_P_vmaxh_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vmaxuh(Rtt,Rss)`	`Word64 Q6_P_vmaxuh_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxh(Rtt,Rss)
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxuh(Rtt,Rss)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector reduce maximum halfwords

Register Rxx contains a maximum value in the low word and the address of that maximum value in the high word. Register Rss contains a vector of four halfword values, and register Ru contains the address of this data. The instruction finds the maximum halfword between the previous maximum in Rxx[0] and the four values in Rss. The address of the new maximum is stored in Rxx[1].

Syntax	Behavior

`Rxx=vrmaxh(Rss,Ru)`	`max = Rxx.h[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 4; i++) {`
	`  if (max < Rss.h[i]) {`
	`  max = Rss.h[i];`
	`  addr = Ru | i<<1;`
	`  };`
	`};`
	`Rxx.w[0]=max;`
	`Rxx.w[1]=addr;`

`Rxx=vrmaxuh(Rss,Ru)`	`max = Rxx.uh[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 4; i++) {`
	`  if (max < Rss.uh[i]) {`
	`  max = Rss.uh[i];`
	`  addr = Ru | i<<1;`
	`  };`
	`};`
	`Rxx.w[0]=max;`
	`Rxx.w[1]=addr;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rxx=vrmaxh(Rss,Ru)`	`Word64 Q6_P_vrmaxh_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`
`Rxx=vrmaxuh(Rss,Ru)`	`Word64 Q6_P_vrmaxuh_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			x5					Min			u5					
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**x**	**x**	**x**	**x**	**x**	0	0	1	**u**	**u**	**u**	**u**	**u**	Rxx=vrmaxh(Rss,Ru)
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**x**	**x**	**x**	**x**	**x**	0	0	1	**u**	**u**	**u**	**u**	**u**	Rxx=vrmaxuh(Rss,Ru)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`s5`	Field to encode register s

Field name	Description
u5	Field to encode register u
x5	Field to encode register x
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Vector reduce maximum words

Find the maximum word between the previous maximum in Rxx[0] and the two values in Rss. The address of the new maximum is stored in Rxx[1].

Register Rxx contains a maximum value in the low word and the address of that maximum value in the high word. Register Rss contains a vector of two word values, and register Ru contains the address of this data.

Syntax	Behavior

`Rxx=vrmaxuw(Rss,Ru)`	`max = Rxx.uw[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 2; i++) {`
	`  if (max < Rss.uw[i]) {`
	`  max = Rss.uw[i];`
	`  addr = Ru | i<<2;`
	`  };`
	`};`
	`Rxx.w[0]=max;`
	`Rxx.w[1]=addr;`

`Rxx=vrmaxw(Rss,Ru)`	`max = Rxx.w[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 2; i++) {`
	`  if (max < Rss.w[i]) {`
	`  max = Rss.w[i];`
	`  addr = Ru | i<<2;`
	`  };`
	`};`
	`Rxx.w[0]=max;`
	`Rxx.w[1]=addr;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rxx=vrmaxuw(Rss,Ru)`	`Word64 Q6_P_vrmaxuw_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`
`Rxx=vrmaxw(Rss,Ru)`	`Word64 Q6_P_vrmaxw_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			x5					Min			u5					
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**x**	**x**	**x**	**x**	**x**	0	1	0	**u**	**u**	**u**	**u**	**u**	Rxx=vrmaxw(Rss,Ru)
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**x**	**x**	**x**	**x**	**x**	0	1	0	**u**	**u**	**u**	**u**	**u**	Rxx=vrmaxuw(Rss,Ru)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
u5	Field to encode register u
x5	Field to encode register x
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Vector maximum words

Compare each of the two words in the 64-bit source vector Rss to the corresponding word in Rtt. For each comparison, select the maximum of the two words and place that word in the corresponding location in Rdd.

Comparisons are available in both signed and unsigned form.

Syntax	Behavior

`Rdd=vmaxuw(Rtt,Rss)`	`for (i = 0; i < 2; i++) {`
	`  Rdd.w[i]=max(Rtt.uw[i],Rss.uw[i]);`
	`};`

`Rdd=vmaxw(Rtt,Rss)`	`for (i = 0; i < 2; i++) {`
	`  Rdd.w[i]=max(Rtt.w[i],Rss.w[i]);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vmaxuw(Rtt,Rss)`	`Word64 Q6_P_vmaxuw_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vmaxw(Rtt,Rss)`	`Word64 Q6_P_vmaxw_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxuw(Rtt,Rss)
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmaxw(Rtt,Rss)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector minimum bytes

Compare each of the eight unsigned bytes in the 64-bit source vector Rss to the corresponding byte in Rtt. For each comparison, select the minimum of the two bytes and place that byte in the corresponding location in Rdd.

Syntax	Behavior

`Rdd,Pe=vminub(Rtt,Rss)`	`for (i = 0; i < 8; i++) {`
	`  Pe.i = (Rtt.ub[i] > Rss.ub[i]);`
	`  Rdd.b[i]=min(Rtt.ub[i],Rss.ub[i]);`
	`};`

`Rdd=vminb(Rtt,Rss)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=min(Rtt.b[i],Rss.b[i]);`
	`};`

`Rdd=vminub(Rtt,Rss)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=min(Rtt.ub[i],Rss.ub[i]);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- The predicate generated by this instruction can not be used as a .new predicate, nor can it be automatically ANDed with another predicate.

Intrinsics

`Rdd=vminb(Rtt,Rss)`	`Word64 Q6_P_vminb_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vminub(Rtt,Rss)`	`Word64 Q6_P_vminub_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vminub(Rtt,Rss)
1	1	0	1	0	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vminb(Rtt,Rss)
ICLASS				RegType				MajOp			s5					Parse			t5						e2		d5					
1	1	1	0	1	0	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**e**	**e**	**d**	**d**	**d**	**d**	**d**	Rdd,Pe=vminub(Rtt,Rss)

Field name	Description
RegType	Register Type
MinOp	Minor Opcode
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
e2	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t

NOTE `Rdd,Pe=vminub` is supported only in processor versions V61 or greater.

Vector minimum halfwords

Compare each of the four halfwords in the 64-bit source vector Rss to the corresponding halfword in Rtt. For each comparison, select the minimum of the two halfwords and place that halfword in the corresponding location in Rdd.

Comparisons are available in both signed and unsigned form.

Syntax	Behavior

`Rdd=vminh(Rtt,Rss)`	`for (i = 0; i < 4; i++) {`
	`  Rdd.h[i]=min(Rtt.h[i],Rss.h[i]);`
	`};`

`Rdd=vminuh(Rtt,Rss)`	`for (i = 0; i < 4; i++) {`
	`  Rdd.h[i]=min(Rtt.uh[i],Rss.uh[i]);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vminh(Rtt,Rss)`	`Word64 Q6_P_vminh_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vminuh(Rtt,Rss)`	`Word64 Q6_P_vminuh_PP(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vminh(Rtt,Rss)
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vminuh(Rtt,Rss)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector reduce minimum halfwords

Find the minimum halfword between the previous minimum in Rxx[0] and the four values in Rss. The address of the new minimum is stored in Rxx[1].

Register Rxx contains a minimum value in the low word and the address of that minimum value in the high word. Register Rss contains a vector of four halfword values, and register Ru contains the address of this data.

Syntax	Behavior

`Rxx=vrminh(Rss,Ru)`	`min = Rxx.h[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 4; i++) {`
	`  if (min > Rss.h[i]) {`
	`  min = Rss.h[i];`
	`  addr = Ru | i<<1;`
	`  };`
	`};`
	`Rxx.w[0]=min;`
	`Rxx.w[1]=addr;`

`Rxx=vrminuh(Rss,Ru)`	`min = Rxx.uh[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 4; i++) {`
	`  if (min > Rss.uh[i]) {`
	`  min = Rss.uh[i];`
	`  addr = Ru | i<<1;`
	`  };`
	`};`
	`Rxx.w[0]=min;`
	`Rxx.w[1]=addr;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rxx=vrminh(Rss,Ru)`	`Word64 Q6_P_vrminh_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`
`Rxx=vrminuh(Rss,Ru)`	`Word64 Q6_P_vrminuh_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			x5					Min			u5					
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**x**	**x**	**x**	**x**	**x**	1	0	1	**u**	**u**	**u**	**u**	**u**	Rxx=vrminh(Rss,Ru)
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**x**	**x**	**x**	**x**	**x**	1	0	1	**u**	**u**	**u**	**u**	**u**	Rxx=vrminuh(Rss,Ru)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
u5	Field to encode register u
x5	Field to encode register x
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Vector reduce minimum words

Find the minimum word between the previous minimum in Rxx[0] and the two values in Rss. The address of the new minimum is stored in Rxx[1].

Register Rxx contains a minimum value in the low word and the address of that minimum value in the high word. Register Rss contains a vector of two word values, and register Ru contains the address of this data.

Syntax	Behavior

`Rxx=vrminuw(Rss,Ru)`	`min = Rxx.uw[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 2; i++) {`
	`  if (min > Rss.uw[i]) {`
	`  min = Rss.uw[i];`
	`  addr = Ru | i<<2;`
	`  };`
	`};`
	`Rxx.w[0]=min;`
	`Rxx.w[1]=addr;`

`Rxx=vrminw(Rss,Ru)`	`min = Rxx.w[0];`
	`addr = Rxx.w[1];`
	`for (i = 0; i < 2; i++) {`
	`  if (min > Rss.w[i]) {`
	`  min = Rss.w[i];`
	`  addr = Ru | i<<2;`
	`  };`
	`};`
	`Rxx.w[0]=min;`
	`Rxx.w[1]=addr;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rxx=vrminuw(Rss,Ru)`	`Word64 Q6_P_vrminuw_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`
`Rxx=vrminw(Rss,Ru)`	`Word64 Q6_P_vrminw_PR(Word64 Rxx, Word64 Rss, Word32 Ru)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			x5					Min			u5					
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**x**	**x**	**x**	**x**	**x**	1	1	0	**u**	**u**	**u**	**u**	**u**	Rxx=vrminw(Rss,Ru)
1	1	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**x**	**x**	**x**	**x**	**x**	1	1	0	**u**	**u**	**u**	**u**	**u**	Rxx=vrminuw(Rss,Ru)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
u5	Field to encode register u
x5	Field to encode register x
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Vector minimum words

Compare each of the two words in the 64-bit source vector Rss to the corresponding word in Rtt. For each comparison, select the minimum of the two words and place that word in the corresponding location in Rdd.

Comparisons are available in both signed and unsigned form.

Syntax	Behavior

`Rdd=vminuw(Rtt,Rss)`	`for (i = 0; i < 2; i++) {`
	`  Rdd.w[i]=min(Rtt.uw[i],Rss.uw[i]);`
	`};`

`Rdd=vminw(Rtt,Rss)`	`for (i = 0; i < 2; i++) {`
	`  Rdd.w[i]=min(Rtt.w[i],Rss.w[i]);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vminuw(Rtt,Rss)`	`Word64 Q6_P_vminuw_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vminw(Rtt,Rss)`	`Word64 Q6_P_vminw_PP(Word64 Rtt, Word64 Rss)`

Encoding

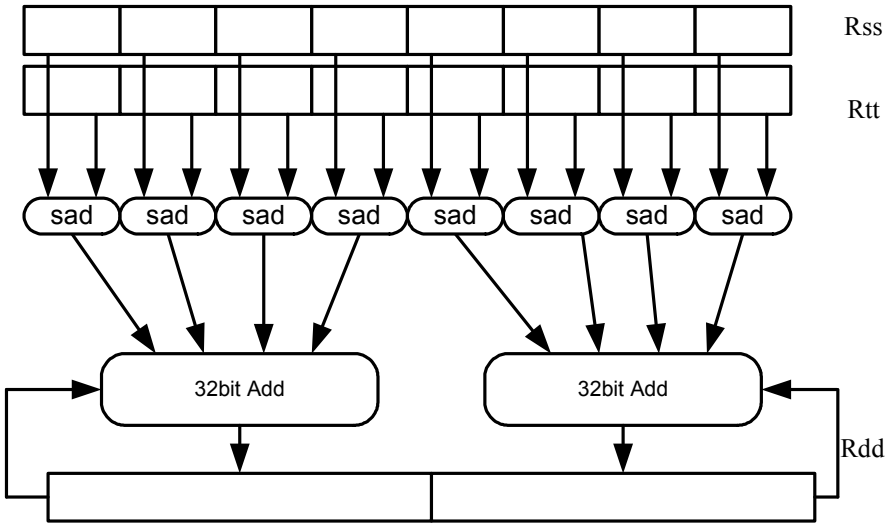
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vminw(Rtt,Rss)
1	1	0	1	0	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vminuw(Rtt,Rss)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector sum of absolute differences unsigned bytes

For each byte in the source vector Rss, subtract the corresponding byte in source vector Rtt. Take the absolute value of the intermediate results, and the upper four together and add the lower four together. Optionally, add the destination upper and lower words to these results.

This instruction is useful in determining distance between two vectors, in applications such as motion estimation.



Syntax	Behavior

`Rdd=vrsadub(Rss,Rtt)`	`Rdd = 0;`
	`for (i = 0; i < 4; i++) {`
	`  Rdd.w[0]=(Rdd.w[0] + ABS((Rss.ub[i] - Rtt.ub[i])));`
	`};`
	`for (i = 4; i < 8; i++) {`
	`  Rdd.w[1]=(Rdd.w[1] + ABS((Rss.ub[i] - Rtt.ub[i])));`
	`};`

`Rxx+=vrsadub(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Rxx.w[0]=(Rxx.w[0] + ABS((Rss.ub[i] - Rtt.ub[i])));`
	`};`
	`for (i = 4; i < 8; i++) {`
	`  Rxx.w[1]=(Rxx.w[1] + ABS((Rss.ub[i] - Rtt.ub[i])));`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vrsadub(Rss,Rtt)`	`Word64 Q6_P_vrsadub_PP(Word64 Rss, Word64 Rtt)`
`Rxx+=vrsadub(Rss,Rtt)`	`Word64 Q6_P_vrsadubacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrsadub(Rss,Rtt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrsadub(Rss,Rtt)

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Vector subtract halfwords

Subtract each of the four halfwords in 64-bit vector Rss from the corresponding halfword in vector Rtt.

Optionally, saturate each 16-bit addition to either a signed or unsigned 16-bit value. Applying saturation to the vsubh instruction clamps the result to the signed range 0x8000 to 0x7fff, whereas applying saturation to the vsubuh instruction ensures that the unsigned result falls within the range 0 to 0xffff.

When saturation is not needed, vsubh should be used.

Syntax	Behavior

`Rdd=vsubh(Rtt,Rss)[:sat]`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=[sat_16](Rtt.h[i]-Rss.h[i]);`
	`};`

`Rdd=vsubuh(Rtt,Rss):sat`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=usat_16(Rtt.uh[i]-Rss.uh[i]);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vsubh(Rtt,Rss)`	`Word64 Q6_P_vsubh_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vsubh(Rtt,Rss):sat`	`Word64 Q6_P_vsubh_PP_sat(Word64 Rtt, Word64 Rss)`
`Rdd=vsubuh(Rtt,Rss):sat`	`Word64 Q6_P_vsubuh_PP_sat(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsubh(Rtt,Rss)
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsubh(Rtt,Rss):sat
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsubuh(Rtt,Rss):sat

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class

Field name	Description
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector subtract bytes

Subtract each of the eight bytes in 64-bit vector Rss from the corresponding byte in vector Rtt.

Optionally, saturate each 8-bit subtraction to an unsigned value between 0 and 255. The eight results are stored in destination register Rdd.

Syntax	Behavior
`Rdd=vsubb(Rss,Rtt)`	`Assembler mapped to: "Rdd=vsubub(Rss,Rtt)"`

`Rdd=vsubub(Rtt,Rss)[:sat]`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=[usat_8](Rtt.ub[i]-Rss.ub[i]);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vsubb(Rss,Rtt)`	`Word64 Q6_P_vsubb_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vsubub(Rtt,Rss)`	`Word64 Q6_P_vsubub_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vsubub(Rtt,Rss):sat`	`Word64 Q6_P_vsubub_PP_sat(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsubub(Rtt,Rss)
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsubub(Rtt,Rss):sat

Field name	Description
RegType	Register Type
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector subtract words

Subtract each of the two words in 64-bit vector Rss from the corresponding word in vector Rtt.

Optionally, saturate each 32-bit subtraction to a signed value between 0x8000_0000 and 0x7fff_ffff. The two word results are stored in destination register Rdd.

Syntax	Behavior

`Rdd=vsubw(Rtt,Rss)[:sat]`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=[sat_32](Rtt.w[i]-Rss.w[i]);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vsubw(Rtt,Rss)`	`Word64 Q6_P_vsubw_PP(Word64 Rtt, Word64 Rss)`
`Rdd=vsubw(Rtt,Rss):sat`	`Word64 Q6_P_vsubw_PP_sat(Word64 Rtt, Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsubw(Rtt,Rss)
1	1	0	1	0	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsubw(Rtt,Rss):sat

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

11.12.2 XTYPE/BIT

The XTYPE/BIT instruction subclass includes instructions for bit manipulation.

Count leading

Count leading zeros (cl0) counts the number of consecutive zeros starting with the most significant bit.

Count leading ones (cl1) counts the number of consecutive ones starting with the most significant bit.

Count leading bits (clb) counts both leading ones and leading zeros and then selects the maximum.

The NORMAMT instruction returns the number of leading bits minus one.

For a two's-complement number, the number of leading zeros is zero for negative numbers. The number of leading ones is zero for positive numbers.

The number of leading bits can be used to judge the magnitude of the value.

Syntax	Behavior
`Rd=add(clb(Rs),#s6)`	`Rd = (max(count_leading_ones(Rs),count_leading_ones(~Rs)))+#s;`
`Rd=add(clb(Rss),#s6)`	`Rd = (max(count_leading_ones(Rss),count_leading_ones(~Rss)))+#s;`
`Rd=cl0(Rs)`	`Rd = count_leading_ones(~Rs);`
`Rd=cl0(Rss)`	`Rd = count_leading_ones(~Rss);`
`Rd=cl1(Rs)`	`Rd = count_leading_ones(Rs);`
`Rd=cl1(Rss)`	`Rd = count_leading_ones(Rss);`
`Rd=clb(Rs)`	`Rd = max(count_leading_ones(Rs),count_leading_ones(~Rs));`
`Rd=clb(Rss)`	`Rd = max(count_leading_ones(Rss),count_leading_ones(~Rss));`

`Rd=normamt(Rs)`	`if (Rs == 0) {`
	`  Rd = 0;`
	`} else {`
	`  Rd = (max(count_leading_ones(Rs),count_leading_ones(~Rs)))-1;`
	`};`

`Rd=normamt(Rss)`	`if (Rss == 0) {`
	`  Rd = 0;`
	`} else {`
	`  Rd = (max(count_leading_ones(Rss),count_leading_ones(~Rss)))-1;`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=add(clb(Rs),#s6)`	`Word32 Q6_R_add_clb_RI(Word32 Rs, Word32 Is6)`
`Rd=add(clb(Rss),#s6)`	`Word32 Q6_R_add_clb_PI(Word64 Rss, Word32 Is6)`
`Rd=cl0(Rs)`	`Word32 Q6_R_cl0_R(Word32 Rs)`
`Rd=cl0(Rss)`	`Word32 Q6_R_cl0_P(Word64 Rss)`
`Rd=cl1(Rs)`	`Word32 Q6_R_cl1_R(Word32 Rs)`
`Rd=cl1(Rss)`	`Word32 Q6_R_cl1_P(Word64 Rss)`
`Rd=clb(Rs)`	`Word32 Q6_R_clb_R(Word32 Rs)`
`Rd=clb(Rss)`	`Word32 Q6_R_clb_P(Word64 Rss)`
`Rd=normamt(Rs)`	`Word32 Q6_R_normamt_R(Word32 Rs)`
`Rd=normamt(Rss)`	`Word32 Q6_R_normamt_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=clb(Rss)
1	0	0	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=cl0(Rss)
1	0	0	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=cl1(Rss)
1	0	0	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=normamt(Rss)
1	0	0	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=add(clb(Rss),#s6)
1	0	0	0	1	1	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=add(clb(Rs),#s6)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=clb(Rs)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=cl0(Rs)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=cl1(Rs)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=normamt(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Count population

Population Count (popcount) counts the number of bits in Rss that are set.

Syntax	Behavior
`Rd=popcount(Rss)`	`Rd = count_ones(Rss);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=popcount(Rss)`	`Word32 Q6_R_popcount_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=popcount(Rss)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Count trailing

Count trailing zeros (ct0) counts the number of consecutive zeros starting with the least significant bit.

Count trailing ones (ct1) counts the number of consecutive ones starting with the least significant bit.

Syntax	Behavior
`Rd=ct0(Rs)`	`Rd = count_leading_ones(~reverse_bits(Rs));`
`Rd=ct0(Rss)`	`Rd = count_leading_ones(~reverse_bits(Rss));`
`Rd=ct1(Rs)`	`Rd = count_leading_ones(reverse_bits(Rs));`
`Rd=ct1(Rss)`	`Rd = count_leading_ones(reverse_bits(Rss));`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=ct0(Rs)`	`Word32 Q6_R_ct0_R(Word32 Rs)`
`Rd=ct0(Rss)`	`Word32 Q6_R_ct0_P(Word64 Rss)`
`Rd=ct1(Rs)`	`Word32 Q6_R_ct1_R(Word32 Rs)`
`Rd=ct1(Rss)`	`Word32 Q6_R_ct1_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=ct0(Rss)
1	0	0	0	1	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=ct1(Rss)
1	0	0	0	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=ct0(Rs)
1	0	0	0	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=ct1(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

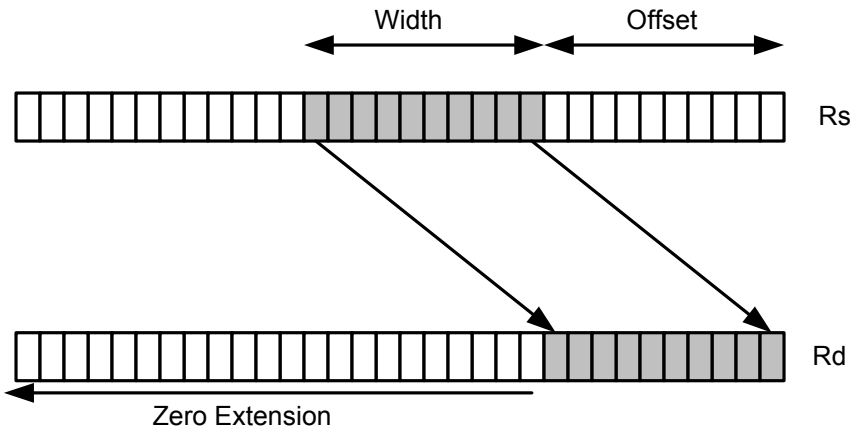
Extract bitfield

Extract a bitfield from the source register (or register pair) and deposit into the least significant bits of the destination register (or register pair). The other, more significant bits in the destination are either cleared or sign-extended, depending on the instruction.

The width of the extracted field is obtained from the first immediate or from the most-significant word of Rtt. The field offset is obtained from either the second immediate or from the least-significant word of Rtt.

For register-based extract, where Rtt supplies the offset and width, the offset value is treated as a signed 7-bit number. If this value is negative, the source register Rss is shifted left (the reverse direction). Width number of bits are then taken from the least-significant portion of this result.

If the shift amount and/or offset captures data beyond the most significant end of the input, then these bits will be taken as zero.



Syntax	Behavior

`Rd=extract(Rs,#u5,#U5)`	`width=#u;`
	`offset=#U;`
	`Rd = sxt`$_{width->32}$`((Rs >> offset));`

`Rd=extract(Rs,Rtt)`	`width=zxt`$_{6->32}$`((Rtt.w[1]));`
	`offset=sxt`$_{7->32}$`((Rtt.w[0]));`
	`Rd = sxt`$_{width->64}$`((offset>0)?(zxt`$_{32->64}$`(zxt`$_{32->64}$`(Rs))>>>offset):(zxt`$_{32->64}$`(zxt`$_{32->64}$`(Rs))<<offset));`

`Rd=extractu(Rs,#u5,#U5)`	`width=#u;`
	`offset=#U;`
	`Rd = zxt`$_{width->32}$`((Rs >> offset));`

`Rd=extractu(Rs,Rtt)`	`width=zxt`$_{6->32}$`((Rtt.w[1]));`
	`offset=sxt`$_{7->32}$`((Rtt.w[0]));`
	`Rd = zxt`$_{width->64}$`((offset>0)?(zxt`$_{32->64}$`(zxt`$_{32->64}$`(Rs))>>>offset):(zxt`$_{32->64}$`(zxt`$_{32->64}$`(Rs))<<offset));`

`Rdd=extract(Rss,#u6,#U6)`	`width=#u;`
	`offset=#U;`
	`Rdd = sxt`$_{width->64}$`((Rss >> offset));`

`Rdd=extract(Rss,Rtt)`	`width=zxt`$_{6->32}$`((Rtt.w[1]));`
	`offset=sxt`$_{7->32}$`((Rtt.w[0]));`
	`Rdd = sxt`$_{width->64}$`((offset>0)?(Rss>>>offset):(Rss<<offset));`

`Rdd=extractu(Rss,#u6,#U6)`	`width=#u;`
	`offset=#U;`
	`Rdd = zxt`$_{width->64}$`((Rss >> offset));`

`Rdd=extractu(Rss,Rtt)`	`width=zxt`$_{6->32}$`((Rtt.w[1]));`
	`offset=sxt`$_{7->32}$`((Rtt.w[0]));`
	`Rdd = zxt`$_{width->64}$`((offset>0)?(Rss>>>offset):(Rss<<offset));`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=extract(Rs,#u5,#U5)`	`Word32 Q6_R_extract_RII(Word32 Rs, Word32 Iu5, Word32 IU5)`
`Rd=extract(Rs,Rtt)`	`Word32 Q6_R_extract_RP(Word32 Rs, Word64 Rtt)`
`Rd=extractu(Rs,#u5,#U5)`	`Word32 Q6_R_extractu_RII(Word32 Rs, Word32 Iu5, Word32 IU5)`
`Rd=extractu(Rs,Rtt)`	`Word32 Q6_R_extractu_RP(Word32 Rs, Word64 Rtt)`
`Rdd=extract(Rss,#u6,#U6)`	`Word64 Q6_P_extract_PII(Word64 Rss, Word32 Iu6, Word32 IU6)`
`Rdd=extract(Rss,Rtt)`	`Word64 Q6_P_extract_PP(Word64 Rss, Word64 Rtt)`
`Rdd=extractu(Rss,#u6,#U6)`	`Word64 Q6_P_extractu_PII(Word64 Rss, Word32 Iu6, Word32 IU6)`
`Rdd=extractu(Rss,Rtt)`	`Word64 Q6_P_extractu_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	1	**I**	**I**	**I**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rdd=extractu(Rss,#u6,#U6)
1	0	0	0	1	0	1	0	**I**	**I**	**I**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rdd=extract(Rss,#u6,#U6)
1	0	0	0	1	1	0	1	0	**I**	**I**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=extractu(Rs,#u5,#U5)
1	0	0	0	1	1	0	1	1	**I**	**I**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**d**	**d**	**d**	**d**	**d**	Rd=extract(Rs,#u5,#U5)
ICLASS				RegType				Maj			s5					Parse			t5						Min			d5					
1	1	0	0	0	0	0	1	0	0	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	0	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=extractu(Rss,Rtt)
1	1	0	0	0	0	0	1	1	1	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	0	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=extract(Rss,Rtt)
1	1	0	0	1	0	0	1	0	0	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	0	**-**	**d**	**d**	**d**	**d**	**d**	Rd=extractu(Rs,Rtt)
1	1	0	0	1	0	0	1	0	0	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	1	**-**	**d**	**d**	**d**	**d**	**d**	Rd=extract(Rs,Rtt)

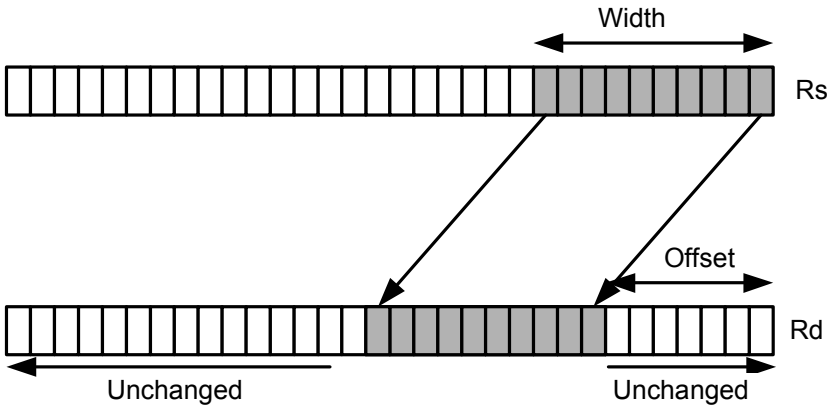
Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`Maj`	Major Opcode
`Min`	Minor Opcode
`RegType`	Register Type
`RegType`	Register Type

Insert bitfield

Replace a bitfield in the destination register (or register pair) with bits from the least significant portion of Rs/Rss. The number of bits is obtained from the first immediate or the most-significant word of Rtt. The bits are shifted by the second immediate or the least significant word of Rtt.

If register Rtt specifies the offset, then the low 7-bits of Rtt are treated as a signed 7-bit value. If this value is negative, the result is zero.

Shift amounts and offsets that are too large may push bits beyond the end of the destination register, in this case the bits will not appear in the destination register.



Syntax	Behavior

`Rx=insert(Rs,#u5,#U5)`	`width=#u;`
	`offset=#U;`
	`Rx &= ~(((1<<width)-1)<<offset);`
	`Rx |= ((Rs & ((1<<width)-1)) << offset);`

`Rx=insert(Rs,Rtt)`	`width=zxt`$_{6->32}$`((Rtt.w[1]));`
	`offset=sxt`$_{7->32}$`((Rtt.w[0]));`
	`mask = ((1<<width)-1);`
	`if (offset < 0) {`
	`  Rx = 0;`
	`} else {`
	`  Rx &= ~(mask<<offset);`
	`  Rx |= ((Rs & mask) << offset);`
	`};`

`Rxx=insert(Rss,#u6,#U6)`	`width=#u;`
	`offset=#U;`
	`Rxx &= ~(((1<<width)-1)<<offset);`
	`Rxx |= ((Rss & ((1<<width)-1)) << offset);`

`Rxx=insert(Rss,Rtt)`	`width=zxt`$_{6->32}$`((Rtt.w[1]));`
	`offset=sxt`$_{7->32}$`((Rtt.w[0]));`
	`mask = ((1<<width)-1);`
	`if (offset < 0) {`
	`  Rxx = 0;`
	`} else {`
	`  Rxx &= ~(mask<<offset);`
	`  Rxx |= ((Rss & mask) << offset);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx=insert(Rs,#u5,#U5)`	`Word32 Q6_R_insert_RII(Word32 Rx, Word32 Rs, Word32 Iu5, Word32 IU5)`
`Rx=insert(Rs,Rtt)`	`Word32 Q6_R_insert_RP(Word32 Rx, Word32 Rs, Word64 Rtt)`
`Rxx=insert(Rss,#u6,#U6)`	`Word64 Q6_P_insert_PII(Word64 Rxx, Word64 Rss, Word32 Iu6, Word32 IU6)`
`Rxx=insert(Rss,Rtt)`	`Word64 Q6_P_insert_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	0	0	0	0	0	1	1	**I**	**I**	**I**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**x**	**x**	**x**	**x**	**x**	Rxx=insert(Rss,#u6,#U6)
1	0	0	0	1	1	1	1	0	**I**	**I**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**I**	**I**	**I**	**x**	**x**	**x**	**x**	**x**	Rx=insert(Rs,#u5,#U5)
ICLASS				RegType							s5					Parse			t5								x5					
1	1	0	0	1	0	0	0	**-**	**-**	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**x**	**x**	**x**	**x**	**x**	Rx=insert(Rs,Rtt)
ICLASS				RegType				Maj			s5					Parse			t5								x5					
1	1	0	0	1	0	1	0	0	**-**	**-**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**x**	**x**	**x**	**x**	**x**	Rxx=insert(Rss,Rtt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x
MajOp	Major Opcode
MinOp	Minor Opcode
Maj	Major Opcode
RegType	Register Type
RegType	Register Type

Interleave/deinterleave

For interleave, bits I+32 of Rss (which are the bits from the upper source word) get placed in the odd bits (I*2)+1 of Rdd, while bits I of Rss (which are the bits from the lower source word) get placed in the even bits (I*2) of Rdd.

For deinterleave, the even bits of the source register are placed in the even register of the result pair, and the odd bits of the source register are placed in the odd register of the result pair.

Note that "r1:0 = deinterleave(r1:0)" is the inverse of "r1:0 = interleave(r1:0)".

Syntax	Behavior
`Rdd=deinterleave(Rss)`	`Rdd = deinterleave(ODD,EVEN);`
`Rdd=interleave(Rss)`	`Rdd = interleave(Rss.w[1],Rss.w[0]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=deinterleave(Rss)`	`Word64 Q6_P_deinterleave_P(Word64 Rss)`
`Rdd=interleave(Rss)`	`Word64 Q6_P_interleave_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=deinterleave(Rss)
1	0	0	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=interleave(Rss)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Linear feedback-shift iteration

Count the number of ones of the logical AND of the two source input values, and take the least significant value of that sum. The first source value is shifted right by one bit, and the parity is placed in the MSB.

Syntax	Behavior
`Rdd=lfs(Rss,Rtt)`	`Rdd = (Rss.u64 >> 1) | ((1&count_ones(Rss & Rtt)).u64<<63) ;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=lfs(Rss,Rtt)`	`Word64 Q6_P_lfs_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=lfs(Rss,Rtt)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major Opcode
`Min`	Minor Opcode
`RegType`	Register Type

Masked parity

Count the number of ones of the logical AND of the two source input values, and take the least significant bit of that sum.

Syntax	Behavior
`Rd=parity(Rs,Rt)`	`Rd = 1&count_ones(Rs & Rt);`
`Rd=parity(Rss,Rtt)`	`Rd = 1&count_ones(Rss & Rtt);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=parity(Rs,Rt)`	`Word32 Q6_R_parity_RR(Word32 Rs, Word32 Rt)`
`Rd=parity(Rss,Rtt)`	`Word32 Q6_R_parity_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5								d5					
1	1	0	1	0	0	0	0	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=parity(Rss,Rtt)
1	1	0	1	0	1	0	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=parity(Rs,Rt)

Field name	Description
`RegType`	Register Type
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Bit reverse

Reverse the order of bits. The most significant gets swapped with the least significant, bit 30 gets swapped with bit 1, and so on.

Syntax	Behavior
`Rd=brev(Rs)`	`Rd = reverse_bits(Rs);`
`Rdd=brev(Rss)`	`Rdd = reverse_bits(Rss);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=brev(Rs)`	`Word32 Q6_R_brev_R(Word32 Rs)`
`Rdd=brev(Rss)`	`Word64 Q6_P_brev_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=brev(Rss)
1	0	0	0	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=brev(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Set/clear/toggle bit

Set (to 1), clear (to 0), or toggle a single bit in the source, and place the resulting value in the destination. The bit to be manipulated can be indicated using an immediate or register value.

If a register is used to indicate the bit position, and the value of the least-significant 7 bits of Rt is out of range, then the destination register will be unchanged.

Syntax	Behavior
`Rd=clrbit(Rs,#u5)`	`Rd = (Rs & (~(1<<#u)));`
`Rd=clrbit(Rs,Rt)`	`Rd = (Rs & (~((sxt7->32(Rt)>0)?(zxt32->64(1)<<sxt7->32(Rt)):(zxt32->64(1)>>>sxt7->32(Rt)))));`
`Rd=setbit(Rs,#u5)`	`Rd = (Rs | (1<<#u));`
`Rd=setbit(Rs,Rt)`	`Rd = (Rs | (sxt7->32(Rt)>0)?(zxt32->64(1)<<sxt7->32(Rt)):(zxt32->64(1)>>>sxt7->32(Rt)));`
`Rd=togglebit(Rs,#u5)`	`Rd = (Rs ^ (1<<#u));`
`Rd=togglebit(Rs,Rt)`	`Rd = (Rs ^ (sxt7->32(Rt)>0)?(zxt32->64(1)<<sxt7->32(Rt)):(zxt32->64(1)>>>sxt7->32(Rt)));`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=clrbit(Rs,#u5)`	`Word32 Q6_R_clrbit_RI(Word32 Rs, Word32 Iu5)`
`Rd=clrbit(Rs,Rt)`	`Word32 Q6_R_clrbit_RR(Word32 Rs, Word32 Rt)`
`Rd=setbit(Rs,#u5)`	`Word32 Q6_R_setbit_RI(Word32 Rs, Word32 Iu5)`
`Rd=setbit(Rs,Rt)`	`Word32 Q6_R_setbit_RR(Word32 Rs, Word32 Rt)`
`Rd=togglebit(Rs,#u5)`	`Word32 Q6_R_togglebit_RI(Word32 Rs, Word32 Iu5)`
`Rd=togglebit(Rs,Rt)`	`Word32 Q6_R_togglebit_RR(Word32 Rs, Word32 Rt)`

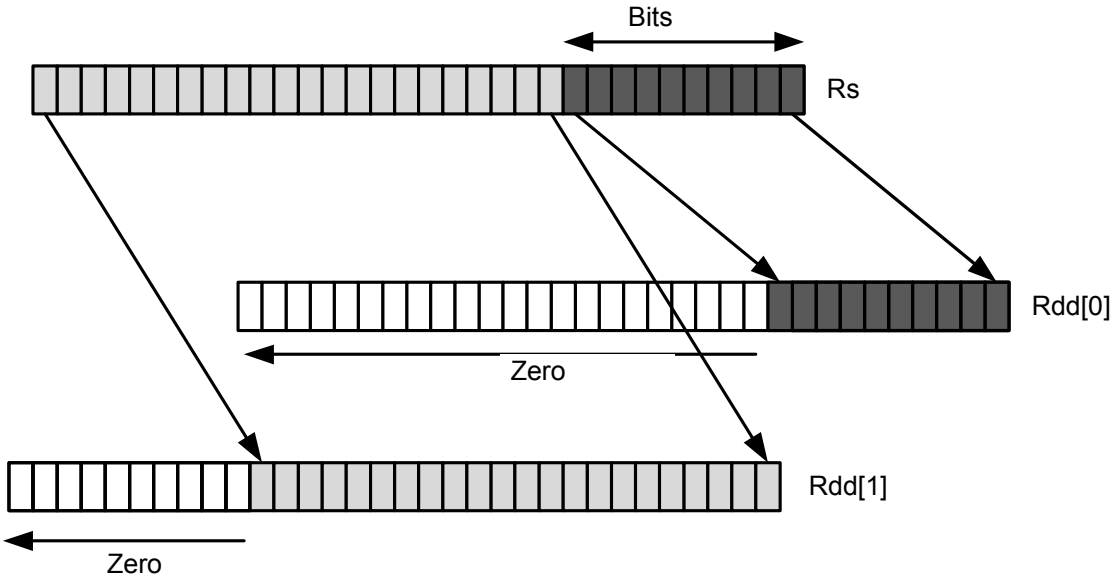
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=setbit(Rs,#u5)
1	0	0	0	1	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=clrbit(Rs,#u5)
1	0	0	0	1	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=togglebit(Rs,#u5)
ICLASS				RegType				Maj			s5					Parse			t5						Min			d5					
1	1	0	0	0	1	1	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=setbit(Rs,Rt)
1	1	0	0	0	1	1	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=clrbit(Rs,Rt)
1	1	0	0	0	1	1	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=togglebit(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
MinOp	Minor Opcode
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type
RegType	Register Type

Split bitfield

Split the bitfield in a register into upper and lower parts of variable size. The lower part is placed in the lower word of a destination register pair, and the upper part is placed in the upper word of the destination. An immediate value or register Rt is used to determine the bit position of the split.



Syntax	Behavior
`Rdd=bitsplit(Rs,#u5)`	`Rdd.w[1]=(Rs>>#u);` `Rdd.w[0]=zxt`$_{\#u->32}$`(Rs);`

`Rdd=bitsplit(Rs,Rt)`	`shamt = zxt`$_{5->32}$`(Rt);`
	`Rdd.w[1]=(Rs>>shamt);`
	`Rdd.w[0]=zxt`$_{shamt->32}$`(Rs);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=bitsplit(Rs,#u5)`	`Word64 Q6_P_bitsplit_RI(Word32 Rs, Word32 Iu5)`
`Rdd=bitsplit(Rs,Rt)`	`Word64 Q6_P_bitsplit_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=bitsplit(Rs,#u5)
ICLASS				RegType							s5					Parse			t5								d5					
1	1	0	1	0	1	0	0	**-**	**-**	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	**-**	**-**	**-**	**d**	**d**	**d**	**d**	**d**	Rdd=bitsplit(Rs,Rt)

Field name	Description
RegType	Register Type
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Table index

The table index instruction supports fast lookup tables where the index into the table is stored in a bit-field. The instruction forms the address of a table element by extracting the bit-field and inserting it into the appropriate bits of a pointer to the table element.

Tables are defined to contain entries of bytes, halfwords, words, or doublewords. The table must be aligned to a power-of-2 size greater than or equal to the table size. For example, a 4Kbyte table should be aligned to a 4Kbyte boundary. This instruction supports tables with a maximum of 32K table entries.

Register Rx contains a pointer to within the table. Register Rs contains a field to be extracted and used as a table index. This instruction first extracts the field from register Rs and then inserts it into register Rx. The insertion point is bit 0 for tables of bytes, bit 1 for tables of halfwords, bit 2 for tables of words, and bit 3 for tables of doublewords.

In the assembly syntax, the width and offset values represent the field in Rs to be extracted. Unsigned constants should be used to specify the width and offsets in assembly. In the encoded instruction, however, these values are adjusted by the assembler as follows.

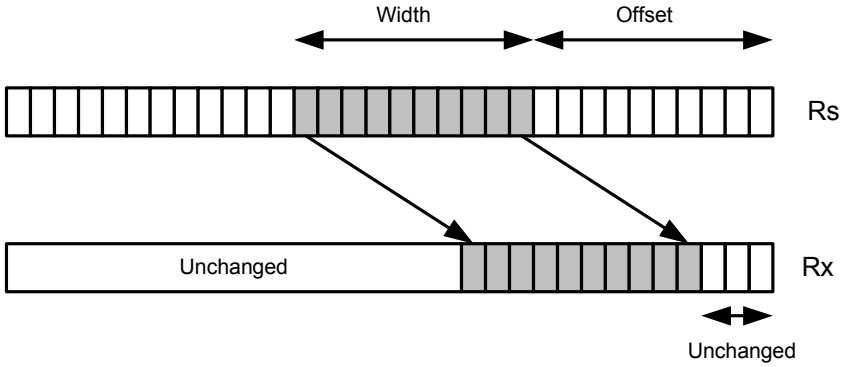
For tableidxb, no adjustment is necessary.

For tableidxh, the assembler encodes offset-1 in the signed immediate field.

For tableidxw, the assembler encodes offset-2 in the signed immediate field.

For tableidxd, the assembler encodes offset-3 in the signed immediate field.

Rx=TABLEIDXD(Rs,#width,#offset)



Syntax	Behavior

`Rx=tableidxb(Rs,#u4,#S6):raw`	`width=#u;`
	`offset=#S;`
	`field = Rs[(width+offset-1):offset];`
	`Rx[(width-1+0):0]=field;`

`Rx=tableidxb(Rs,#u4,#U5)`	`Assembler mapped to:` `"Rx=tableidxb(Rs,#u4,#U5):raw"`

`Rx=tableidxd(Rs,#u4,#S6):raw`	`width=#u;`
	`offset=#S+3;`
	`field = Rs[(width+offset-1):offset];`
	`Rx[(width-1+3):3]=field;`

`Rx=tableidxd(Rs,#u4,#U5)`	`Assembler mapped to: "Rx=tableidxd(Rs,#u4,#U5-3):raw"`

`Rx=tableidxh(Rs,#u4,#S6):raw`	`width=#u;`
	`offset=#S+1;`
	`field = Rs[(width+offset-1):offset];`
	`Rx[(width-1+1):1]=field;`

`Rx=tableidxh(Rs,#u4,#U5)`	`Assembler mapped to: "Rx=tableidxh(Rs,#u4,#U5-1):raw"`

`Rx=tableidxw(Rs,#u4,#S6):raw`	`width=#u;`
	`offset=#S+2;`
	`field = Rs[(width+offset-1):offset];`
	`Rx[(width-1+2):2]=field;`

`Rx=tableidxw(Rs,#u4,#U5)`	`Assembler mapped to: "Rx=tableidxw(Rs,#u4,#U5-2):raw"`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx=tableidxb(Rs,#u4,#U5)`	`Word32 Q6_R_tableidxb_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5)`
`Rx=tableidxd(Rs,#u4,#U5)`	`Word32 Q6_R_tableidxd_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5)`
`Rx=tableidxh(Rs,#u4,#U5)`	`Word32 Q6_R_tableidxh_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5)`
`Rx=tableidxw(Rs,#u4,#U5)`	`Word32 Q6_R_tableidxw_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	0	0	0	0	1	1	1	0	0	i	s	s	s	s	s	P	P	I	I	I	I	I	I	i	i	i	x	x	x	x	x	Rx=tableidxb(Rs,#u4,#S6):raw
1	0	0	0	0	1	1	1	0	1	i	s	s	s	s	s	P	P	I	I	I	I	I	I	i	i	i	x	x	x	x	x	Rx=tableidxh(Rs,#u4,#S6):raw
1	0	0	0	0	1	1	1	1	0	i	s	s	s	s	s	P	P	I	I	I	I	I	I	i	i	i	x	x	x	x	x	Rx=tableidxw(Rs,#u4,#S6):raw
1	0	0	0	0	1	1	1	1	1	i	s	s	s	s	s	P	P	I	I	I	I	I	I	i	i	i	x	x	x	x	x	Rx=tableidxd(Rs,#u4,#S6):raw

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
x5	Field to encode register x
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

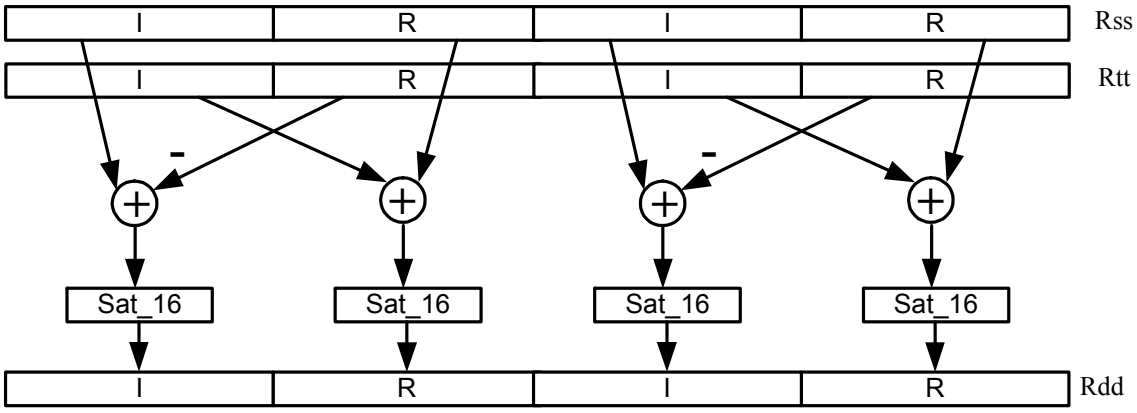
11.12.3 XTYPE/COMPLEX

The XTYPE/COMPLEX instruction subclass includes instructions which are for complex math, using imaginary values.

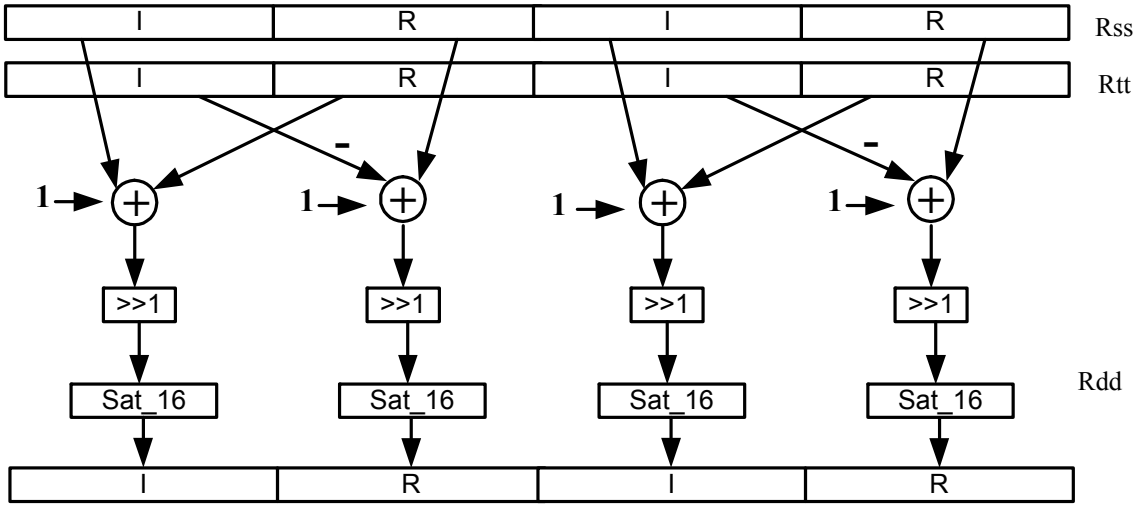
Complex add/sub halfwords

Cross vector add-sub or sub-add used to perform X+jY and X-jY complex operations. Each 16-bit result is saturated to 16-bits.

Rdd=vxaddsubh(Rss,Rtt):sat



Rdd=vxsubaddh(Rss,Rt):rnd:>>1:sat



Syntax	Behavior

`Rdd=vxaddsubh(Rss,Rtt):rnd:>>1:sat`	`Rdd.h[0]=sat_16((Rss.h[0]+Rtt.h[1]+1)>>1);`
	`Rdd.h[1]=sat_16((Rss.h[1]-Rtt.h[0]+1)>>1);`
	`Rdd.h[2]=sat_16((Rss.h[2]+Rtt.h[3]+1)>>1);`
	`Rdd.h[3]=sat_16((Rss.h[3]-Rtt.h[2]+1)>>1);`

`Rdd=vxaddsubh(Rss,Rtt):sat`	`Rdd.h[0]=sat_16(Rss.h[0]+Rtt.h[1]);`
	`Rdd.h[1]=sat_16(Rss.h[1]-Rtt.h[0]);`
	`Rdd.h[2]=sat_16(Rss.h[2]+Rtt.h[3]);`
	`Rdd.h[3]=sat_16(Rss.h[3]-Rtt.h[2]);`

`Rdd=vxsubaddh(Rss,Rtt):rnd:>>1:sat`	`Rdd.h[0]=sat_16((Rss.h[0]-Rtt.h[1]+1)>>1);`
	`Rdd.h[1]=sat_16((Rss.h[1]+Rtt.h[0]+1)>>1);`
	`Rdd.h[2]=sat_16((Rss.h[2]-Rtt.h[3]+1)>>1);`
	`Rdd.h[3]=sat_16((Rss.h[3]+Rtt.h[2]+1)>>1);`

`Rdd=vxsubaddh(Rss,Rtt):sat`	`Rdd.h[0]=sat_16(Rss.h[0]-Rtt.h[1]);`
	`Rdd.h[1]=sat_16(Rss.h[1]+Rtt.h[0]);`
	`Rdd.h[2]=sat_16(Rss.h[2]-Rtt.h[3]);`
	`Rdd.h[3]=sat_16(Rss.h[3]+Rtt.h[2]);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vxaddsubh(Rss,Rtt):rnd:>>1:sat`	`Word64 Q6_P_vxaddsubh_PP_rnd_rs1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vxaddsubh(Rss,Rtt):sat`	`Word64 Q6_P_vxaddsubh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vxsubaddh(Rss,Rtt):rnd:>>1:sat`	`Word64 Q6_P_vxsubaddh_PP_rnd_rs1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vxsubaddh(Rss,Rtt):sat`	`Word64 Q6_P_vxsubaddh_PP_sat(Word64 Rss, Word64 Rtt)`

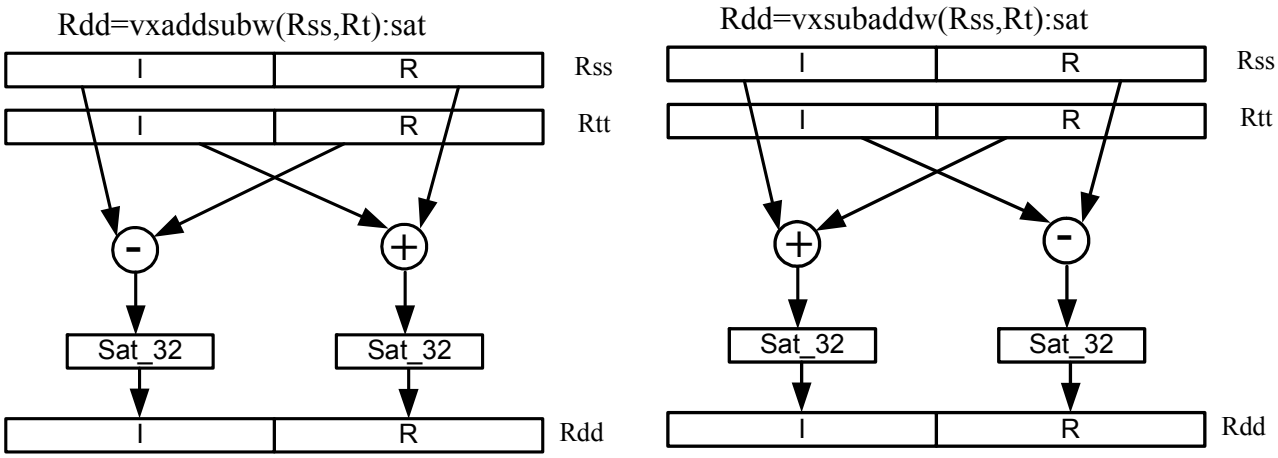
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	0	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	0	d	d	d	d	d	Rdd=vxaddsubh(Rss,Rtt):sat
1	1	0	0	0	0	0	1	0	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	0	d	d	d	d	d	Rdd=vxsubaddh(Rss,Rtt):sat
1	1	0	0	0	0	0	1	1	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	0	-	d	d	d	d	d	Rdd=vxaddsubh(Rss,Rtt):rnd:>>1:sat
1	1	0	0	0	0	0	1	1	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	0	1	-	d	d	d	d	d	Rdd=vxsubaddh(Rss,Rtt):rnd:>>1:sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Complex add/sub words

Cross vector add-sub or sub-add used to perform X+jY and X-jY complex operations. Each 32-bit result is saturated to 32-bits.



Syntax	Behavior
`Rdd=vxaddsubw(Rss,Rtt):sat`	`Rdd.w[0]=sat_32(Rss.w[0]+Rtt.w[1]);` `Rdd.w[1]=sat_32(Rss.w[1]-Rtt.w[0]);`
`Rdd=vxsubaddw(Rss,Rtt):sat`	`Rdd.w[0]=sat_32(Rss.w[0]-Rtt.w[1]);` `Rdd.w[1]=sat_32(Rss.w[1]+Rtt.w[0]);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vxaddsubw(Rss,Rtt):sat`	`Word64 Q6_P_vxaddsubw_PP_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vxsubaddw(Rss,Rtt):sat`	`Word64 Q6_P_vxsubaddw_PP_sat(Word64 Rss, Word64 Rtt)`

Encoding

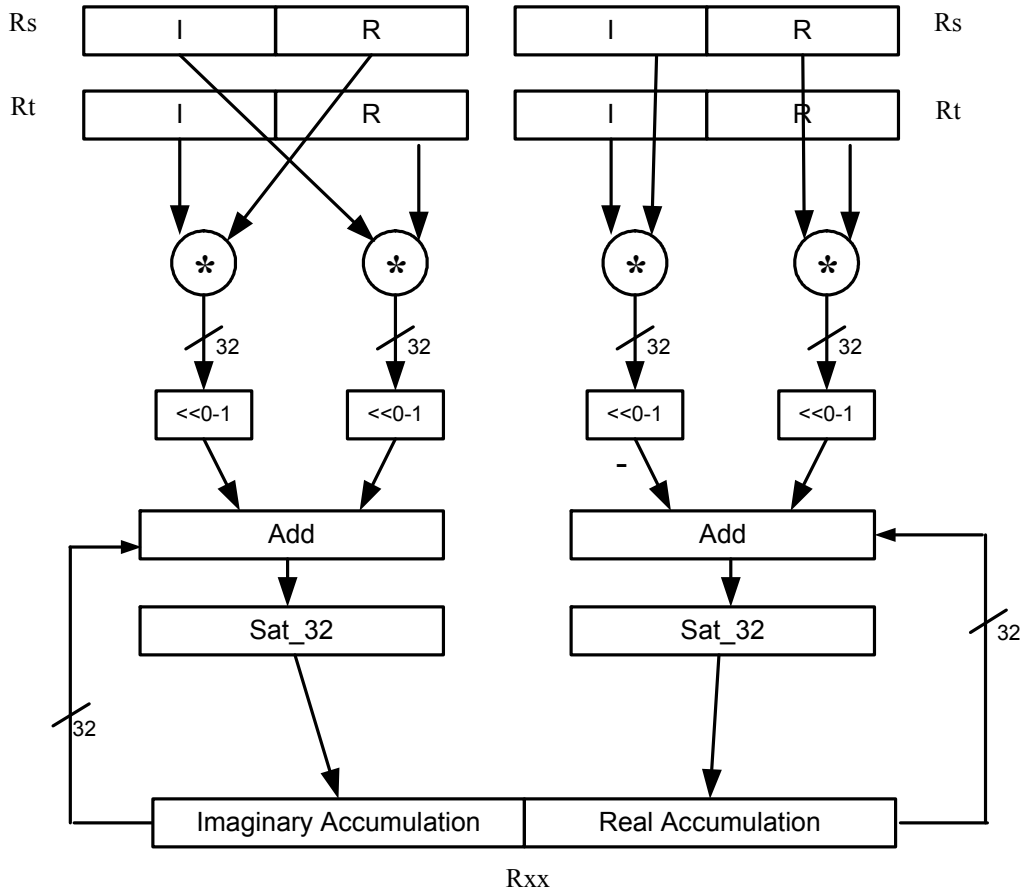
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vxaddsubw(Rss,Rtt):sat
1	1	0	0	0	0	0	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vxsubaddw(Rss,Rtt):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Complex multiply

Multiply complex values Rs and Rt. The inputs have a real 16-bit value in the low halfword and an imaginary 16-bit value in the high halfword. Optionally, scale the result by 0-1 bits. Optionally, add a complex accumulator. Saturate the real and imaginary portions to 32-bits. The output has a real 32-bit value in the low word and an imaginary 32-bit value in the high word. The Rt input can be optionally conjugated. Another option is that the result can be subtracted from the destination rather than accumulated.

Rxx+=cmpy(Rs,Rt):sat



Syntax	Behavior
`Rdd=cmpy(Rs,Rt)[:<<1]:sat`	`Rdd.w[1]=sat_32((Rs.h[1] * Rt.h[0])[<<1] + (Rs.h[0] * Rt.h[1])[<<1]);` `Rdd.w[0]=sat_32((Rs.h[0] * Rt.h[0])[<<1] - (Rs.h[1] * Rt.h[1])[<<1]);`

Syntax	Behavior
`Rdd=cmpy(Rs,Rt*)[:<<1]:sat`	`Rdd.w[1]=sat_32((Rs.h[1] * Rt.h[0])[<<1] - (Rs.h[0] * Rt.h[1])[<<1]); Rdd.w[0]=sat_32((Rs.h[0] * Rt.h[0])[<<1] + (Rs.h[1] * Rt.h[1])[<<1]);`
`Rxx+=cmpy(Rs,Rt)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (Rs.h[1] * Rt.h[0])[<<1] + (Rs.h[0] * Rt.h[1])[<<1]); Rxx.w[0]=sat_32(Rxx.w[0] + (Rs.h[0] * Rt.h[0])[<<1] - (Rs.h[1] * Rt.h[1])[<<1]);`
`Rxx+=cmpy(Rs,Rt*)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (Rs.h[1] * Rt.h[0])[<<1] - (Rs.h[0] * Rt.h[1])[<<1]); Rxx.w[0]=sat_32(Rxx.w[0] + (Rs.h[0] * Rt.h[0])[<<1] + (Rs.h[1] * Rt.h[1])[<<1]);`
`Rxx-=cmpy(Rs,Rt)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] - ((Rs.h[1] * Rt.h[0])[<<1] + (Rs.h[0] * Rt.h[1])[<<1])); Rxx.w[0]=sat_32(Rxx.w[0] - ((Rs.h[0] * Rt.h[0])[<<1] - (Rs.h[1] * Rt.h[1])[<<1]));`
`Rxx-=cmpy(Rs,Rt*)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] - ((Rs.h[1] * Rt.h[0])[<<1] - (Rs.h[0] * Rt.h[1])[<<1])); Rxx.w[0]=sat_32(Rxx.w[0] - ((Rs.h[0] * Rt.h[0])[<<1] + (Rs.h[1] * Rt.h[1])[<<1]));`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=cmpy(Rs,Rt):<<1:sat`	`Word64 Q6_P_cmpy_RR_s1_sat(Word32 Rs, Word32 Rt)`
`Rdd=cmpy(Rs,Rt):sat`	`Word64 Q6_P_cmpy_RR_sat(Word32 Rs, Word32 Rt)`
`Rdd=cmpy(Rs,Rt*):<<1:sat`	`Word64 Q6_P_cmpy_RR_conj_s1_sat(Word32 Rs, Word32 Rt)`
`Rdd=cmpy(Rs,Rt*):sat`	`Word64 Q6_P_cmpy_RR_conj_sat(Word32 Rs, Word32 Rt)`
`Rxx+=cmpy(Rs,Rt):<<1:sat`	`Word64 Q6_P_cmpyacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=cmpy(Rs,Rt):sat`	`Word64 Q6_P_cmpyacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=cmpy(Rs,Rt*):<<1:sat`	`Word64 Q6_P_cmpyacc_RR_conj_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`

`Rxx+=cmpy(Rs,Rt*):sat`	`Word64 Q6_P_cmpyacc_RR_conj_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=cmpy(Rs,Rt):<<1:sat`	`Word64 Q6_P_cmpynac_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=cmpy(Rs,Rt):sat`	`Word64 Q6_P_cmpynac_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=cmpy(Rs,Rt*):<<1:sat`	`Word64 Q6_P_cmpynac_RR_conj_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=cmpy(Rs,Rt*):sat`	`Word64 Q6_P_cmpynac_RR_conj_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`

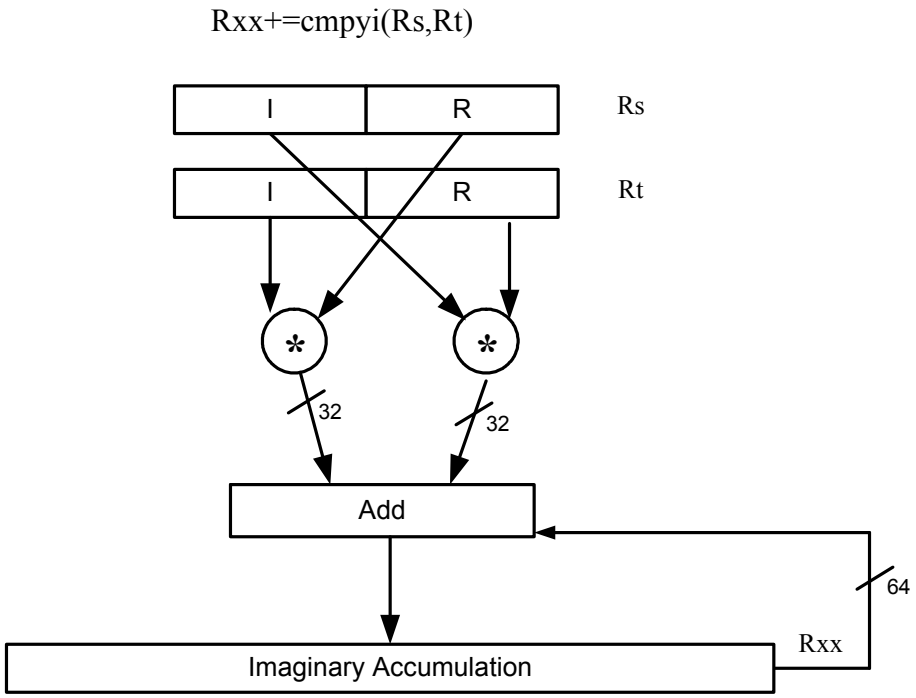
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=cmpy(Rs,Rt)[:<<N]:sat
1	1	1	0	0	1	0	1	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=cmpy(Rs,Rt*)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpy(Rs,Rt)[:<<N]:sat
1	1	1	0	0	1	1	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx-=cmpy(Rs,Rt)[:<<N]:sat
1	1	1	0	0	1	1	1	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpy(Rs,Rt*)[:<<N]:sat
1	1	1	0	0	1	1	1	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx-=cmpy(Rs,Rt*)[:<<N]:sat

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Complex multiply real or imaginary

Multiply complex values Rs and Rt. The inputs have a real 16-bit value in the low halfword and an imaginary 16-bit value in the high halfword. Take either the real or imaginary result and optionally accumulate with a 64-bit destination.



Syntax	Behavior
`Rdd=cmpyi(Rs,Rt)`	`Rdd = (Rs.h[1] * Rt.h[0]) + (Rs.h[0] * Rt.h[1]);`
`Rdd=cmpyr(Rs,Rt)`	`Rdd = (Rs.h[0] * Rt.h[0]) - (Rs.h[1] * Rt.h[1]);`
`Rxx+=cmpyi(Rs,Rt)`	`Rxx = Rxx + (Rs.h[1] * Rt.h[0]) + (Rs.h[0] * Rt.h[1]);`
`Rxx+=cmpyr(Rs,Rt)`	`Rxx = Rxx + (Rs.h[0] * Rt.h[0]) - (Rs.h[1] * Rt.h[1]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=cmpyi(Rs,Rt)`	`Word64 Q6_P_cmpyi_RR(Word32 Rs, Word32 Rt)`
`Rdd=cmpyr(Rs,Rt)`	`Word64 Q6_P_cmpyr_RR(Word32 Rs, Word32 Rt)`
`Rxx+=cmpyi(Rs,Rt)`	`Word64 Q6_P_cmpyiacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=cmpyr(Rs,Rt)`	`Word64 Q6_P_cmpyracc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`

Encoding

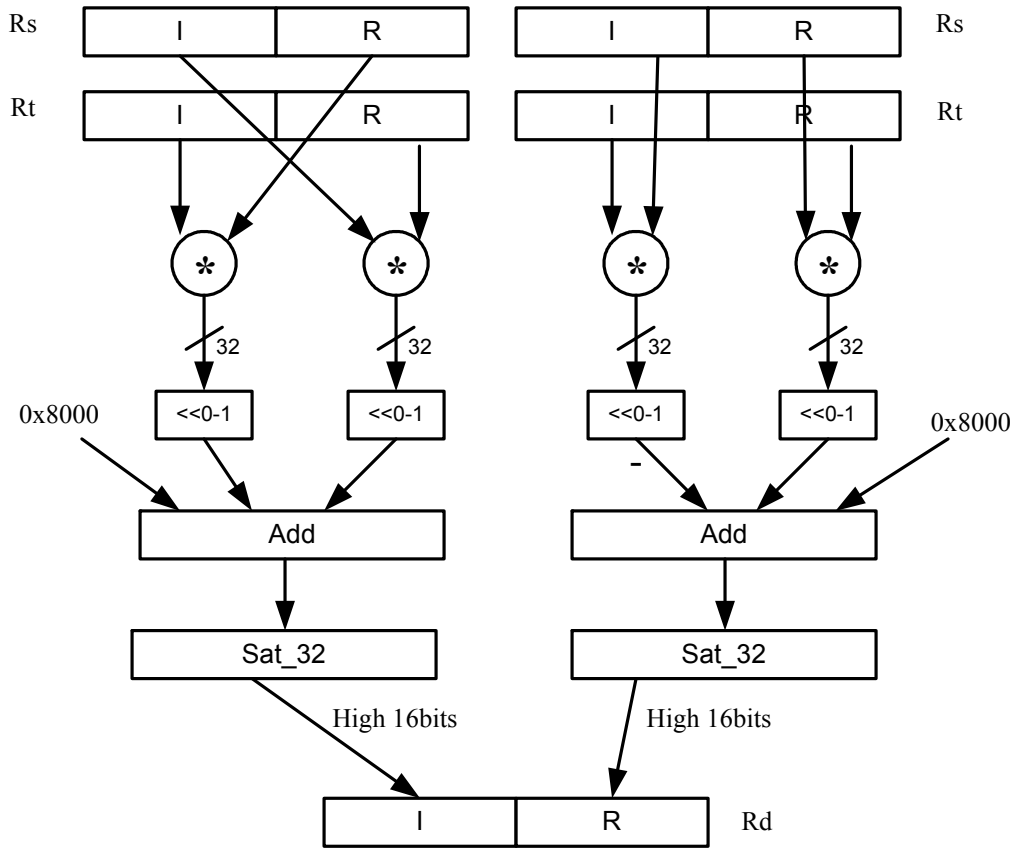
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5						
1	1	1	0	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=cmpyi(Rs,Rt)
1	1	1	0	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=cmpyr(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpyi(Rs,Rt)
1	1	1	0	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=cmpyr(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Complex multiply with round and pack

Multiply complex values Rs and Rt. The inputs have a real 16-bit value in the low halfword and an imaginary 16-bit value in the high halfword. The Rt input is optionally conjugated. The multiplier results are optionally scaled by 0-1 bits. A rounding constant is added to each real and imaginary sum. The real and imaginary parts are individually saturated to 32bits. The upper 16-bits of each 32-bit results are packed in a 32-bit destination register.

Rd=cmpy(Rs,Rt):rnd:sat



Syntax	Behavior
`Rd=cmpy(Rs,Rt)[:<<1]:rnd:sat`	`Rd.h[1]=(sat_32((Rs.h[1] * Rt.h[0])[<<1] + (Rs.h[0] * Rt.h[1])[<<1] + 0x8000)).h[1]; Rd.h[0]=(sat_32((Rs.h[0] * Rt.h[0])[<<1] - (Rs.h[1] * Rt.h[1])[<<1] + 0x8000)).h[1];`
`Rd=cmpy(Rs,Rt*)[:<<1]:rnd:sat`	`Rd.h[1]=(sat_32((Rs.h[1] * Rt.h[0])[<<1] - (Rs.h[0] * Rt.h[1])[<<1] + 0x8000)).h[1]; Rd.h[0]=(sat_32((Rs.h[0] * Rt.h[0])[<<1] + (Rs.h[1] * Rt.h[1])[<<1] + 0x8000)).h[1];`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=cmpy(Rs,Rt):<<1:rnd:sat`	`Word32 Q6_R_cmpy_RR_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=cmpy(Rs,Rt):rnd:sat`	`Word32 Q6_R_cmpy_RR_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=cmpy(Rs,Rt*):<<1:rnd:sat`	`Word32 Q6_R_cmpy_RR_conj_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=cmpy(Rs,Rt*):rnd:sat`	`Word32 Q6_R_cmpy_RR_conj_rnd_sat(Word32 Rs, Word32 Rt)`

Encoding

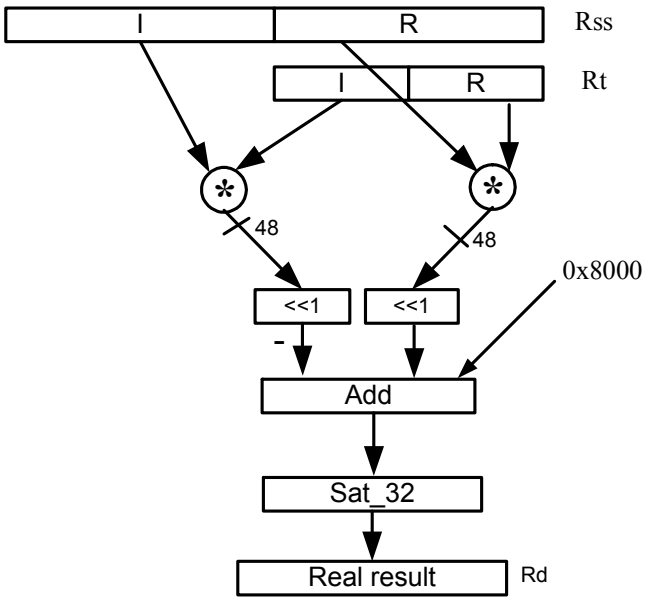
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	1	0	1	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=cmpy(Rs,Rt)[:<<N]:rnd:sat
1	1	1	0	1	1	0	1	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=cmpy(Rs,Rt*)[:<<N]:rnd:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

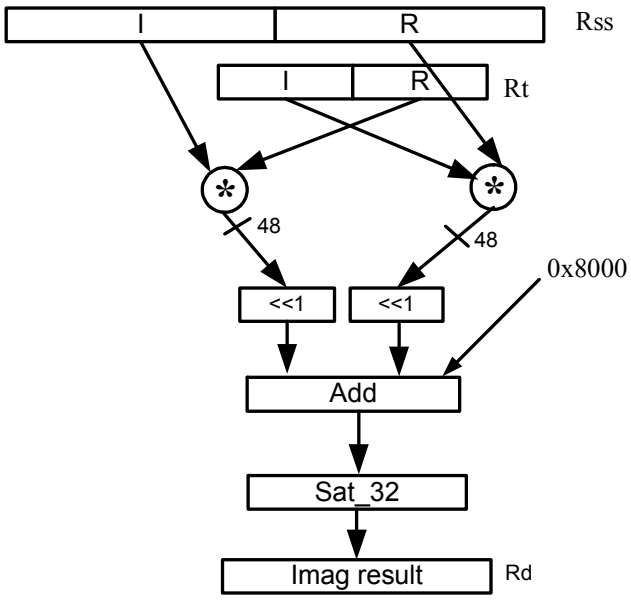
Complex multiply 32x16

Multiply 32 by 16 bit complex values Rss and Rt. The inputs have a real value in the low part of a register and the imaginary value in the upper part. The multiplier results are scaled by 1 bit and accumulated with a rounding constant. The result is saturated to 32bits.

Rd=cmpyrwh(Rss,Rt):<<1:rnd:sat



Rd=cmpyiwh(Rss,Rt):<<1:rnd:sat



Syntax	Behavior
`Rd=cmpyiwh(Rss,Rt):<<1:rnd:sat`	`Rd = sat_32(( (Rss.w[0] * Rt.h[1]) + (Rss.w[1] * Rt.h[0]) + 0x4000)>>15);`
`Rd=cmpyiwh(Rss,Rt*):<<1:rnd:sat`	`Rd = sat_32(( (Rss.w[1] * Rt.h[0]) - (Rss.w[0] * Rt.h[1]) + 0x4000)>>15);`
`Rd=cmpyrwh(Rss,Rt):<<1:rnd:sat`	`Rd = sat_32(( (Rss.w[0] * Rt.h[0]) - (Rss.w[1] * Rt.h[1]) + 0x4000)>>15);`
`Rd=cmpyrwh(Rss,Rt*):<<1:rnd:sat`	`Rd = sat_32(( (Rss.w[0] * Rt.h[0]) + (Rss.w[1] * Rt.h[1]) + 0x4000)>>15);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=cmpyiwh(Rss,Rt):<<1:rnd:sat`	`Word32 Q6_R_cmpyiwh_PR_s1_rnd_sat(Word64 Rss, Word32 Rt)`
`Rd=cmpyiwh(Rss,Rt*):<<1:rnd:sat`	`Word32 Q6_R_cmpyiwh_PR_conj_s1_rnd_sat(Word64 Rss, Word32 Rt)`
`Rd=cmpyrwh(Rss,Rt):<<1:rnd:sat`	`Word32 Q6_R_cmpyrwh_PR_s1_rnd_sat(Word64 Rss, Word32 Rt)`
`Rd=cmpyrwh(Rss,Rt*):<<1:rnd:sat`	`Word32 Q6_R_cmpyrwh_PR_conj_s1_rnd_sat(Word64 Rss, Word32 Rt)`

Encoding

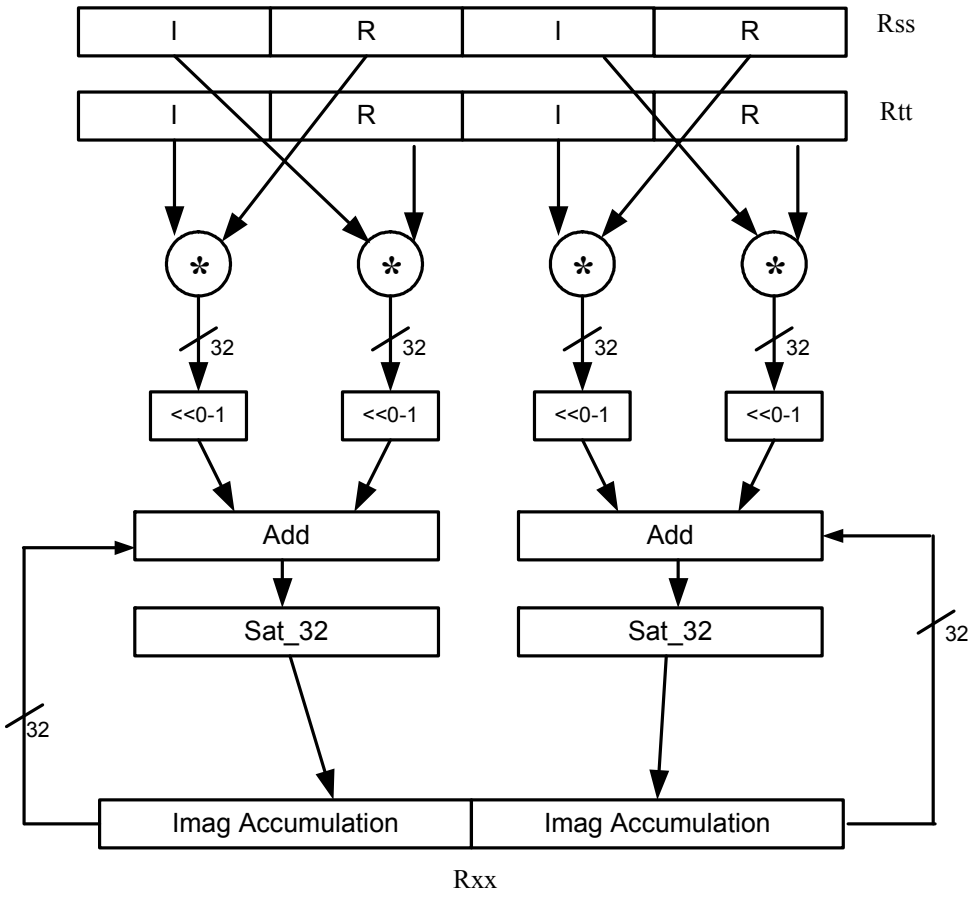
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					Min			d5					
1	1	0	0	0	1	0	1	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	0	d	d	d	d	d	Rd=cmpyiwh(Rss,Rt):<<1:rnd:sat
1	1	0	0	0	1	0	1	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	0	1	d	d	d	d	d	Rd=cmpyiwh(Rss,Rt*):<<1:rnd:sat
1	1	0	0	0	1	0	1	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	0	d	d	d	d	d	Rd=cmpyrwh(Rss,Rt):<<1:rnd:sat
1	1	0	0	0	1	0	1	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	1	d	d	d	d	d	Rd=cmpyrwh(Rss,Rt*):<<1:rnd:sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Min	Minor Opcode
RegType	Register Type

Vector complex multiply real or imaginary

The inputs Rss and Rtt are a vector of two complex values. Each complex value is composed of a 16-bit imaginary portion in the upper halfword and a 16-bit real portion in the lower halfword. Generate two complex results, either the real result or the imaginary result. These results are optionally shifted left by 0-1 bits, and optionally accumulated with the destination register.

Rxx+=vcmpyi(Rss,Rtt):sat



Syntax	Behavior
`Rdd=vcmpyi(Rss,Rtt)[:<<1]:sat`	`Rdd.w[0]=sat_32((Rss.h[1] * Rtt.h[0]) + (Rss.h[0] * Rtt.h[1])[<<1]);` `Rdd.w[1]=sat_32((Rss.h[3] * Rtt.h[2]) + (Rss.h[2] * Rtt.h[3])[<<1]);`
`Rdd=vcmpyr(Rss,Rtt)[:<<1]:sat`	`Rdd.w[0]=sat_32((Rss.h[0] * Rtt.h[0]) - (Rss.h[1] * Rtt.h[1])[<<1]);` `Rdd.w[1]=sat_32((Rss.h[2] * Rtt.h[2]) - (Rss.h[3] * Rtt.h[3])[<<1]);`
`Rxx+=vcmpyi(Rss,Rtt):sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rss.h[1] * Rtt.h[0]) + (Rss.h[0] * Rtt.h[1])<<0);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rss.h[3] * Rtt.h[2]) + (Rss.h[2] * Rtt.h[3])<<0);`
`Rxx+=vcmpyr(Rss,Rtt):sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0]) - (Rss.h[1] * Rtt.h[1])<<0);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2]) - (Rss.h[3] * Rtt.h[3])<<0);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vcmpyi(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vcmpyi_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vcmpyi(Rss,Rtt):sat`	`Word64 Q6_P_vcmpyi_PP_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vcmpyr(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vcmpyr_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vcmpyr(Rss,Rtt):sat`	`Word64 Q6_P_vcmpyr_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+=vcmpyi(Rss,Rtt):sat`	`Word64 Q6_P_vcmpyiacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vcmpyr(Rss,Rtt):sat`	`Word64 Q6_P_vcmpyracc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vcmpyr(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vcmpyi(Rss,Rtt)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vcmpyr(Rss,Rtt):sat
1	1	1	0	1	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vcmpyi(Rss,Rtt):sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector complex conjugate

Perform a vector complex conjugate of both complex values in vector Rss. This is done by negating the imaginary halfwords, and placing the result in destination Rdd.

Syntax	Behavior

`Rdd=vconj(Rss):sat`	`Rdd.h[1]=sat_16(-Rss.h[1]);`
	`Rdd.h[0]=Rss.h[0];`
	`Rdd.h[3]=sat_16(-Rss.h[3]);`
	`Rdd.h[2]=Rss.h[2];`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vconj(Rss):sat`	`Word64 Q6_P_vconj_P_sat(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vconj(Rss):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Vector complex rotate

Take the least significant bits of Rt, and use these bits to rotate each of the two complex values in the source vector a multiple of 90 degrees. Bits 0 and 1 control the rotation factor for word 0, and bits 2 and 3 control the rotation factor for word 1.

If the rotation control bits are 0, the rotation is 0: the real and imaginary halves of the source appear unchanged and unmoved in the destination.

If the rotation control bits are 1, the rotation is -pi/2: the real half of the destination gets the imaginary half of the source, and the imaginary half of the destination gets the negative real half of the source.

If the rotation control bits are 2, the rotation is pi/2: the real half of the destination gets the negative imaginary half of the source, and the imaginary half of the destination gets the real half of the source.

If the rotation control bits are 3, the rotation is pi: the real half of the destination gets the negative real half of the source, and the imaginary half of the destination gets the negative imaginary half of the source.

Syntax

```
Rdd=vcrotate(Rss,Rt)
```

Behavior

```
tmp = Rt[1:0];
if (tmp == 0) {
    Rdd.h[0]=Rss.h[0];
    Rdd.h[1]=Rss.h[1];
} else if (tmp == 1) {
    Rdd.h[0]=Rss.h[1];
    Rdd.h[1]=sat_16(-Rss.h[0]);
} else if (tmp == 2) {
    Rdd.h[0]=sat_16(-Rss.h[1]);
    Rdd.h[1]=Rss.h[0];
} else {
    Rdd.h[0]=sat_16(-Rss.h[0]);
    Rdd.h[1]=sat_16(-Rss.h[1]);
};
tmp = Rt[3:2];
if (tmp == 0) {
    Rdd.h[2]=Rss.h[2];
    Rdd.h[3]=Rss.h[3];
} else if (tmp == 1) {
    Rdd.h[2]=Rss.h[3];
    Rdd.h[3]=sat_16(-Rss.h[2]);
} else if (tmp == 2) {
    Rdd.h[2]=sat_16(-Rss.h[3]);
    Rdd.h[3]=Rss.h[2];
} else {
    Rdd.h[2]=sat_16(-Rss.h[2]);
    Rdd.h[3]=sat_16(-Rss.h[3]);
};
```

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

```
Rdd=vcrotate(Rss,Rt)          Word64 Q6_P_vcrotate_PR(Word64 Rss, Word32 Rt)
```

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vcrotate(Rss,Rt)

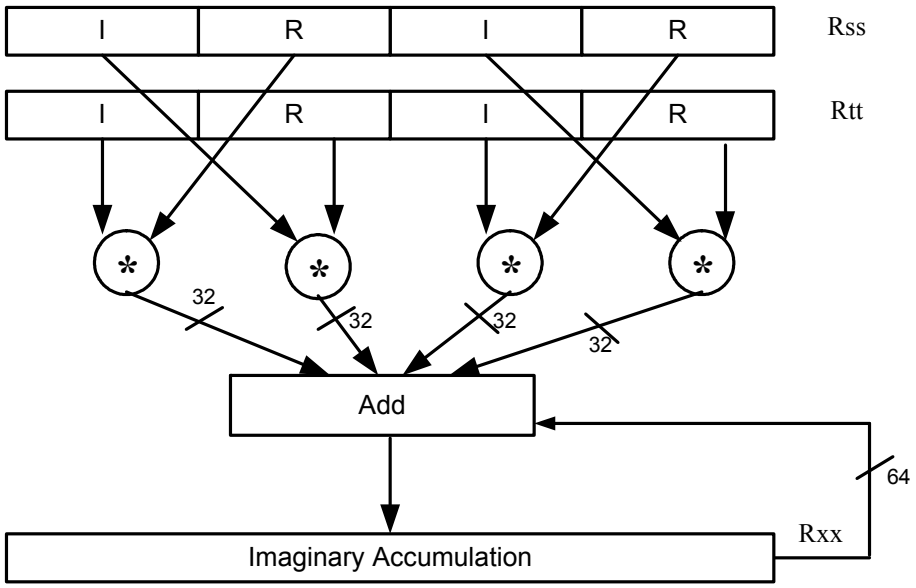
Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Vector reduce complex multiply real or imaginary

The input vectors are two packed complex values, each with a real low halfword and imaginary high halfword. Compute either the real or imaginary products, add the intermediate results together and optionally accumulate with the destination. The Rtt input is optionally conjugated (negate the imaginary portion) before multiplication.

Using vrcmpyr and vrcmpyi, it is possible to sustain an average of one full complex multiply per cycle in a complex FIR, while also keeping both the real and imaginary accumulators in full precision 64-bit values.

Rxx+=vrcmpyi(Rss,Rtt)



Syntax	Behavior
`Rdd=vrcmpyi(Rss,Rtt)`	`Rdd = (Rss.h[1] * Rtt.h[0]) + (Rss.h[0] * Rtt.h[1]) + (Rss.h[3] * Rtt.h[2]) + (Rss.h[2] * Rtt.h[3]);`
`Rdd=vrcmpyi(Rss,Rtt*)`	`Rdd = (Rss.h[1] * Rtt.h[0]) - (Rss.h[0] * Rtt.h[1]) + (Rss.h[3] * Rtt.h[2]) - (Rss.h[2] * Rtt.h[3]);`
`Rdd=vrcmpyr(Rss,Rtt)`	`Rdd = (Rss.h[0] * Rtt.h[0]) - (Rss.h[1] * Rtt.h[1]) + (Rss.h[2] * Rtt.h[2]) - (Rss.h[3] * Rtt.h[3]);`

Syntax	Behavior
`Rdd=vrcmpyr(Rss,Rtt*)`	`Rdd = (Rss.h[0] * Rtt.h[0]) + (Rss.h[1] * Rtt.h[1]) + (Rss.h[2] * Rtt.h[2]) + (Rss.h[3] * Rtt.h[3]);`
`Rxx+=vrcmpyi(Rss,Rtt)`	`Rxx = Rxx + (Rss.h[1] * Rtt.h[0]) + (Rss.h[0] * Rtt.h[1]) + (Rss.h[3] * Rtt.h[2]) + (Rss.h[2] * Rtt.h[3]);`
`Rxx+=vrcmpyi(Rss,Rtt*)`	`Rxx = Rxx + (Rss.h[1] * Rtt.h[0]) - (Rss.h[0] * Rtt.h[1]) + (Rss.h[3] * Rtt.h[2]) - (Rss.h[2] * Rtt.h[3]);`
`Rxx+=vrcmpyr(Rss,Rtt)`	`Rxx = Rxx + (Rss.h[0] * Rtt.h[0]) - (Rss.h[1] * Rtt.h[1]) + (Rss.h[2] * Rtt.h[2]) - (Rss.h[3] * Rtt.h[3]);`
`Rxx+=vrcmpyr(Rss,Rtt*)`	`Rxx = Rxx + (Rss.h[0] * Rtt.h[0]) + (Rss.h[1] * Rtt.h[1]) + (Rss.h[2] * Rtt.h[2]) + (Rss.h[3] * Rtt.h[3]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vrcmpyi(Rss,Rtt)`	`Word64 Q6_P_vrcmpyi_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vrcmpyi(Rss,Rtt*)`	`Word64 Q6_P_vrcmpyi_PP_conj(Word64 Rss, Word64 Rtt)`
`Rdd=vrcmpyr(Rss,Rtt)`	`Word64 Q6_P_vrcmpyr_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vrcmpyr(Rss,Rtt*)`	`Word64 Q6_P_vrcmpyr_PP_conj(Word64 Rss, Word64 Rtt)`
`Rxx+=vrcmpyi(Rss,Rtt)`	`Word64 Q6_P_vrcmpyiacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vrcmpyi(Rss,Rtt*)`	`Word64 Q6_P_vrcmpyiacc_PP_conj(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vrcmpyr(Rss,Rtt)`	`Word64 Q6_P_vrcmpyracc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vrcmpyr(Rss,Rtt*)`	`Word64 Q6_P_vrcmpyracc_PP_conj(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			d5					
1	1	1	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrcmpyi(Rss,Rtt)
1	1	1	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vrcmpyr(Rss,Rtt)
1	1	1	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrcmpyi(Rss,Rtt*)
1	1	1	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vrcmpyr(Rss,Rtt*)

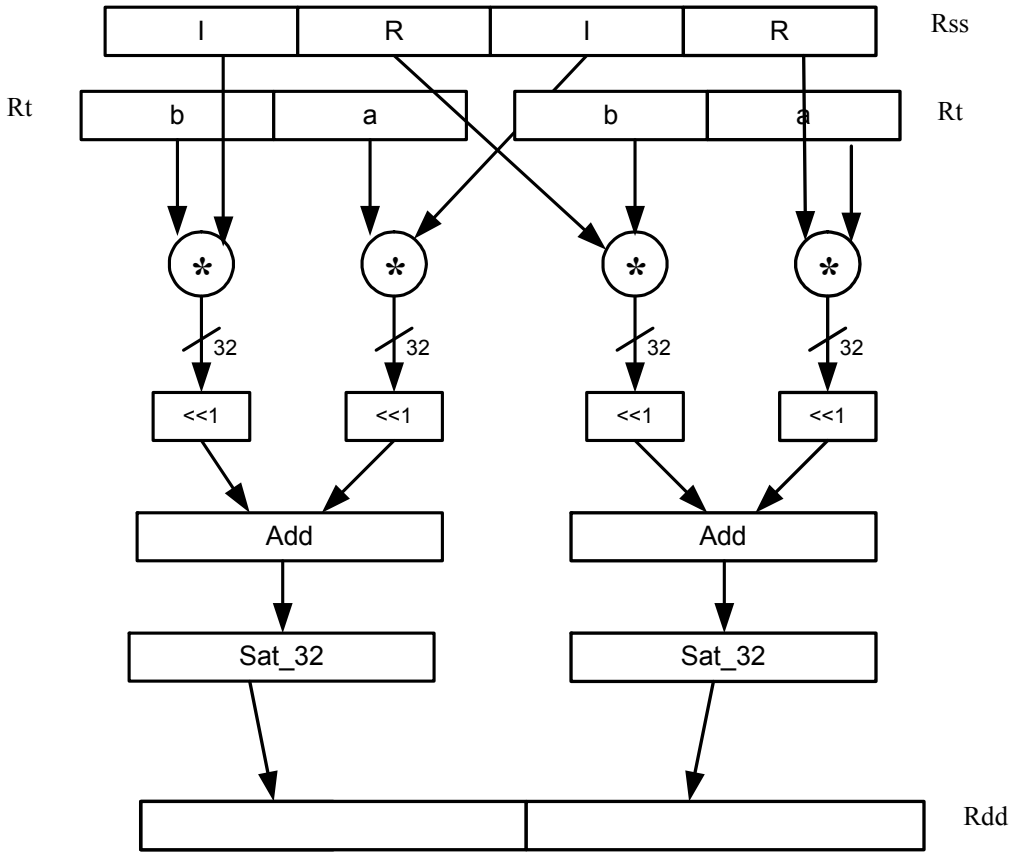
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrcmpyi(Rss,Rtt)
1	1	1	0	1	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vrcmpyr(Rss,Rtt)
1	1	1	0	1	0	1	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrcmpyi(Rss,Rtt*)
1	1	1	0	1	0	1	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vrcmpyr(Rss,Rtt*)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector reduce complex multiply by scalar

Multiply a complex number by a scalar. Rss contains two complex numbers. The real portions are each multiplied by two scalars contained in register Rt, scaled, summed, optionally accumulated, saturated, and stored in the lower word of Rdd. A similar operation is done on the two imaginary portions of Rss.

Rdd=vrcmpys(Rss,Rt):<<1:sat



Syntax	Behavior

Rdd=vrcmpys(Rss,Rt):<<1:sat	if ("Rt & 1") {
	Assembler mapped to:
	"Rdd=vrcmpys(Rss,Rtt):<<1:sat:raw:hi";
	} else {
	Assembler mapped to:
	"Rdd=vrcmpys(Rss,Rtt):<<1:sat:raw:lo";
	};

Rdd=vrcmpys(Rss,Rtt):<<1:sat:raw:hi	Rdd.w[1]=sat_32((Rss.h[1] * Rtt.w[1].h[0])<<1 + (Rss.h[3] * Rtt.w[1].h[1])<<1); Rdd.w[0]=sat_32((Rss.h[0] * Rtt.w[1].h[0])<<1 + (Rss.h[2] * Rtt.w[1].h[1])<<1);
Rdd=vrcmpys(Rss,Rtt):<<1:sat:raw:lo	Rdd.w[1]=sat_32((Rss.h[1] * Rtt.w[0].h[0])<<1 + (Rss.h[3] * Rtt.w[0].h[1])<<1); Rdd.w[0]=sat_32((Rss.h[0] * Rtt.w[0].h[0])<<1 + (Rss.h[2] * Rtt.w[0].h[1])<<1);

Rxx+=vrcmpys(Rss,Rt):<<1:sat	if ("Rt & 1") {
	Assembler mapped to:
	"Rxx+=vrcmpys(Rss,Rtt):<<1:sat:raw:hi";
	} else {
	Assembler mapped to:
	"Rxx+=vrcmpys(Rss,Rtt):<<1:sat:raw:lo";
	};

Rxx+=vrcmpys(Rss,Rtt):<<1:sat:raw:hi	Rxx.w[1]=sat_32(Rxx.w[1] + (Rss.h[1] * Rtt.w[1].h[0])<<1 + (Rss.h[3] * Rtt.w[1].h[1])<<1); Rxx.w[0]=sat_32(Rxx.w[0] + (Rss.h[0] * Rtt.w[1].h[0])<<1 + (Rss.h[2] * Rtt.w[1].h[1])<<1);
Rxx+=vrcmpys(Rss,Rtt):<<1:sat:raw:lo	Rxx.w[1]=sat_32(Rxx.w[1] + (Rss.h[1] * Rtt.w[0].h[0])<<1 + (Rss.h[3] * Rtt.w[0].h[1])<<1); Rxx.w[0]=sat_32(Rxx.w[0] + (Rss.h[0] * Rtt.w[0].h[0])<<1 + (Rss.h[2] * Rtt.w[0].h[1])<<1);

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

Rdd=vrcmpys(Rss,Rt):<<1:sat	Word64 Q6_P_vrcmpys_PR_s1_sat(Word64 Rss, Word32 Rt)
Rxx+=vrcmpys(Rss,Rt):<<1:sat	Word64 Q6_P_vrcmpysacc_PR_s1_sat(Word64 Rxx, Word64 Rss, Word32 Rt)

Encoding

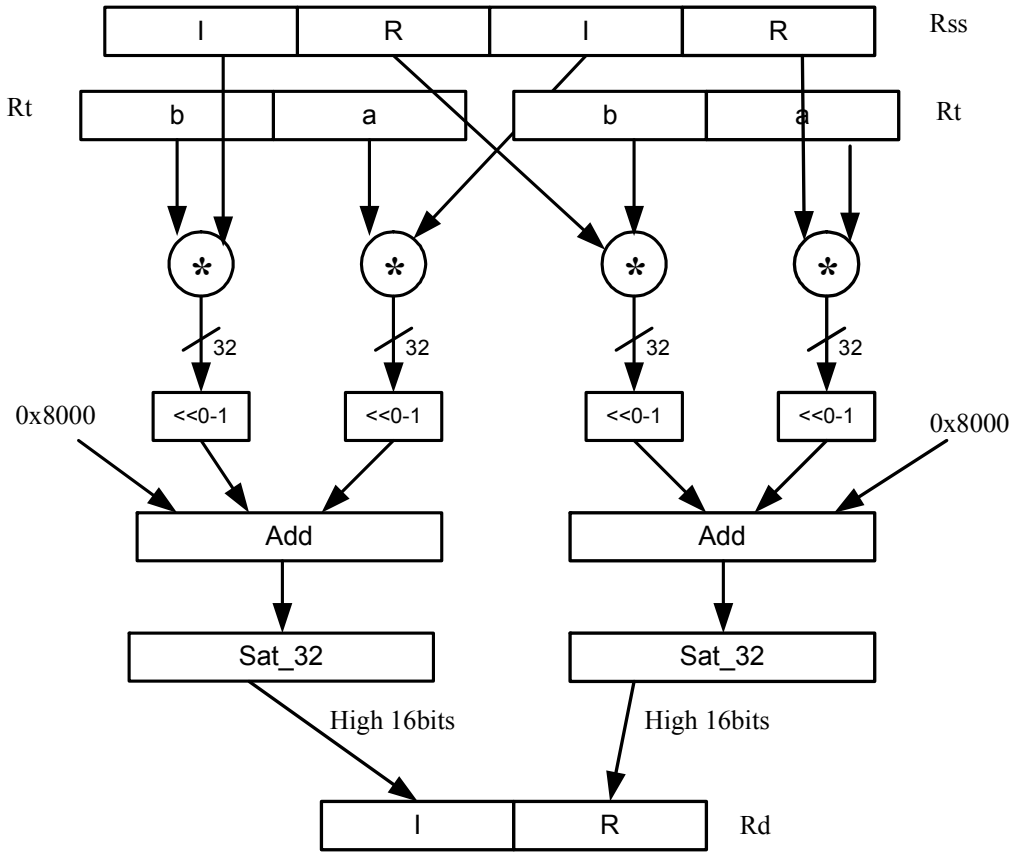
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrcmpys(Rss,Rtt):<<1 :sat:raw:hi
1	1	1	0	1	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrcmpys(Rss,Rtt):<<1 :sat:raw:lo
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrcmpys(Rss,Rtt):<< 1:sat:raw:hi
1	1	1	0	1	0	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrcmpys(Rss,Rtt):<< 1:sat:raw:lo

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector reduce complex multiply by scalar with round and pack

Multiply a complex number by scalar. Rss contains two complex numbers. The real portions are each multiplied by two scalars contained in register Rt, scaled, summed, rounded, and saturated. The upper 16bits of this result are packed in the lower halfword of Rd. A similar operation is done on the two imaginary portions of Rss.

Rd=vrcmpys(Rss,Rt):<<1:rnd:sat



Syntax	Behavior

`Rd=vrcmpys(Rss,Rt):<<1:rnd:sat`	`if ("Rt & 1") {`
	`  Assembler mapped to:`
	`"Rd=vrcmpys(Rss,Rtt):<<1:rnd:sat:raw:hi";`
	`} else {`
	`  Assembler mapped to:`
	`"Rd=vrcmpys(Rss,Rtt):<<1:rnd:sat:raw:lo";`
	`};`

`Rd=vrcmpys(Rss,Rtt):<<1:rnd:sat:raw:hi`	`Rd.h[1]=sat_32((Rss.h[1] * Rtt.w[1].h[0])<<1 +`
	`(Rss.h[3] * Rtt.w[1].h[1])<<1 + 0x8000).h[1];`
	`Rd.h[0]=sat_32((Rss.h[0] * Rtt.w[1].h[0])<<1 +`
	`(Rss.h[2] * Rtt.w[1].h[1])<<1 + 0x8000).h[1];`

`Rd=vrcmpys(Rss,Rtt):<<1:rnd:sat:raw:lo`	`Rd.h[1]=sat_32((Rss.h[1] * Rtt.w[0].h[0])<<1 +`
	`(Rss.h[3] * Rtt.w[0].h[1])<<1 + 0x8000).h[1];`
	`Rd.h[0]=sat_32((Rss.h[0] * Rtt.w[0].h[0])<<1 +`
	`(Rss.h[2] * Rtt.w[0].h[1])<<1 + 0x8000).h[1];`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vrcmpys(Rss,Rt):<<1:rnd:sat`	`Word32 Q6_R_vrcmpys_PR_s1_rnd_sat(Word64 Rss, Word32 Rt)`

Encoding

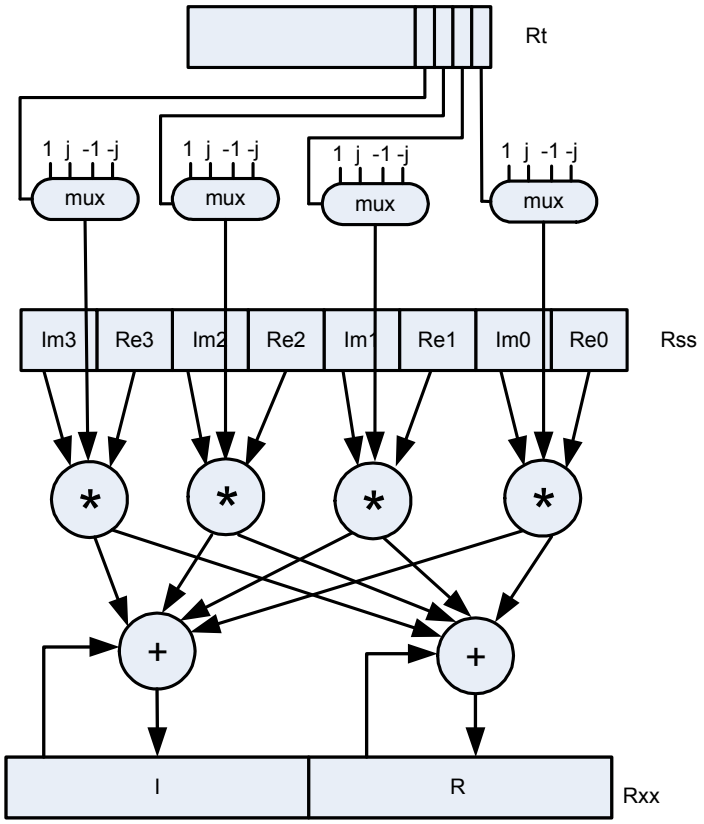
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	1	1	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=vrcmpys(Rss,Rtt):<<1:rnd:sat:raw:hi
1	1	1	0	1	0	0	1	1	-	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=vrcmpys(Rss,Rtt):<<1:rnd:sat:raw:lo

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector reduce complex rotate

This instruction is useful for CDMA despreading. An unsigned 2-bit immediate specifies a byte to use in Rt. Each of four 2-bit fields in the specified byte selects a rotation amount for one of the four complex numbers in Rss. The real and imaginary products are accumulated and stored as a 32-bit complex number in Rd. Optionally, the destination register can also be accumulated.

Rxx += vrcrotate(Rss,Rt,#0)



Syntax	Behavior

`Rdd=vrcrotate(Rss,Rt,#u2)`	`sumr = 0;`
	`sumi = 0;`
	`control = Rt.ub[#u];`
	`for (i = 0; i < 8; i += 2) {`
	`  tmpr = Rss.b[i];`
	`  tmpi = Rss.b[i+1];`
	`  switch (control & 3) {`
	`  case 0: sumr += tmpr;`
	`  sumi += tmpi;`
	`  break;`
	`  case 1: sumr += tmpi;`
	`  sumi -= tmpr;`
	`  break;`
	`  case 2: sumr -= tmpi;`
	`  sumi += tmpr;`
	`  break;`
	`  case 3: sumr -= tmpr;`
	`  sumi -= tmpi;`
	`  break;`
	`  };`
	`  control = control >> 2;`
	`};`
	`Rdd.w[0]=sumr;`
	`Rdd.w[1]=sumi;`

`Rxx+=vrcrotate(Rss,Rt,#u2)`	`sumr = 0;`
	`sumi = 0;`
	`control = Rt.ub[#u];`
	`for (i = 0; i < 8; i += 2) {`
	`  tmpr = Rss.b[i];`
	`  tmpi = Rss.b[i+1];`
	`  switch (control & 3) {`
	`  case 0: sumr += tmpr;`
	`  sumi += tmpi;`
	`  break;`
	`  case 1: sumr += tmpi;`
	`  sumi -= tmpr;`
	`  break;`
	`  case 2: sumr -= tmpi;`
	`  sumi += tmpr;`
	`  break;`
	`  case 3: sumr -= tmpr;`
	`  sumi -= tmpi;`
	`  break;`
	`  };`
	`  control = control >> 2;`
	`};`
	`Rxx.w[0]=Rxx.w[0] + sumr;`
	`Rxx.w[1]=Rxx.w[1] + sumi;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vrcrotate(Rss,Rt,#u2)`	`Word64 Q6_P_vrcrotate_PRI(Word64 Rss, Word32 Rt, Word32 Iu2)`
`Rxx+=vrcrotate(Rss,Rt,#u2)`	`Word64 Q6_P_vrcrotateacc_PRI(Word64 Rxx, Word64 Rss, Word32 Rt, Word32 Iu2)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	1	1	-	s	s	s	s	s	P	P	i	t	t	t	t	t	1	1	i	d	d	d	d	d	Rdd=vrcrotate(Rss,Rt,#u2)
ICLASS				RegType				Maj			s5					Parse			t5								x5					
1	1	0	0	1	0	1	1	1	0	1	s	s	s	s	s	P	P	i	t	t	t	t	t	-	-	i	x	x	x	x	x	Rxx+=vrcrotate(Rss,Rt,#u2)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

11.12.4 XTYPE/FP

(Please Insert Description for 'XTYPE/FP')

Floating point addition

Add two floating point values

Syntax	Behavior
`Rd=sfadd(Rs,Rt)`	`Rd=Rs+Rt;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=sfadd(Rs,Rt)`	`Word32 Q6_R_sfadd_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sfadd(Rs,Rt)

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Classify floating-point value

Classify floating point values. Classes are Normal, Subnormal, Zero, NaN, or Infinity. If the number is one of the specified classes, return true.

Syntax	Behavior

`Pd=dfclass(Rss,#u5)`	`Pd = 0;`
	`class = fpclassify(Rss);`
	`if (#u.0 && (class == FP_ZERO)) Pd = 0xff;`
	`if (#u.1 && (class == FP_NORMAL)) Pd = 0xff;`
	`if (#u.2 && (class == FP_SUBNORMAL)) Pd = 0xff;`
	`if (#u.3 && (class == FP_INFINITE)) Pd = 0xff;`
	`if (#u.4 && (class == FP_NAN)) Pd = 0xff;`
	`cancel_flags();`

`Pd=sfclass(Rs,#u5)`	`Pd = 0;`
	`class = fpclassify(Rs);`
	`if (#u.0 && (class == FP_ZERO)) Pd = 0xff;`
	`if (#u.1 && (class == FP_NORMAL)) Pd = 0xff;`
	`if (#u.2 && (class == FP_SUBNORMAL)) Pd = 0xff;`
	`if (#u.3 && (class == FP_INFINITE)) Pd = 0xff;`
	`if (#u.4 && (class == FP_NAN)) Pd = 0xff;`
	`cancel_flags();`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=dfclass(Rss,#u5)`	`Byte Q6_p_dfclass_PI(Word64 Rss, Word32 Iu5)`
`Pd=sfclass(Rs,#u5)`	`Byte Q6_p_sfclass_RI(Word32 Rs, Word32 Iu5)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse														d2			
1	0	0	0	0	1	0	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	-	-	-	-	-	-	**d**	**d**	Pd=sfclass(Rs,#u5)
ICLASS				RegType							s5					Parse														d2		
1	1	0	1	1	1	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	0	0	**i**	**i**	**i**	**i**	**i**	1	0	-	**d**	**d**	Pd=dfclass(Rss,#u5)

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`RegType`	Register Type

Compare floating-point value

Compare floating point values. p0 returns true if at least one value is a NaN, zero otherwise.

Syntax	Behavior
`Pd=dfcmp.eq(Rss,Rtt)`	`Pd=Rss==Rtt ? 0xff : 0x00;`
`Pd=dfcmp.ge(Rss,Rtt)`	`Pd=Rss>=Rtt ? 0xff : 0x00;`
`Pd=dfcmp.gt(Rss,Rtt)`	`Pd=Rss>Rtt ? 0xff : 0x00;`
`Pd=dfcmp.uo(Rss,Rtt)`	`Pd=isunordered(Rss,Rtt) ? 0xff : 0x00;`
`Pd=sfcmp.eq(Rs,Rt)`	`Pd=Rs==Rt ? 0xff : 0x00;`
`Pd=sfcmp.ge(Rs,Rt)`	`Pd=Rs>=Rt ? 0xff : 0x00;`
`Pd=sfcmp.gt(Rs,Rt)`	`Pd=Rs>Rt ? 0xff : 0x00;`
`Pd=sfcmp.uo(Rs,Rt)`	`Pd=isunordered(Rs,Rt) ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=dfcmp.eq(Rss,Rtt)`	`Byte Q6_p_dfcmp_eq_PP(Word64 Rss, Word64 Rtt)`
`Pd=dfcmp.ge(Rss,Rtt)`	`Byte Q6_p_dfcmp_ge_PP(Word64 Rss, Word64 Rtt)`
`Pd=dfcmp.gt(Rss,Rtt)`	`Byte Q6_p_dfcmp_gt_PP(Word64 Rss, Word64 Rtt)`
`Pd=dfcmp.uo(Rss,Rtt)`	`Byte Q6_p_dfcmp_uo_PP(Word64 Rss, Word64 Rtt)`
`Pd=sfcmp.eq(Rs,Rt)`	`Byte Q6_p_sfcmp_eq_RR(Word32 Rs, Word32 Rt)`
`Pd=sfcmp.ge(Rs,Rt)`	`Byte Q6_p_sfcmp_ge_RR(Word32 Rs, Word32 Rt)`
`Pd=sfcmp.gt(Rs,Rt)`	`Byte Q6_p_sfcmp_gt_RR(Word32 Rs, Word32 Rt)`
`Pd=sfcmp.uo(Rs,Rt)`	`Byte Q6_p_sfcmp_uo_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min						d2		
1	1	0	0	0	1	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	-	-	-	**d**	**d**	Pd=sfcmp.ge(Rs,Rt)
1	1	0	0	0	1	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	-	-	-	**d**	**d**	Pd=sfcmp.uo(Rs,Rt)
1	1	0	0	0	1	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	-	-	-	**d**	**d**	Pd=sfcmp.eq(Rs,Rt)
1	1	0	0	0	1	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	-	-	-	**d**	**d**	Pd=sfcmp.gt(Rs,Rt)
ICLASS				RegType							s5					Parse			t5					MinOp						d2		
1	1	0	1	0	0	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	-	-	-	**d**	**d**	Pd=dfcmp.eq(Rss,Rtt)
1	1	0	1	0	0	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	-	-	-	**d**	**d**	Pd=dfcmp.gt(Rss,Rtt)
1	1	0	1	0	0	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	-	-	-	**d**	**d**	Pd=dfcmp.ge(Rss,Rtt)
1	1	0	1	0	0	1	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	-	-	-	**d**	**d**	Pd=dfcmp.uo(Rss,Rtt)

Field name	Description
RegType	Register Type
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Convert floating-point value to other format

Convert floating point values. If rounding is required, it happens according to the rounding mode.

Syntax	Behavior
`Rd=convert_df2sf(Rss)`	`Rd = conv_df_to_sf(Rss);`
`Rdd=convert_sf2df(Rs)`	`Rdd = conv_sf_to_df(Rs);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=convert_df2sf(Rss)`	`Word32 Q6_R_convert_df2sf_P(Word64 Rss)`
`Rdd=convert_sf2df(Rs)`	`Word64 Q6_P_convert_sf2df_R(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	1	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=convert_sf2df(Rs)
1	0	0	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_df2sf(Rss)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Convert integer to floating-point value

Convert floating point values. If rounding is required, it happens according to the rounding mode unless the :chop option is specified.

Syntax	Behavior
`Rd=convert_d2sf(Rss)`	`Rd = conv_8s_to_sf(Rss.s64);`
`Rd=convert_ud2sf(Rss)`	`Rd = conv_8u_to_sf(Rss.u64);`
`Rd=convert_uw2sf(Rs)`	`Rd = conv_4u_to_sf(Rs.uw[0]);`
`Rd=convert_w2sf(Rs)`	`Rd = conv_4s_to_sf(Rs.s32);`
`Rdd=convert_d2df(Rss)`	`Rdd = conv_8s_to_df(Rss.s64);`
`Rdd=convert_ud2df(Rss)`	`Rdd = conv_8u_to_df(Rss.u64);`
`Rdd=convert_uw2df(Rs)`	`Rdd = conv_4u_to_df(Rs.uw[0]);`
`Rdd=convert_w2df(Rs)`	`Rdd = conv_4s_to_df(Rs.s32);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=convert_d2sf(Rss)`	`Word32 Q6_R_convert_d2sf_P(Word64 Rss)`
`Rd=convert_ud2sf(Rss)`	`Word32 Q6_R_convert_ud2sf_P(Word64 Rss)`
`Rd=convert_uw2sf(Rs)`	`Word32 Q6_R_convert_uw2sf_R(Word32 Rs)`
`Rd=convert_w2sf(Rs)`	`Word32 Q6_R_convert_w2sf_R(Word32 Rs)`
`Rdd=convert_d2df(Rss)`	`Word64 Q6_P_convert_d2df_P(Word64 Rss)`
`Rdd=convert_ud2df(Rss)`	`Word64 Q6_P_convert_ud2df_P(Word64 Rss)`
`Rdd=convert_uw2df(Rs)`	`Word64 Q6_P_convert_uw2df_R(Word32 Rs)`
`Rdd=convert_w2df(Rs)`	`Word64 Q6_P_convert_w2df_R(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=convert_ud2df(Rss)
1	0	0	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=convert_d2df(Rss)
1	0	0	0	0	1	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=convert_uw2df(Rs)
1	0	0	0	0	1	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=convert_w2df(Rs)
1	0	0	0	1	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_ud2sf(Rss)
1	0	0	0	1	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_d2sf(Rss)
1	0	0	0	1	0	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=convert_uw2sf(Rs)
1	0	0	0	1	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=convert_w2sf(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Convert floating-point value to integer

Convert floating point values. If rounding is required, it happens according to the rounding mode unless the :chop option is specified. If the value is out of range of the destination integer type, the INVALID flag is raised and closest integer is chosen, including for infinite inputs. For NaN inputs, the INVALID flag is also raised, and the output value is IMPLEMENTATION DEFINED.

Syntax	Behavior
`Rd=convert_df2uw(Rss)`	`Rd = conv_df_to_4u(Rss).uw[0];`
`Rd=convert_df2uw(Rss):chop`	`round_to_zero();` `Rd = conv_df_to_4u(Rss).uw[0];`
`Rd=convert_df2w(Rss)`	`Rd = conv_df_to_4s(Rss).s32;`
`Rd=convert_df2w(Rss):chop`	`round_to_zero();` `Rd = conv_df_to_4s(Rss).s32;`
`Rd=convert_sf2uw(Rs)`	`Rd = conv_sf_to_4u(Rs).uw[0];`
`Rd=convert_sf2uw(Rs):chop`	`round_to_zero();` `Rd = conv_sf_to_4u(Rs).uw[0];`
`Rd=convert_sf2w(Rs)`	`Rd = conv_sf_to_4s(Rs).s32;`
`Rd=convert_sf2w(Rs):chop`	`round_to_zero();` `Rd = conv_sf_to_4s(Rs).s32;`
`Rdd=convert_df2d(Rss)`	`Rdd = conv_df_to_8s(Rss).s64;`
`Rdd=convert_df2d(Rss):chop`	`round_to_zero();` `Rdd = conv_df_to_8s(Rss).s64;`
`Rdd=convert_df2ud(Rss)`	`Rdd = conv_df_to_8u(Rss).u64;`
`Rdd=convert_df2ud(Rss):chop`	`round_to_zero();` `Rdd = conv_df_to_8u(Rss).u64;`
`Rdd=convert_sf2d(Rs)`	`Rdd = conv_sf_to_8s(Rs).s64;`
`Rdd=convert_sf2d(Rs):chop`	`round_to_zero();` `Rdd = conv_sf_to_8s(Rs).s64;`
`Rdd=convert_sf2ud(Rs)`	`Rdd = conv_sf_to_8u(Rs).u64;`
`Rdd=convert_sf2ud(Rs):chop`	`round_to_zero();` `Rdd = conv_sf_to_8u(Rs).u64;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=convert_df2uw(Rss)`	`Word32 Q6_R_convert_df2uw_P(Word64 Rss)`
`Rd=convert_df2uw(Rss):chop`	`Word32 Q6_R_convert_df2uw_P_chop(Word64 Rss)`
`Rd=convert_df2w(Rss)`	`Word32 Q6_R_convert_df2w_P(Word64 Rss)`
`Rd=convert_df2w(Rss):chop`	`Word32 Q6_R_convert_df2w_P_chop(Word64 Rss)`
`Rd=convert_sf2uw(Rs)`	`Word32 Q6_R_convert_sf2uw_R(Word32 Rs)`
`Rd=convert_sf2uw(Rs):chop`	`Word32 Q6_R_convert_sf2uw_R_chop(Word32 Rs)`

`Rd=convert_sf2w(Rs)`	`Word32 Q6_R_convert_sf2w_R(Word32 Rs)`
`Rd=convert_sf2w(Rs):chop`	`Word32 Q6_R_convert_sf2w_R_chop(Word32 Rs)`
`Rdd=convert_df2d(Rss)`	`Word64 Q6_P_convert_df2d_P(Word64 Rss)`
`Rdd=convert_df2d(Rss):chop`	`Word64 Q6_P_convert_df2d_P_chop(Word64 Rss)`
`Rdd=convert_df2ud(Rss)`	`Word64 Q6_P_convert_df2ud_P(Word64 Rss)`
`Rdd=convert_df2ud(Rss):chop`	`Word64 Q6_P_convert_df2ud_P_chop(Word64 Rss)`
`Rdd=convert_sf2d(Rs)`	`Word64 Q6_P_convert_sf2d_R(Word32 Rs)`
`Rdd=convert_sf2d(Rs):chop`	`Word64 Q6_P_convert_sf2d_R_chop(Word32 Rs)`
`Rdd=convert_sf2ud(Rs)`	`Word64 Q6_P_convert_sf2ud_R(Word32 Rs)`
`Rdd=convert_sf2ud(Rs):chop`	`Word64 Q6_P_convert_sf2ud_R_chop(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=convert_df2d(Rss)
1	0	0	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=convert_df2ud(Rss)
1	0	0	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=convert_df2d(Rss):chop
1	0	0	0	0	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=convert_df2ud(Rss):chop
1	0	0	0	0	1	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=convert_sf2ud(Rs)
1	0	0	0	0	1	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=convert_sf2d(Rs)
1	0	0	0	0	1	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=convert_sf2ud(Rs):chop
1	0	0	0	0	1	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=convert_sf2d(Rs):chop
1	0	0	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_df2uw(Rss)
1	0	0	0	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_df2w(Rss)
1	0	0	0	1	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_df2uw(Rss):chop
1	0	0	0	1	0	0	0	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_df2w(Rss):chop
1	0	0	0	1	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=convert_sf2uw(Rs)
1	0	0	0	1	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_sf2uw(Rs):chop
1	0	0	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=convert_sf2w(Rs)
1	0	0	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=convert_sf2w(Rs):chop

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Floating point extreme value assistance

For divide and square root routines, certain values are problematic for the default routine. These instructions appropriately fix up the numerator (fixupn), denominator (fixupd), or radicand (fixupr) for proper calculations when combined with the divide or square root approximation instructions.

Syntax	Behavior
`Rd=sffixupd(Rs,Rt)`	`(Rs,Rt,Rd,adjust)=recip_common(Rs,Rt);` `Rd = Rt;`
`Rd=sffixupn(Rs,Rt)`	`(Rs,Rt,Rd,adjust)=recip_common(Rs,Rt);` `Rd = Rs;`
`Rd=sffixupr(Rs)`	`(Rs,Rd,adjust)=invsqrt_common(Rs);` `Rd = Rs;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=sffixupd(Rs,Rt)`	`Word32 Q6_R_sffixupd_RR(Word32 Rs, Word32 Rt)`
`Rd=sffixupn(Rs,Rt)`	`Word32 Q6_R_sffixupn_RR(Word32 Rs, Word32 Rt)`
`Rd=sffixupr(Rs)`	`Word32 Q6_R_sffixupr_R(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sffixupr(Rs)
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			d5					
1	1	1	0	1	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sffixupn(Rs,Rt)
1	1	1	0	1	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=sffixupd(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Floating point fused multiply-add

Multiply two values, and add to (or subtract from) the accumulator. Full intermediate precision is kept.

Syntax	Behavior
`Rx+=sfmpy(Rs,Rt)`	`Rx=fmaf(Rs,Rt,Rx);`
`Rx-=sfmpy(Rs,Rt)`	`Rx=fmaf(-Rs,Rt,Rx);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx+=sfmpy(Rs,Rt)`	`Word32 Q6_R_sfmpyacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=sfmpy(Rs,Rt)`	`Word32 Q6_R_sfmpynac_RR(Word32 Rx, Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=sfmpy(Rs,Rt)
1	1	1	0	1	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rx-=sfmpy(Rs,Rt)

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Floating point fused multiply-add with scaling

Multiply two values, and add to (or subtract from) the accumulator. Full intermediate precision is kept. Additionally, scale the output. This instruction has special handling of corner cases. If a multiplicand source is zero, the accumulator is left unchanged; this is important as -0.0 + 0.0*x yields -0.0. The scaling factor is the predicate taken as a two's compliment number for single precision. The scaling factor is twice the predicate taken as a two's compliment number for double precision.

Syntax	Behavior

`Rx+=sfmpy(Rs,Rt,Pu):scale`	`if (isnan(Rx) || isnan(Rs) || isnan(Rt)) Rx = NaN;`
	`;`
	`tmp=fmaf(Rs,Rt,Rx) * 2**(Pu);`
	`if ((tmp != Rx) || ((Rs != 0.0) && (Rt != 0.0)))`
	`Rx = tmp;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx+=sfmpy(Rs,Rt,Pu):scale`	`Word32 Q6_R_sfmpyacc_RRp_scale(Word32 Rx, Word32 Rs, Word32 Rt, Byte Pu)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						u2		x5						
1	1	1	0	1	1	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**u**	**u**	**x**	**x**	**x**	**x**	**x**	Rx+=sfmpy(Rs,Rt,Pu):scale

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u
x5	Field to encode register x

Floating point reciprocal square root approximation

Provides an approximation of the reciprocal square root of the radicand (Rs), if combined with the appropriate fixup instruction. Certain values (such as infinities or zeros) in the numerator or denominator may yield values that are not reciprocal approximations, but yield the correct answer when combined with fixup instructions and the appropriate routines.

For compatibility, exact results of these instructions can not be relied on. The precision of the approximation for this architecture and later is at least 6.6 bits.

Syntax	Behavior

`Rd,Pe=sfinvsqrta(Rs)`	`if ((Rs,Rd,adjust)=invsqrt_common(Rs)) {`
	`  Pe = adjust;`
	`  idx = (Rs >> 17) & 0x7f;`
	`  mant = (invsqrt_lut[idx] << 15);`
	`  exp = 127 - ((exponent(Rs) - 127) >> 1) - 1;`
	`  Rd = -1**Rs.31 * 1.MANT * 2**(exp-BIAS);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- This instruction provides a certain amount of accuracy. In future versions the accuracy may increase. For future compatibility, dependence on exact values must be avoided.
- The predicate generated by this instruction can not be used as a .new predicate, nor can it be automatically ANDed with another predicate.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									e2		d5						
1	0	0	0	1	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	**e**	**e**	**d**	**d**	**d**	**d**	**d**	Rd,Pe=sfinvsqrta(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
e2	Field to encode register e
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Floating point fused multiply-add for library routines

Multiply two values, and add to (or subtract from) the accumulator. Full intermediate precision is kept. This instruction has special handling of corner cases. Addition of infinities with opposite signs, or subtraction of infinities with like signs, is defined as (positive) zero. Rounding is always Nearest-Even, except that overflows to infinity round to maximal finite values. If a multiplicand source is zero, the accumulator is left unchanged; this is important as -0.0 + 0.0*x yields -0.0. Flags and Exceptions are not generated.

Syntax	Behavior

`Rx+=sfmpy(Rs,Rt):lib`	`round_to_nearest();`
	`infminusinf = ((isinf(Rx)) && (isinf(Rs*Rt)) && (Rs ^ Rx ^ Rt.31 != 0));`
	`infinp = (isinf(Rx)) || (isinf(Rt)) || (isinf(Rs));`
	`if (isnan(Rx) || isnan(Rs) || isnan(Rt)) Rx = NaN;`
	`;`
	`if ((Rs != 0.0) && (Rt != 0.0)) {`
	`  Rx=fmaf(Rs,Rt,Rx);`
	`} else {`
	`  if (isinf(Rs) || isinf(Rt)) Rx = NaN;`
	`};`
	`cancel_flags();`
	`if (isinf(Rx) && !infinp) Rx = Rx - 1;`
	`if (infminusinf) Rx = 0;`

`Rx-=sfmpy(Rs,Rt):lib`	`round_to_nearest();`
	`infinp = (isinf(Rx)) || (isinf(Rt)) || (isinf(Rs));`
	`infminusinf = ((isinf(Rx)) && (isinf(Rs*Rt)) && (Rs ^ Rx ^ Rt.31 == 0));`
	`if (isnan(Rx) || isnan(Rs) || isnan(Rt)) Rx = NaN;`
	`;`
	`if ((Rs != 0.0) && (Rt != 0.0)) {`
	`  Rx=fmaf(-Rs,Rt,Rx);`
	`} else {`
	`  if (isinf(Rs) || isinf(Rt)) Rx = NaN;`
	`};`
	`cancel_flags();`
	`if (isinf(Rx) && !infinp) Rx = Rx - 1;`
	`if (infminusinf) Rx = 0;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rx+=sfmpy(Rs,Rt):lib`	`Word32 Q6_R_sfmpyacc_RR_lib(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=sfmpy(Rs,Rt):lib`	`Word32 Q6_R_sfmpynac_RR_lib(Word32 Rx, Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rx+=sfmpy(Rs,Rt):lib
1	1	1	0	1	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rx-=sfmpy(Rs,Rt):lib

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Create floating-point constant

Using ten bits of immediate, form a floating-point constant.

Syntax	Behavior

`Rd=sfmake(#u10):neg`	`Rd = (127 - 6) << 23;`
	`Rd += (#u << 17);`
	`Rd |= (1 << 31);`

`Rd=sfmake(#u10):pos`	`Rd = (127 - 6) << 23;` `Rd += #u << 17;`

`Rdd=dfmake(#u10):neg`	`Rdd = (1023ULL - 6) << 52;`
	`Rdd += (#u) << 46;`
	`Rdd |= ((1ULL) << 63);`

`Rdd=dfmake(#u10):pos`	`Rdd = (1023ULL - 6) << 52;` `Rdd += (#u) << 46;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=sfmake(#u10):neg`	`Word32 Q6_R_sfmake_I_neg(Word32 Iu10)`
`Rd=sfmake(#u10):pos`	`Word32 Q6_R_sfmake_I_pos(Word32 Iu10)`
`Rdd=dfmake(#u10):neg`	`Word64 Q6_P_dfmake_I_neg(Word32 Iu10)`
`Rdd=dfmake(#u10):pos`	`Word64 Q6_P_dfmake_I_pos(Word32 Iu10)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType												Parse								MinOp			d5					
1	1	0	1	0	1	1	0	0	0	**i**	-	-	-	-	-	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=sfmake(#u10):pos
1	1	0	1	0	1	1	0	0	1	**i**	-	-	-	-	-	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=sfmake(#u10):neg
ICLASS				RegType												Parse											d5					
1	1	0	1	1	0	0	1	0	0	**i**	-	-	-	-	-	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=dfmake(#u10):pos
1	1	0	1	1	0	0	1	0	1	**i**	-	-	-	-	-	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rdd=dfmake(#u10):neg

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d

Floating point maximum

Maximum of two floating point values. If one value is a NaN, the other is chosen.

Syntax	Behavior
`Rd=sfmax(Rs,Rt)`	`Rd = fmaxf(Rs,Rt);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=sfmax(Rs,Rt)`	`Word32 Q6_R_sfmax_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sfmax(Rs,Rt)

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Floating point minimum

Minimum of two floating point values. If one value is a NaN, the other is chosen.

Syntax	Behavior
`Rd=sfmin(Rs,Rt)`	`Rd = fmin(Rs,Rt);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=sfmin(Rs,Rt)`	`Word32 Q6_R_sfmin_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=sfmin(Rs,Rt)

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Floating point multiply

Add two floating point values

Syntax	Behavior
`Rd=sfmpy(Rs,Rt)`	`Rd=Rs*Rt;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=sfmpy(Rs,Rt)`	`Word32 Q6_R_sfmpy_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sfmpy(Rs,Rt)

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Floating point reciprocal approximation

Provides an approximation of the reciprocal of the denominator (Rt), if combined with the appropriate fixup instructions. Certain values (such as infinities or zeros) in the numerator or denominator may yield values that are not reciprocal approximations, but yield the correct answer when combined with fixup instructions and the appropriate routines.

For compatibility, exact results of these instructions can not be relied on. The precision of the approximation for this architecture and later is at least 6.6 bits.

Syntax	Behavior

`Rd,Pe=sfrecipa(Rs,Rt)`	`if ((Rs,Rt,Rd,adjust)=recip_common(Rs,Rt)) {`
	`  Pe = adjust;`
	`  idx = (Rt >> 16) & 0x7f;`
	`  mant = (recip_lut[idx] << 15) | 1;`
	`  exp = 127 - (exponent(Rt) - 127) - 1;`
	`  Rd = -1**Rt.31 * 1.MANT * 2**(exp-BIAS);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- This instruction provides a certain amount of accuracy. In future versions the accuracy may increase. For future compatibility, dependence on exact values must be avoided.
- The predicate generated by this instruction can not be used as a .new predicate, nor can it be automatically ANDed with another predicate.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5							e2		d5					
1	1	1	0	1	0	1	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	**e**	**e**	**d**	**d**	**d**	**d**	**d**	Rd,Pe=sfrecipa(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
e2	Field to encode register e
s5	Field to encode register s
t5	Field to encode register t

Floating point subtraction

Subtract two floating point values

Syntax	Behavior
`Rd=sfsub(Rs,Rt)`	`Rd=Rs-Rt;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=sfsub(Rs,Rt)`	`Word32 Q6_R_sfsub_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=sfsub(Rs,Rt)

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

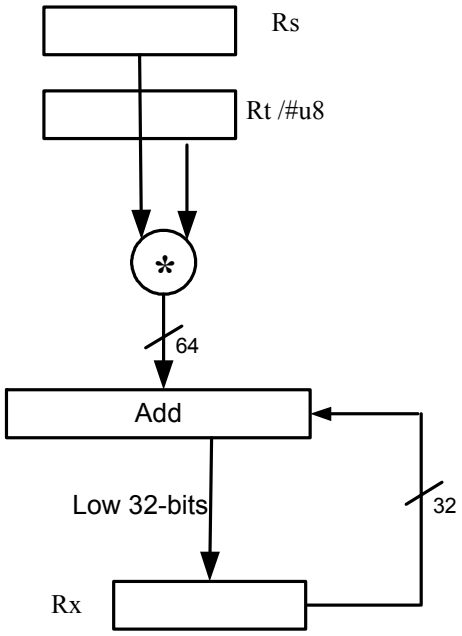
11.12.5 XTYPE/MPY

The XTYPE/MPY instruction subclass includes instructions which perform multiplication.

Multiply and use lower result

Multiply the signed 32-bit integer in Rs by either the signed 32-bit integer in Rt or an unsigned immediate value. The 64-bit result is optionally accumulated with the 32-bit destination, or added to an immediate. The least-significant 32-bits of the result are written to the single destination register.

This multiply produces the correct results for the ANSI C multiplication of two signed or unsigned integers with an integer result.



Syntax	Behavior
`Rd=+mpyi(Rs,#u8)`	`apply_extension(#u);` `Rd=Rs*#u;`
`Rd=-mpyi(Rs,#u8)`	`Rd=Rs*-#u;`
`Rd=add(#u6,mpyi(Rs,#U6))`	`apply_extension(#u);` `Rd = #u + Rs*#U;`
`Rd=add(#u6,mpyi(Rs,Rt))`	`apply_extension(#u);` `Rd = #u + Rs*Rt;`
`Rd=add(Ru,mpyi(#u6:2,Rs))`	`Rd = Ru + Rs*#u;`
`Rd=add(Ru,mpyi(Rs,#u6))`	`apply_extension(#u);` `Rd = Ru + Rs*#u;`

`Rd=mpyi(Rs,#m9)`	`if ("((#m9<0) && (#m9>-256))") {`
	`  Assembler mapped to: "Rd=-mpyi(Rs,#m9*(-1))";`
	`} else {`
	`  Assembler mapped to: "Rd=+mpyi(Rs,#m9)";`
	`};`

`Rd=mpyi(Rs,Rt)`	`Rd=Rs*Rt;`

Syntax	Behavior
Rd=mpyui(Rs,Rt)	Assembler mapped to: "Rd=mpyi(Rs,Rt)"
Rx+=mpyi(Rs,#u8)	apply_extension(#u); Rx=Rx + (Rs*#u);
Rx+=mpyi(Rs,Rt)	Rx=Rx + Rs*Rt;
Rx-=mpyi(Rs,#u8)	apply_extension(#u); Rx=Rx - (Rs*#u);
Ry=add(Ru,mpyi(Ry,Rs))	Ry = Ru + Rs*Ry;

Class: XTYPE (slots 2,3)

Intrinsics

Rd=add(#u6,mpyi(Rs,#U6))	Word32 Q6_R_add_mpyi_IRI(Word32 Iu6, Word32 Rs, Word32 IU6)
Rd=add(#u6,mpyi(Rs,Rt))	Word32 Q6_R_add_mpyi_IRR(Word32 Iu6, Word32 Rs, Word32 Rt)
Rd=add(Ru,mpyi(#u6:2,Rs))	Word32 Q6_R_add_mpyi_RIR(Word32 Ru, Word32 Iu6_2, Word32 Rs)
Rd=add(Ru,mpyi(Rs,#u6))	Word32 Q6_R_add_mpyi_RRI(Word32 Ru, Word32 Rs, Word32 Iu6)
Rd=mpyi(Rs,#m9)	Word32 Q6_R_mpyi_RI(Word32 Rs, Word32 Im9)
Rd=mpyi(Rs,Rt)	Word32 Q6_R_mpyi_RR(Word32 Rs, Word32 Rt)
Rd=mpyui(Rs,Rt)	Word32 Q6_R_mpyui_RR(Word32 Rs, Word32 Rt)
Rx+=mpyi(Rs,#u8)	Word32 Q6_R_mpyiacc_RI(Word32 Rx, Word32 Rs, Word32 Iu8)
Rx+=mpyi(Rs,Rt)	Word32 Q6_R_mpyiacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyi(Rs,#u8)	Word32 Q6_R_mpyinac_RI(Word32 Rx, Word32 Rs, Word32 Iu8)
Ry=add(Ru,mpyi(Ry,Rs))	Word32 Q6_R_add_mpyi_RRR(Word32 Ru, Word32 Ry, Word32 Rs)

Encoding

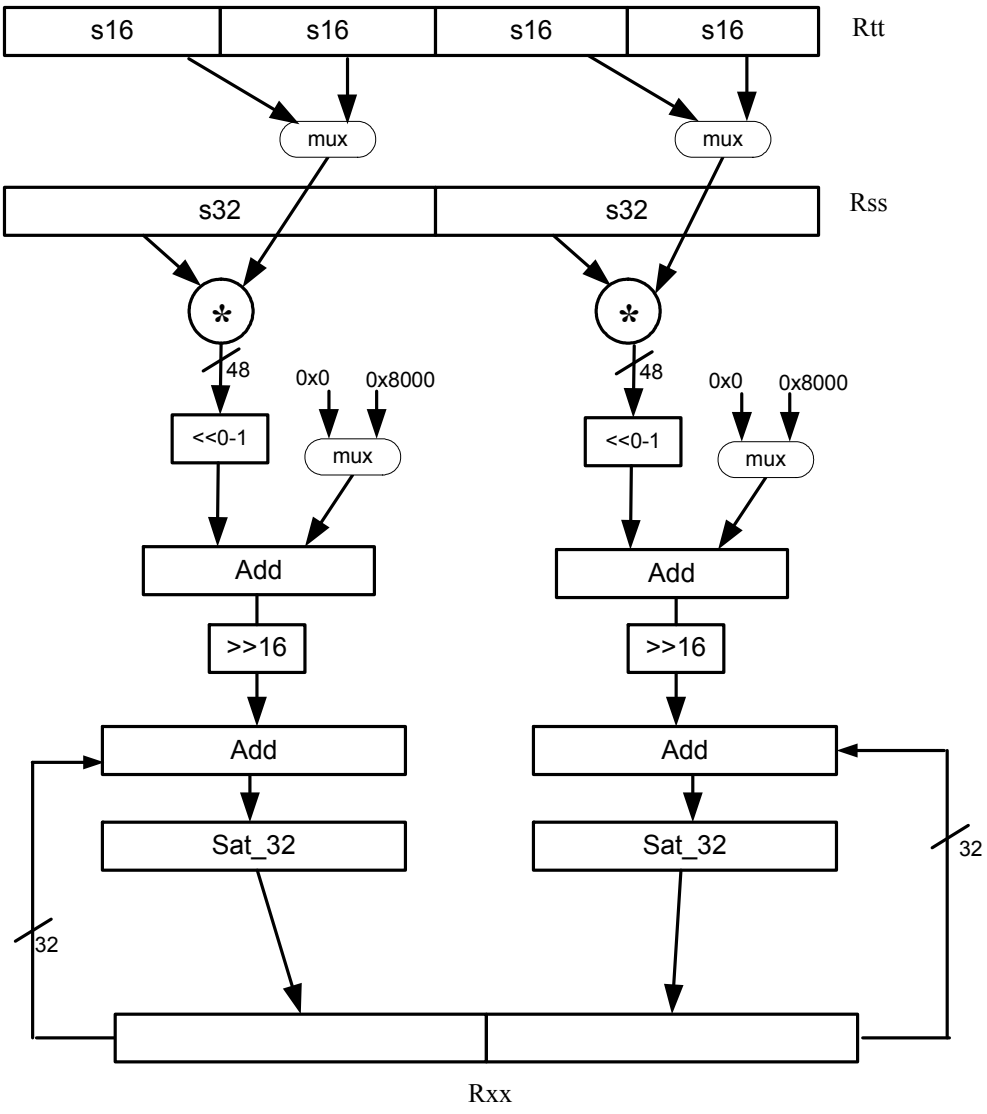
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp			d5					
1	1	0	1	0	1	1	1	0	i	i	s	s	s	s	s	P	P	i	t	t	t	t	t	i	i	i	d	d	d	d	d	Rd=add(#u6,mpyi(Rs,Rt))
ICLASS				RegType							s5					Parse			d5													
1	1	0	1	1	0	0	0	I	i	i	s	s	s	s	s	P	P	i	d	d	d	d	d	i	i	i	I	I	I	I	I	Rd=add(#u6,mpyi(Rs,#U6))
ICLASS				RegType							s5					Parse			d5								u5					
1	1	0	1	1	1	1	1	0	i	i	s	s	s	s	s	P	P	i	d	d	d	d	d	i	i	i	u	u	u	u	u	Rd=add(Ru,mpyi(#u6:2,Rs))
1	1	0	1	1	1	1	1	1	i	i	s	s	s	s	s	P	P	i	d	d	d	d	d	i	i	i	u	u	u	u	u	Rd=add(Ru,mpyi(Rs,#u6))
ICLASS				RegType				MajOp			s5					Parse			y5								u5					
1	1	1	0	0	0	1	1	0	0	0	s	s	s	s	s	P	P	-	y	y	y	y	y	-	-	-	u	u	u	u	u	Ry=add(Ru,mpyi(Ry,Rs))
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	0	0	0	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=+mpyi(Rs,#u8)
1	1	1	0	0	0	0	0	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=-mpyi(Rs,#u8)
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	1	1	0	0	0	0	1	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	Rx+=mpyi(Rs,#u8)
1	1	1	0	0	0	0	1	1	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**x**	**x**	**x**	**x**	**x**	Rx-=mpyi(Rs,#u8)
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			d5					
1	1	1	0	1	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpyi(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5						MinOp			x5					
1	1	1	0	1	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpyi(Rs,Rt)

Field name	Description
RegType	Register Type
MajOp	Major Opcode
MinOp	Minor Opcode
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u5	Field to encode register u
x5	Field to encode register x
y5	Field to encode register y

Vector multiply word by signed half (32x16)

Perform mixed precision vector multiply operations. A 32-bit word from vector Rss is multiplied by a 16-bit halfword (either even or odd) from vector Rtt. The multiplication is performed as a signed 32x16, which produces a 48-bit result. This result is optionally scaled left by one bit. This result is then shifted right by 16 bits, optionally accumulated and then saturated to 32-bits. This operation is available in vector form (vmpyweh/vmpywoh) and non-vector form (mpyweh/mpywoh).


s16
s16
s16
s16
Rtt
mux
mux
s32
s32
Rss
*
*
48
48
0x0
0x8000
0x0
0x8000
<<0-1
<<0-1
mux
mux
Add
Add
>>16
>>16
Add
Add
Sat_32
Sat_32
32
32
Rxx

Syntax	Behavior
`Rdd=vmpyweh(Rss,Rtt)[:<<1]:rnd:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.h[2])[<<1]+0x8000)>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.h[0])[<<1]+0x8000)>>16);`
`Rdd=vmpyweh(Rss,Rtt)[:<<1]:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.h[2])[<<1])>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.h[0])[<<1])>>16);`
`Rdd=vmpywoh(Rss,Rtt)[:<<1]:rnd:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.h[3])[<<1]+0x8000)>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.h[1])[<<1]+0x8000)>>16);`
`Rdd=vmpywoh(Rss,Rtt)[:<<1]:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.h[3])[<<1])>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.h[1])[<<1])>>16);`
`Rxx+=vmpyweh(Rss,Rtt)[:<<1]:rnd:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.h[2])[<<1]+0x8000)>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.h[0])[<<1]+0x8000)>>16));`
`Rxx+=vmpyweh(Rss,Rtt)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.h[2])[<<1])>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.h[0])[<<1])>>16));`
`Rxx+=vmpywoh(Rss,Rtt)[:<<1]:rnd:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.h[3])[<<1]+0x8000)>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.h[1])[<<1]+0x8000)>>16 ));`
`Rxx+=vmpywoh(Rss,Rtt)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.h[3])[<<1])>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.h[1])[<<1])>>16 ));`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vmpyweh(Rss,Rtt):<<1:rnd:sat`	`Word64 Q6_P_vmpyweh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpyweh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpyweh_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpyweh(Rss,Rtt):rnd:sat`	`Word64 Q6_P_vmpyweh_PP_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpyweh(Rss,Rtt):sat`	`Word64 Q6_P_vmpyweh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpywoh(Rss,Rtt):<<1:rnd:sat`	`Word64 Q6_P_vmpywoh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpywoh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpywoh_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpywoh(Rss,Rtt):rnd:sat`	`Word64 Q6_P_vmpywoh_PP_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpywoh(Rss,Rtt):sat`	`Word64 Q6_P_vmpywoh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyweh(Rss,Rtt):<<1:rnd:sat`	`Word64 Q6_P_vmpywehacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyweh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpywehacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyweh(Rss,Rtt):rnd:sat`	`Word64 Q6_P_vmpywehacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyweh(Rss,Rtt):sat`	`Word64 Q6_P_vmpywehacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpywoh(Rss,Rtt):<<1:rnd:sat`	`Word64 Q6_P_vmpywohacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpywoh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpywohacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpywoh(Rss,Rtt):rnd:sat`	`Word64 Q6_P_vmpywohacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpywoh(Rss,Rtt):sat`	`Word64 Q6_P_vmpywohacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

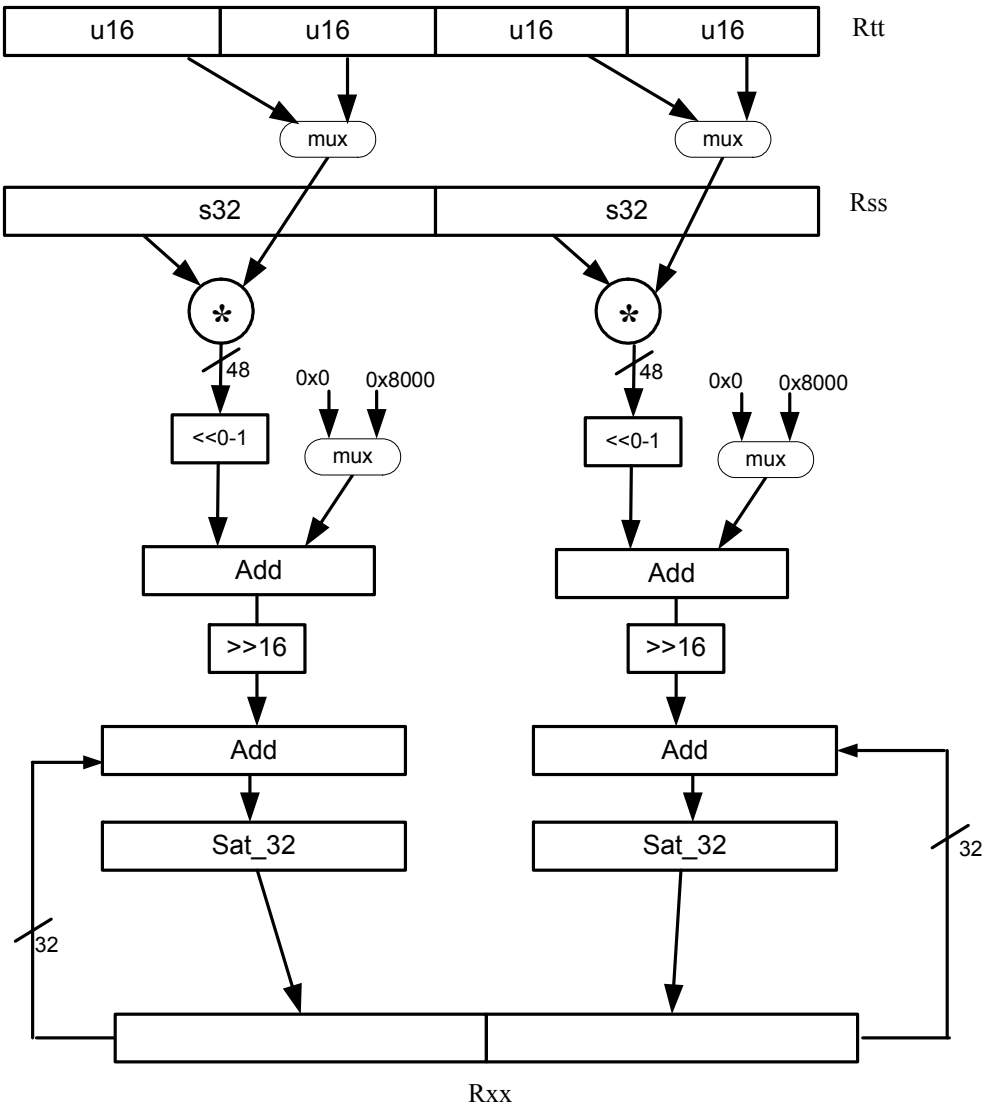
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyweh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpywoh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyweh(Rss,Rtt)[:<<N]:rnd:sat
1	1	1	0	1	0	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpywoh(Rss,Rtt)[:<<N]:rnd:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyweh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpywoh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyweh(Rss,Rtt)[:<<N]:rnd:sat
1	1	1	0	1	0	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpywoh(Rss,Rtt)[:<<N]:rnd:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector multiply word by unsigned half (32x16)

Perform mixed precision vector multiply operations. A 32-bit signed word from vector Rss is multiplied by a 16-bit unsigned halfword (either odd or even) from vector Rtt. This multiplication produces a 48-bit result. This result is optionally scaled left by one bit, and then a rounding constant is optionally added to the lower 16-bits. This result is then shifted right by 16 bits, optionally accumulated and then saturated to 32-bits. This is a dual vector operation and is performed for both high and low word of Rss.


u16
u16
u16
u16
Rtt
mux
mux
s32
s32
Rss
*
*
48
48
0x0
0x8000
0x0
0x8000
<<0-1
<<0-1
mux
mux
Add
Add
>>16
>>16
Add
Add
Sat_32
Sat_32
32
32
Rxx

Syntax	Behavior
`Rdd=vmpyweuh(Rss,Rtt)[:<<1]:rnd:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.uh[2])[<<1]+0x8000)>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.uh[0])[<<1]+0x8000)>>16);`
`Rdd=vmpyweuh(Rss,Rtt)[:<<1]:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.uh[2])[<<1])>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.uh[0])[<<1])>>16);`
`Rdd=vmpywouh(Rss,Rtt)[:<<1]:rnd:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.uh[3])[<<1]+0x8000)>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.uh[1])[<<1]+0x8000)>>16);`
`Rdd=vmpywouh(Rss,Rtt)[:<<1]:sat`	`Rdd.w[1]=sat_32(((Rss.w[1] * Rtt.uh[3])[<<1])>>16); Rdd.w[0]=sat_32(((Rss.w[0] * Rtt.uh[1])[<<1])>>16);`
`Rxx+=vmpyweuh(Rss,Rtt)[:<<1]:rnd:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.uh[2])[<<1]+0x8000)>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.uh[0])[<<1]+0x8000)>>16));`
`Rxx+=vmpyweuh(Rss,Rtt)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.uh[2])[<<1])>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.uh[0])[<<1])>>16));`
`Rxx+=vmpywouh(Rss,Rtt)[:<<1]:rnd:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.uh[3])[<<1]+0x8000)>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.uh[1])[<<1]+0x8000)>>16 ));`
`Rxx+=vmpywouh(Rss,Rtt)[:<<1]:sat`	`Rxx.w[1]=sat_32(Rxx.w[1] + (((Rss.w[1] * Rtt.uh[3])[<<1])>>16)); Rxx.w[0]=sat_32(Rxx.w[0] + (((Rss.w[0] * Rtt.uh[1])[<<1])>>16 ));`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vmpyweuh(Rss,Rtt):<<1:rnd:sat`	`Word64 Q6_P_vmpyweuh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpyweuh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpyweuh_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpyweuh(Rss,Rtt):rnd:sat`	`Word64 Q6_P_vmpyweuh_PP_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpyweuh(Rss,Rtt):sat`	`Word64 Q6_P_vmpyweuh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpywouh(Rss,Rtt):<<1:rnd:sat`	`Word64 Q6_P_vmpywouh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpywouh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpywouh_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpywouh(Rss,Rtt):rnd:sat`	`Word64 Q6_P_vmpywouh_PP_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpywouh(Rss,Rtt):sat`	`Word64 Q6_P_vmpywouh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyweuh(Rss,Rtt):<<1:rnd:sat`	`Word64 Q6_P_vmpyweuhacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyweuh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpyweuhacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyweuh(Rss,Rtt):rnd:sat`	`Word64 Q6_P_vmpyweuhacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyweuh(Rss,Rtt):sat`	`Word64 Q6_P_vmpyweuhacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpywouh(Rss,Rtt):<<1:rnd:sat`	`Word64 Q6_P_vmpywouhacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpywouh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpywouhacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpywouh(Rss,Rtt):rnd:sat`	`Word64 Q6_P_vmpywouhacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpywouh(Rss,Rtt):sat`	`Word64 Q6_P_vmpywouhacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

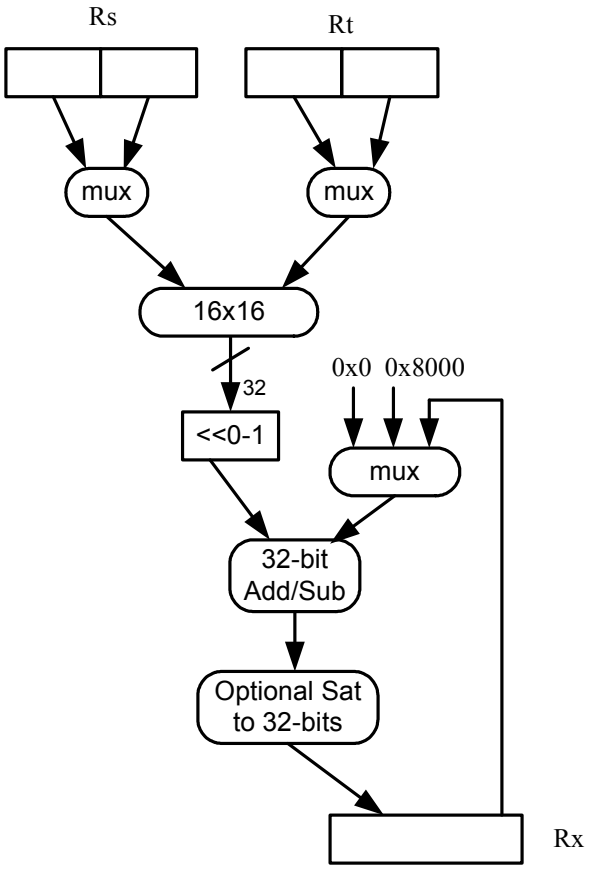
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyweuh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpywouh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	0	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyweuh(Rss,Rtt)[:<<N]:rnd:sat
1	1	1	0	1	0	0	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpywouh(Rss,Rtt)[:<<N]:rnd:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyweuh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpywouh(Rss,Rtt)[:<<N]:sat
1	1	1	0	1	0	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyweuh(Rss,Rtt)[:<<N]:rnd:sat
1	1	1	0	1	0	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpywouh(Rss,Rtt)[:<<N]:rnd:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

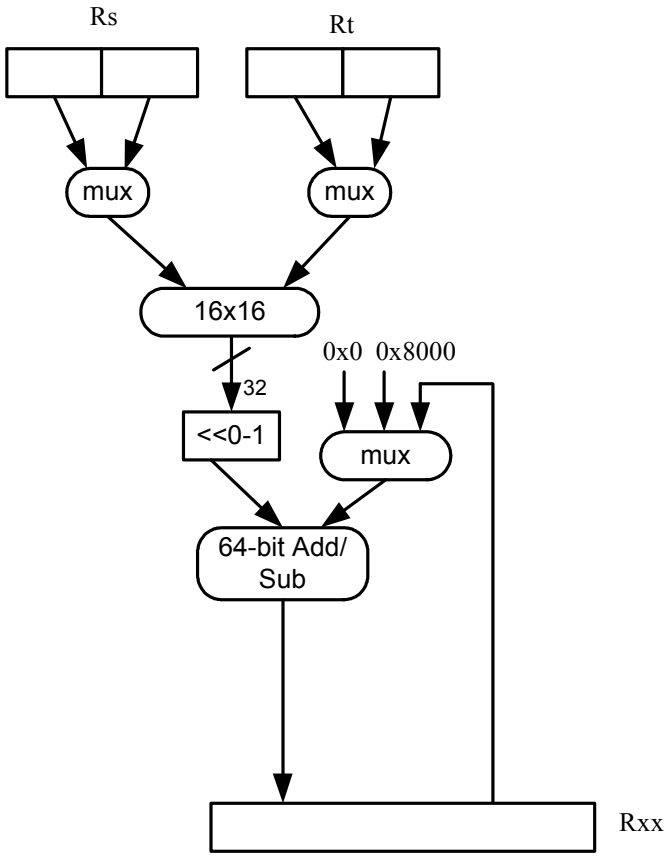
Multiply signed halfwords

Multiply two signed halfwords. Optionally shift the multiplier result by 1 bit. This result can be accumulated or rounded. The destination/accumulator can be either 32 or 64-bits. For 32-bit results, saturation is optional.

Rx+=mpy(Rs.[HL],Rt.[HL])[:<<1][:sat]
Rd = mpy(Rs.[HL],Rt.[HL])[:<<1][:rnd][:sat]



Rxx+=mpy(Rs.[HL],Rt.[HL])[:<<1]
Rdd = mpy(Rs.[HL],Rt.[HL])[:<<1][:rnd]



Syntax	Behavior
`Rd=mpy(Rs.[HL],Rt.[HL])[:<<1][:rnd][:sat]`	`Rd=[sat_32]([round]((Rs.h[01] * Rt.h[01])[<<1]));`
`Rdd=mpy(Rs.[HL],Rt.[HL])[:<<1][:rnd]`	`Rdd=[round]((Rs.h[01] * Rt.h[01])[<<1]);`
`Rx+=mpy(Rs.[HL],Rt.[HL])[:<<1][:sat]`	`Rx=[sat_32](Rx+ (Rs.h[01] * Rt.h[01])[<<1]);`
`Rx-=mpy(Rs.[HL],Rt.[HL])[:<<1][:sat]`	`Rx=[sat_32](Rx- (Rs.h[01] * Rt.h[01])[<<1]);`
`Rxx+=mpy(Rs.[HL],Rt.[HL])[:<<1]`	`Rxx=Rxx+ (Rs.h[01] * Rt.h[01])[<<1];`
`Rxx-=mpy(Rs.[HL],Rt.[HL])[:<<1]`	`Rxx=Rxx- (Rs.h[01] * Rt.h[01])[<<1];`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

Rd=mpy(Rs.H,Rt.H)	Word32 Q6_R_mpy_RhRh(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.H):<<1	Word32 Q6_R_mpy_RhRh_s1(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.H):<<1:rnd	Word32 Q6_R_mpy_RhRh_s1_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.H):<<1:rnd:sat	Word32 Q6_R_mpy_RhRh_s1_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.H):<<1:sat	Word32 Q6_R_mpy_RhRh_s1_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.H):rnd	Word32 Q6_R_mpy_RhRh_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.H):rnd:sat	Word32 Q6_R_mpy_RhRh_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.H):sat	Word32 Q6_R_mpy_RhRh_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L)	Word32 Q6_R_mpy_RhRl(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):<<1	Word32 Q6_R_mpy_RhRl_s1(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):<<1:rnd	Word32 Q6_R_mpy_RhRl_s1_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):<<1:rnd:sat	Word32 Q6_R_mpy_RhRl_s1_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):<<1:sat	Word32 Q6_R_mpy_RhRl_s1_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):rnd	Word32 Q6_R_mpy_RhRl_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):rnd:sat	Word32 Q6_R_mpy_RhRl_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.H,Rt.L):sat	Word32 Q6_R_mpy_RhRl_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H)	Word32 Q6_R_mpy_RlRh(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):<<1	Word32 Q6_R_mpy_RlRh_s1(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):<<1:rnd	Word32 Q6_R_mpy_RlRh_s1_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):<<1:rnd:sat	Word32 Q6_R_mpy_RlRh_s1_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):<<1:sat	Word32 Q6_R_mpy_RlRh_s1_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):rnd	Word32 Q6_R_mpy_RlRh_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):rnd:sat	Word32 Q6_R_mpy_RlRh_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.H):sat	Word32 Q6_R_mpy_RlRh_sat(Word32 Rs, Word32 Rt)

Rd=mpy(Rs.L,Rt.L)	Word32 Q6_R_mpy_RlRl(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):<<1	Word32 Q6_R_mpy_RlRl_s1(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):<<1:rnd	Word32 Q6_R_mpy_RlRl_s1_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):<<1:rnd:sat	Word32 Q6_R_mpy_RlRl_s1_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):<<1:sat	Word32 Q6_R_mpy_RlRl_s1_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):rnd	Word32 Q6_R_mpy_RlRl_rnd(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):rnd:sat	Word32 Q6_R_mpy_RlRl_rnd_sat(Word32 Rs, Word32 Rt)
Rd=mpy(Rs.L,Rt.L):sat	Word32 Q6_R_mpy_RlRl_sat(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.H)	Word64 Q6_P_mpy_RhRh(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.H):<<1	Word64 Q6_P_mpy_RhRh_s1(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.H):<<1:rnd	Word64 Q6_P_mpy_RhRh_s1_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.H):rnd	Word64 Q6_P_mpy_RhRh_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.L)	Word64 Q6_P_mpy_RhRl(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.L):<<1	Word64 Q6_P_mpy_RhRl_s1(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.L):<<1:rnd	Word64 Q6_P_mpy_RhRl_s1_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.H,Rt.L):rnd	Word64 Q6_P_mpy_RhRl_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.H)	Word64 Q6_P_mpy_RlRh(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.H):<<1	Word64 Q6_P_mpy_RlRh_s1(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.H):<<1:rnd	Word64 Q6_P_mpy_RlRh_s1_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.H):rnd	Word64 Q6_P_mpy_RlRh_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.L)	Word64 Q6_P_mpy_RlRl(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.L):<<1	Word64 Q6_P_mpy_RlRl_s1(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.L):<<1:rnd	Word64 Q6_P_mpy_RlRl_s1_rnd(Word32 Rs, Word32 Rt)
Rdd=mpy(Rs.L,Rt.L):rnd	Word64 Q6_P_mpy_RlRl_rnd(Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.H)	Word32 Q6_R_mpyacc_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.H):<<1	Word32 Q6_R_mpyacc_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.H):<<1:sat	Word32 Q6_R_mpyacc_RhRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.H):sat	Word32 Q6_R_mpyacc_RhRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.L)	Word32 Q6_R_mpyacc_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.L):<<1	Word32 Q6_R_mpyacc_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.H,Rt.L):<<1:sat	Word32 Q6_R_mpyacc_RhRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)

Rx+=mpy(Rs.H,Rt.L):sat	Word32 Q6_R_mpyacc_RhRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.H)	Word32 Q6_R_mpyacc_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.H):<<1	Word32 Q6_R_mpyacc_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.H):<<1:sat	Word32 Q6_R_mpyacc_RlRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.H):sat	Word32 Q6_R_mpyacc_RlRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.L)	Word32 Q6_R_mpyacc_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.L):<<1	Word32 Q6_R_mpyacc_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.L):<<1:sat	Word32 Q6_R_mpyacc_RlRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpy(Rs.L,Rt.L):sat	Word32 Q6_R_mpyacc_RlRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.H)	Word32 Q6_R_mpynac_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.H):<<1	Word32 Q6_R_mpynac_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.H):<<1:sat	Word32 Q6_R_mpynac_RhRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.H):sat	Word32 Q6_R_mpynac_RhRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.L)	Word32 Q6_R_mpynac_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.L):<<1	Word32 Q6_R_mpynac_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.L):<<1:sat	Word32 Q6_R_mpynac_RhRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.H,Rt.L):sat	Word32 Q6_R_mpynac_RhRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.H)	Word32 Q6_R_mpynac_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.H):<<1	Word32 Q6_R_mpynac_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.H):<<1:sat	Word32 Q6_R_mpynac_RlRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.H):sat	Word32 Q6_R_mpynac_RlRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.L)	Word32 Q6_R_mpynac_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.L):<<1	Word32 Q6_R_mpynac_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpy(Rs.L,Rt.L):<<1:sat	Word32 Q6_R_mpynac_RlRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)

`Rx-=mpy(Rs.L,Rt.L):sat`	`Word32 Q6_R_mpynac_RlRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs.H,Rt.H)`	`Word64 Q6_P_mpyacc_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs.H,Rt.H):<<1`	`Word64 Q6_P_mpyacc_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs.H,Rt.L)`	`Word64 Q6_P_mpyacc_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs.H,Rt.L):<<1`	`Word64 Q6_P_mpyacc_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs.L,Rt.H)`	`Word64 Q6_P_mpyacc_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs.L,Rt.H):<<1`	`Word64 Q6_P_mpyacc_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs.L,Rt.L)`	`Word64 Q6_P_mpyacc_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs.L,Rt.L):<<1`	`Word64 Q6_P_mpyacc_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.H,Rt.H)`	`Word64 Q6_P_mpynac_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.H,Rt.H):<<1`	`Word64 Q6_P_mpynac_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.H,Rt.L)`	`Word64 Q6_P_mpynac_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.H,Rt.L):<<1`	`Word64 Q6_P_mpynac_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.L,Rt.H)`	`Word64 Q6_P_mpynac_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.L,Rt.H):<<1`	`Word64 Q6_P_mpynac_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.L,Rt.L)`	`Word64 Q6_P_mpynac_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs.L,Rt.L):<<1`	`Word64 Q6_P_mpynac_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5							sH	tH	d5					
1	1	1	0	0	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	0	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.L,Rt.L)[:<<N]:rnd
1	1	1	0	0	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.L,Rt.H)[:<<N]:rnd
1	1	1	0	0	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.H,Rt.L)[:<<N]:rnd
1	1	1	0	0	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs.H,Rt.H)[:<<N]:rnd

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	x5						
1	1	1	0	0	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs.H,Rt.H)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	d5						
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.L)[:<<N]:sat
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.H)[:<<N]:sat
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.L)[:<<N]:sat
1	1	1	0	1	1	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.H)[:<<N]:sat
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.L)[:<<N]:rnd
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.H)[:<<N]:rnd
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.L)[:<<N]:rnd
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.H)[:<<N]:rnd
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.L)[:<<N]:rnd:sat
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.L,Rt.H)[:<<N]:rnd:sat
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.L)[:<<N]:rnd:sat
1	1	1	0	1	1	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs.H,Rt.H)[:<<N]:rnd:sat
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	x5						
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.L,Rt.L)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.L,Rt.H)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.H,Rt.L)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs.H,Rt.H)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.L,Rt.H)[:<<N]

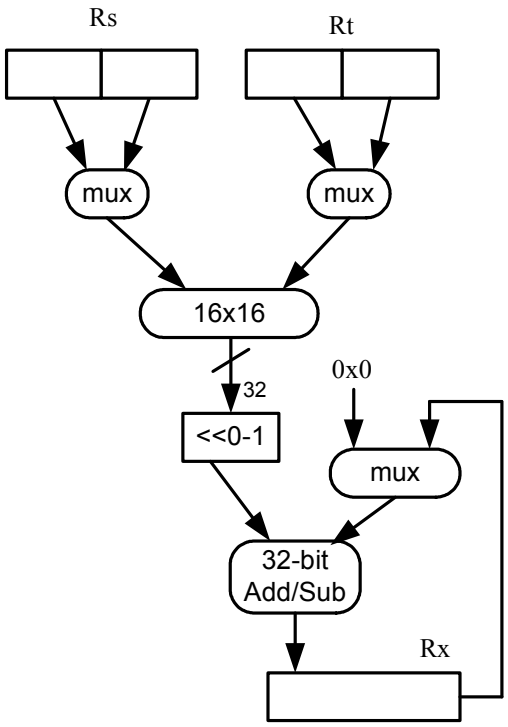
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.H,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.L,Rt.L)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.L,Rt.H)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.H,Rt.L)[:<<N]:sat
1	1	1	0	1	1	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs.H,Rt.H)[:<<N]:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
sH	Rs is High
tH	Rt is High
sH	Rs is High
tH	Rt is High
sH	Rs is High
tH	Rt is High
sH	Rs is High
tH	Rt is High
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

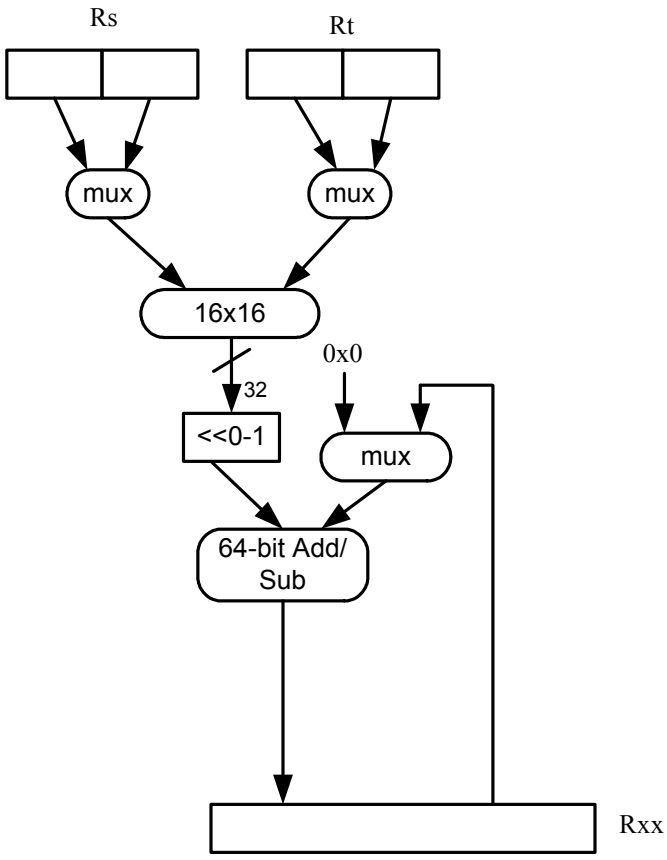
Multiply unsigned halfwords

Multiply two unsigned halfwords. Scale the result by 0-3 bits. Optionally, add or subtract the result from the accumulator.

Rx+=mpyu(Rs.[HL],Rt.[HL])[:<<1]
Rd = mpyu(Rs.[HL],Rt.[HL])[:<<1]



Rxx+=mpyu(Rs.[HL],Rt.[HL])[:<<1]
Rdd = mpyu(Rs.[HL],Rt.[HL])[:<<1]



Syntax	Behavior
`Rd=mpyu(Rs.[HL],Rt.[HL])[:<<1]`	`Rd=(Rs.uh[01] * Rt.uh[01])[<<1];`
`Rdd=mpyu(Rs.[HL],Rt.[HL])[:<<1]`	`Rdd=(Rs.uh[01] * Rt.uh[01])[<<1];`
`Rx+=mpyu(Rs.[HL],Rt.[HL])[:<<1]`	`Rx=Rx+ (Rs.uh[01] * Rt.uh[01])[<<1];`
`Rx-=mpyu(Rs.[HL],Rt.[HL])[:<<1]`	`Rx=Rx- (Rs.uh[01] * Rt.uh[01])[<<1];`
`Rxx+=mpyu(Rs.[HL],Rt.[HL])[:<<1]`	`Rxx=Rxx+ (Rs.uh[01] * Rt.uh[01])[<<1];`
`Rxx-=mpyu(Rs.[HL],Rt.[HL])[:<<1]`	`Rxx=Rxx- (Rs.uh[01] * Rt.uh[01])[<<1];`

Class: XTYPE (slots 2,3)

Intrinsics

Rd=mpyu(Rs.H,Rt.H)	UWord32 Q6_R_mpyu_RhRh(Word32 Rs, Word32 Rt)
Rd=mpyu(Rs.H,Rt.H):<<1	UWord32 Q6_R_mpyu_RhRh_s1(Word32 Rs, Word32 Rt)
Rd=mpyu(Rs.H,Rt.L)	UWord32 Q6_R_mpyu_RhRl(Word32 Rs, Word32 Rt)
Rd=mpyu(Rs.H,Rt.L):<<1	UWord32 Q6_R_mpyu_RhRl_s1(Word32 Rs, Word32 Rt)
Rd=mpyu(Rs.L,Rt.H)	UWord32 Q6_R_mpyu_RlRh(Word32 Rs, Word32 Rt)
Rd=mpyu(Rs.L,Rt.H):<<1	UWord32 Q6_R_mpyu_RlRh_s1(Word32 Rs, Word32 Rt)
Rd=mpyu(Rs.L,Rt.L)	UWord32 Q6_R_mpyu_RlRl(Word32 Rs, Word32 Rt)
Rd=mpyu(Rs.L,Rt.L):<<1	UWord32 Q6_R_mpyu_RlRl_s1(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.H,Rt.H)	UWord64 Q6_P_mpyu_RhRh(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.H,Rt.H):<<1	UWord64 Q6_P_mpyu_RhRh_s1(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.H,Rt.L)	UWord64 Q6_P_mpyu_RhRl(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.H,Rt.L):<<1	UWord64 Q6_P_mpyu_RhRl_s1(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.L,Rt.H)	UWord64 Q6_P_mpyu_RlRh(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.L,Rt.H):<<1	UWord64 Q6_P_mpyu_RlRh_s1(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.L,Rt.L)	UWord64 Q6_P_mpyu_RlRl(Word32 Rs, Word32 Rt)
Rdd=mpyu(Rs.L,Rt.L):<<1	UWord64 Q6_P_mpyu_RlRl_s1(Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.H,Rt.H)	Word32 Q6_R_mpyuacc_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.H,Rt.H):<<1	Word32 Q6_R_mpyuacc_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.H,Rt.L)	Word32 Q6_R_mpyuacc_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.H,Rt.L):<<1	Word32 Q6_R_mpyuacc_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.L,Rt.H)	Word32 Q6_R_mpyuacc_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.L,Rt.H):<<1	Word32 Q6_R_mpyuacc_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.L,Rt.L)	Word32 Q6_R_mpyuacc_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=mpyu(Rs.L,Rt.L):<<1	Word32 Q6_R_mpyuacc_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.H,Rt.H)	Word32 Q6_R_mpyunac_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.H,Rt.H):<<1	Word32 Q6_R_mpyunac_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.H,Rt.L)	Word32 Q6_R_mpyunac_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=mpyu(Rs.H,Rt.L):<<1	Word32 Q6_R_mpyunac_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)

`Rx-=mpyu(Rs.L,Rt.H)`	`Word32 Q6_R_mpyunac_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=mpyu(Rs.L,Rt.H):<<1`	`Word32 Q6_R_mpyunac_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=mpyu(Rs.L,Rt.L)`	`Word32 Q6_R_mpyunac_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=mpyu(Rs.L,Rt.L):<<1`	`Word32 Q6_R_mpyunac_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rxx+=mpyu(Rs.H,Rt.H)`	`Word64 Q6_P_mpyuacc_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpyu(Rs.H,Rt.H):<<1`	`Word64 Q6_P_mpyuacc_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpyu(Rs.H,Rt.L)`	`Word64 Q6_P_mpyuacc_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpyu(Rs.H,Rt.L):<<1`	`Word64 Q6_P_mpyuacc_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpyu(Rs.L,Rt.H)`	`Word64 Q6_P_mpyuacc_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpyu(Rs.L,Rt.H):<<1`	`Word64 Q6_P_mpyuacc_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpyu(Rs.L,Rt.L)`	`Word64 Q6_P_mpyuacc_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpyu(Rs.L,Rt.L):<<1`	`Word64 Q6_P_mpyuacc_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpyu(Rs.H,Rt.H)`	`Word64 Q6_P_mpyunac_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpyu(Rs.H,Rt.H):<<1`	`Word64 Q6_P_mpyunac_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpyu(Rs.H,Rt.L)`	`Word64 Q6_P_mpyunac_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpyu(Rs.H,Rt.L):<<1`	`Word64 Q6_P_mpyunac_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpyu(Rs.L,Rt.H)`	`Word64 Q6_P_mpyunac_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpyu(Rs.L,Rt.H):<<1`	`Word64 Q6_P_mpyunac_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpyu(Rs.L,Rt.L)`	`Word64 Q6_P_mpyunac_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpyu(Rs.L,Rt.L):<<1`	`Word64 Q6_P_mpyunac_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)`

Encoding

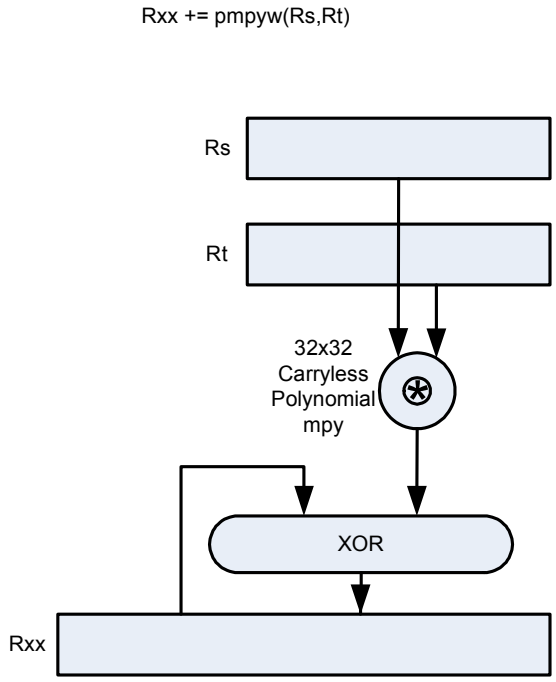
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	d5						
1	1	1	0	0	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs.H,Rt.H)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	x5						
1	1	1	0	0	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs.H,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	0	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs.H,Rt.H)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	d5						
1	1	1	0	1	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs.H,Rt.H)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5						sH	tH	x5						
1	1	1	0	1	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx+=mpyu(Rs.H,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpyu(Rs.L,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpyu(Rs.L,Rt.H)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rx-=mpyu(Rs.H,Rt.L)[:<<N]
1	1	1	0	1	1	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpyu(Rs.H,Rt.H)[:<<N]

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
sH	Rs is High
tH	Rt is High

Field name	Description
sH	Rs is High
tH	Rt is High
sH	Rs is High
tH	Rt is High
sH	Rs is High
tH	Rt is High
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Polynomial multiply words

Perform a 32x32 carryless polynomial multiply using 32-bit source registers Rs and Rt. The 64-bit result is optionally accumulated (XORed) with the destination register. Finite field multiply instructions are useful for many algorithms including scramble code generation, cryptographic algorithms, convolutional, and Reed Solomon codes.



Syntax	Behavior

`Rdd=pmpyw(Rs,Rt)`	`x = Rs.uw[0];`
	`y = Rt.uw[0];`
	`prod = 0;`
	`for(i=0; i < 32; i++) {`
	`  if((y >> i) & 1) prod ^= (x << i);`
	`};`
	`Rdd = prod;`

`Rxx^=pmpyw(Rs,Rt)`	`x = Rs.uw[0];`
	`y = Rt.uw[0];`
	`prod = 0;`
	`for(i=0; i < 32; i++) {`
	`  if((y >> i) & 1) prod ^= (x << i);`
	`};`
	`Rxx ^= prod;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=pmpyw(Rs,Rt)`	`Word64 Q6_P_pmpyw_RR(Word32 Rs, Word32 Rt)`
`Rxx^=pmpyw(Rs,Rt)`	`Word64 Q6_P_pmpywxacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`

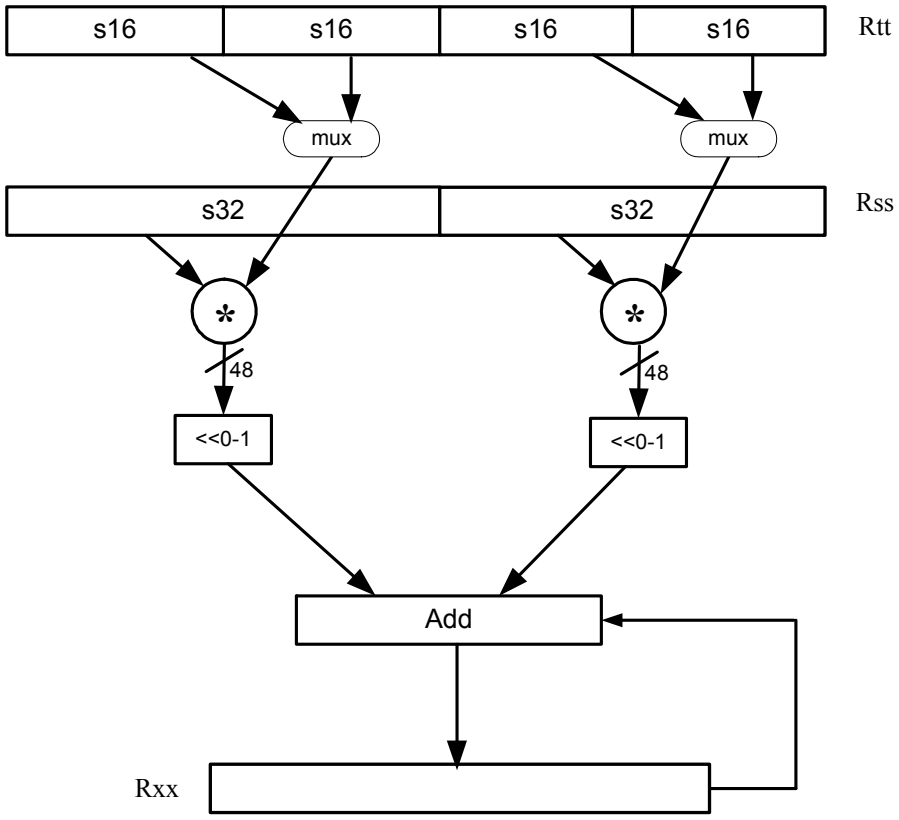
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=pmpyw(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx^=pmpyw(Rs,Rt)

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Vector reduce multiply word by signed half (32x16)

Perform mixed precision vector multiply operations and accumulate the results. A 32-bit word from vector Rss is multiplied by a 16-bit halfword (either even or odd) from vector Rtt. The multiplication is performed as a signed 32x16, which produces a 48-bit result. This result is optionally scaled left by one bit. A similar operation is performed for both words in Rss, and the two results are accumulated. The final result is optionally accumulated with Rxx.



Syntax	Behavior
`Rdd=vrmpyweh(Rss,Rtt)[:<<1]`	`Rdd = (Rss.w[1] * Rtt.h[2])[<<1] + (Rss.w[0] * Rtt.h[0])[<<1];`
`Rdd=vrmpywoh(Rss,Rtt)[:<<1]`	`Rdd = (Rss.w[1] * Rtt.h[3])[<<1] + (Rss.w[0] * Rtt.h[1])[<<1];`
`Rxx+=vrmpyweh(Rss,Rtt)[:<<1]`	`Rxx += (Rss.w[1] * Rtt.h[2])[<<1] + (Rss.w[0] * Rtt.h[0])[<<1];`
`Rxx+=vrmpywoh(Rss,Rtt)[:<<1]`	`Rxx += (Rss.w[1] * Rtt.h[3])[<<1] + (Rss.w[0] * Rtt.h[1])[<<1];`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vrmpyweh(Rss,Rtt)`	`Word64 Q6_P_vrmpyweh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vrmpyweh(Rss,Rtt):<<1`	`Word64 Q6_P_vrmpyweh_PP_s1(Word64 Rss, Word64 Rtt)`
`Rdd=vrmpywoh(Rss,Rtt)`	`Word64 Q6_P_vrmpywoh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vrmpywoh(Rss,Rtt):<<1`	`Word64 Q6_P_vrmpywoh_PP_s1(Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpyweh(Rss,Rtt)`	`Word64 Q6_P_vrmpywehacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpyweh(Rss,Rtt):<<1`	`Word64 Q6_P_vrmpywehacc_PP_s1(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpywoh(Rss,Rtt)`	`Word64 Q6_P_vrmpywohacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpywoh(Rss,Rtt):<<1`	`Word64 Q6_P_vrmpywohacc_PP_s1(Word64 Rxx, Word64 Rss, Word64 Rtt)`

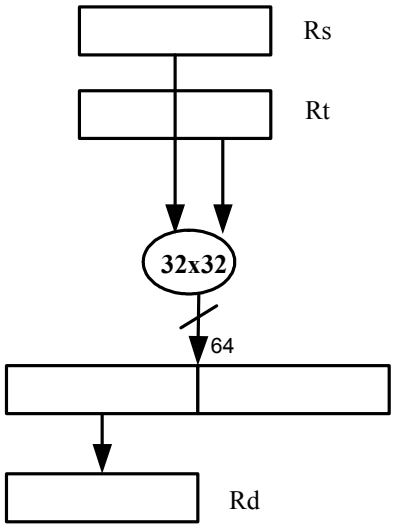
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrmpywoh(Rss,Rtt)[:<<N]
1	1	1	0	1	0	0	0	**N**	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrmpyweh(Rss,Rtt)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrmpyweh(Rss,Rtt)[:<<N]
1	1	1	0	1	0	1	0	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrmpywoh(Rss,Rtt)[:<<N]

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Multiply and use upper result

Multiply two signed or unsigned 32-bit words. Take the upper 32-bits of this results store to a single destination register. Optional rounding is available.



Syntax	Behavior
`Rd=mpy(Rs,Rt.H):<<1:rnd:sat`	`Rd = sat_32(((Rs * Rt.h[1])<<1+0x8000)>>16);`
`Rd=mpy(Rs,Rt.H):<<1:sat`	`Rd = sat_32(((Rs * Rt.h[1])<<1)>>16);`
`Rd=mpy(Rs,Rt.L):<<1:rnd:sat`	`Rd = sat_32(((Rs * Rt.h[0])<<1+0x8000)>>16);`
`Rd=mpy(Rs,Rt.L):<<1:sat`	`Rd = sat_32(((Rs * Rt.h[0])<<1)>>16);`
`Rd=mpy(Rs,Rt)`	`Rd=(Rs * Rt)>>32;`
`Rd=mpy(Rs,Rt):<<1`	`Rd=(Rs * Rt)>>31;`
`Rd=mpy(Rs,Rt):<<1:sat`	`Rd=sat_32((Rs * Rt)>>31);`
`Rd=mpy(Rs,Rt):rnd`	`Rd=((Rs * Rt)+0x80000000)>>32;`
`Rd=mpysu(Rs,Rt)`	`Rd=(Rs * Rt.uw[0])>>32;`
`Rd=mpyu(Rs,Rt)`	`Rd=(Rs.uw[0] * Rt.uw[0])>>32;`
`Rx+=mpy(Rs,Rt):<<1:sat`	`Rx=sat_32((Rx) + ((Rs * Rt)>>31));`
`Rx-=mpy(Rs,Rt):<<1:sat`	`Rx=sat_32((Rx) - ((Rs * Rt)>>31));`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=mpy(Rs,Rt.H):<<1:rnd:sat`	`Word32 Q6_R_mpy_RRh_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt.H):<<1:sat`	`Word32 Q6_R_mpy_RRh_s1_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt.L):<<1:rnd:sat`	`Word32 Q6_R_mpy_RRl_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt.L):<<1:sat`	`Word32 Q6_R_mpy_RRl_s1_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt)`	`Word32 Q6_R_mpy_RR(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt):<<1`	`Word32 Q6_R_mpy_RR_s1(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt):<<1:sat`	`Word32 Q6_R_mpy_RR_s1_sat(Word32 Rs, Word32 Rt)`
`Rd=mpy(Rs,Rt):rnd`	`Word32 Q6_R_mpy_RR_rnd(Word32 Rs, Word32 Rt)`
`Rd=mpysu(Rs,Rt)`	`Word32 Q6_R_mpysu_RR(Word32 Rs, Word32 Rt)`
`Rd=mpyu(Rs,Rt)`	`UWord32 Q6_R_mpyu_RR(Word32 Rs, Word32 Rt)`
`Rx+=mpy(Rs,Rt):<<1:sat`	`Word32 Q6_R_mpyacc_RR_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)`
`Rx-=mpy(Rs,Rt):<<1:sat`	`Word32 Q6_R_mpynac_RR_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)`

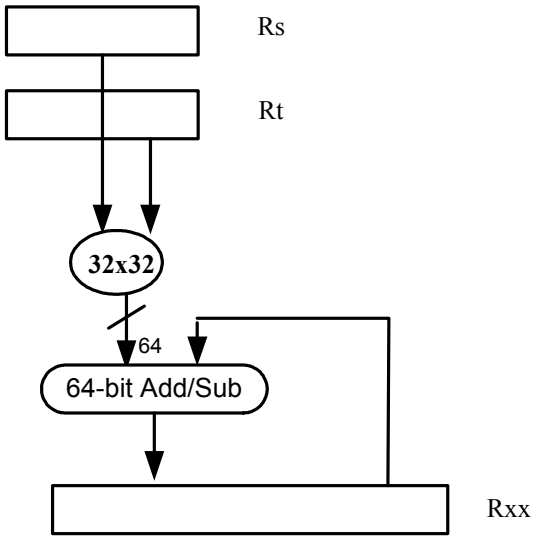
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt):rnd
1	1	1	0	1	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpyu(Rs,Rt)
1	1	1	0	1	1	0	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpysu(Rs,Rt)
1	1	1	0	1	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt.H):<<1:sat
1	1	1	0	1	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt.L):<<1:sat
1	1	1	0	1	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt.H):<<1:rnd:sat
1	1	1	0	1	1	0	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt):<<1:sat
1	1	1	0	1	1	0	1	1	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt.L):<<1:rnd:sat
1	1	1	0	1	1	0	1	**N**	0	**N**	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	**N**	**N**	**d**	**d**	**d**	**d**	**d**	Rd=mpy(Rs,Rt)[:<<N]
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	1	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rx+=mpy(Rs,Rt):<<1:sat
1	1	1	0	1	1	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rx-=mpy(Rs,Rt):<<1:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Multiply and use full result

Multiply two signed or unsigned 32-bit words. Optionally, add or subtract this value from the 64-bit accumulator. The result is a full-precision 64-bit value.



Syntax	Behavior
`Rdd=mpy(Rs,Rt)`	`Rdd=(Rs * Rt);`
`Rdd=mpyu(Rs,Rt)`	`Rdd=(Rs.uw[0] * Rt.uw[0]);`
`Rxx[+-]=mpy(Rs,Rt)`	`Rxx= Rxx[+-](Rs * Rt);`
`Rxx[+-]=mpyu(Rs,Rt)`	`Rxx= Rxx[+-](Rs.uw[0] * Rt.uw[0]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=mpy(Rs,Rt)`	`Word64 Q6_P_mpy_RR(Word32 Rs, Word32 Rt)`
`Rdd=mpyu(Rs,Rt)`	`UWord64 Q6_P_mpyu_RR(Word32 Rs, Word32 Rt)`
`Rxx+=mpy(Rs,Rt)`	`Word64 Q6_P_mpyacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=mpyu(Rs,Rt)`	`Word64 Q6_P_mpyuacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpy(Rs,Rt)`	`Word64 Q6_P_mpynac_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx-=mpyu(Rs,Rt)`	`Word64 Q6_P_mpyunac_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`

Encoding

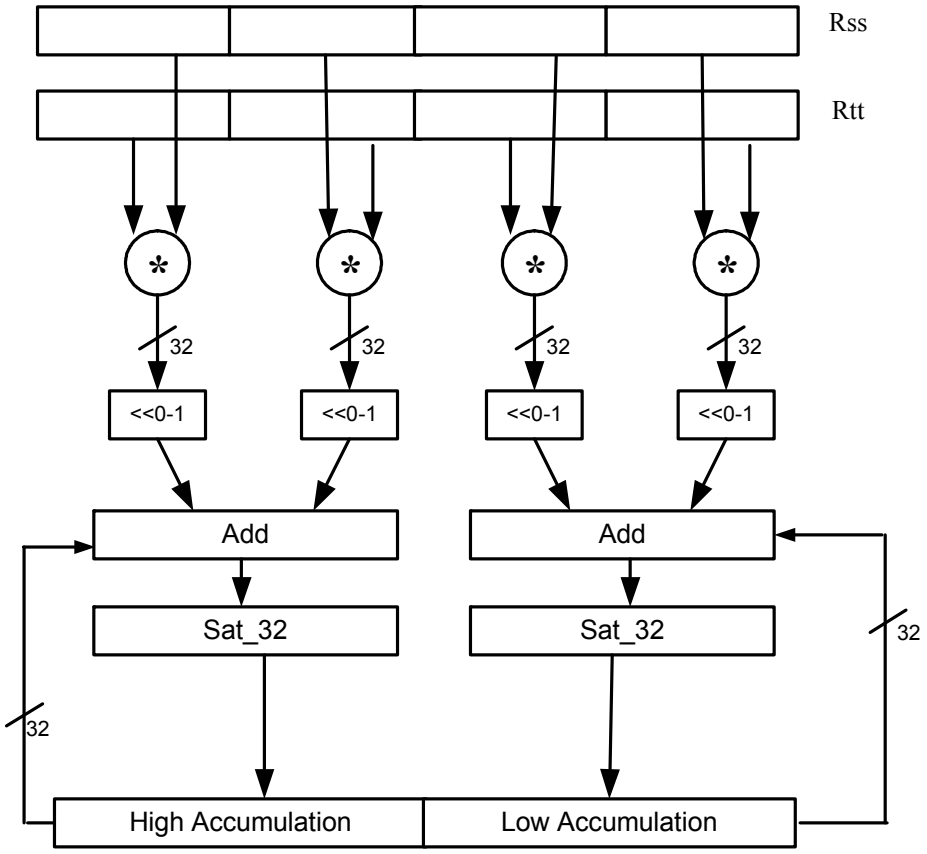
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5						
1	1	1	0	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpy(Rs,Rt)
1	1	1	0	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=mpyu(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpy(Rs,Rt)
1	1	1	0	0	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpy(Rs,Rt)
1	1	1	0	0	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=mpyu(Rs,Rt)
1	1	1	0	0	1	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	**x**	**x**	**x**	**x**	**x**	Rxx-=mpyu(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector dual multiply

Multiply four 16-bit halfwords in Rss by the corresponding 16-bit halfwords in Rtt. The two lower results are scaled and added. The lower word of the accumulator is optionally added. This result is saturated to 32-bits and stored in the lower word of the accumulator. The same operation is performed on the upper two products using the upper word of the accumulator.

Rxx+=vdmpy(Rss,Rtt):sat



Syntax	Behavior
`Rdd=vdmpy(Rss,Rtt):<<1:sat`	`Rdd.w[0]=sat_32((Rss.h[0] * Rtt.h[0])<<1 + (Rss.h[1] * Rtt.h[1])<<1);` `Rdd.w[1]=sat_32((Rss.h[2] * Rtt.h[2])<<1 + (Rss.h[3] * Rtt.h[3])<<1);`
`Rdd=vdmpy(Rss,Rtt):sat`	`Rdd.w[0]=sat_32((Rss.h[0] * Rtt.h[0])<<0 + (Rss.h[1] * Rtt.h[1])<<0);` `Rdd.w[1]=sat_32((Rss.h[2] * Rtt.h[2])<<0 + (Rss.h[3] * Rtt.h[3])<<0);`
`Rxx+=vdmpy(Rss,Rtt):<<1:sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0])<<1 + (Rss.h[1] * Rtt.h[1])<<1);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2])<<1 + (Rss.h[3] * Rtt.h[3])<<1);`
`Rxx+=vdmpy(Rss,Rtt):sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0])<<0 + (Rss.h[1] * Rtt.h[1])<<0);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2])<<0 + (Rss.h[3] * Rtt.h[3])<<0);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vdmpy(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vdmpy_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vdmpy(Rss,Rtt):sat`	`Word64 Q6_P_vdmpy_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+=vdmpy(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vdmpyacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vdmpy(Rss,Rtt):sat`	`Word64 Q6_P_vdmpyacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

Encoding

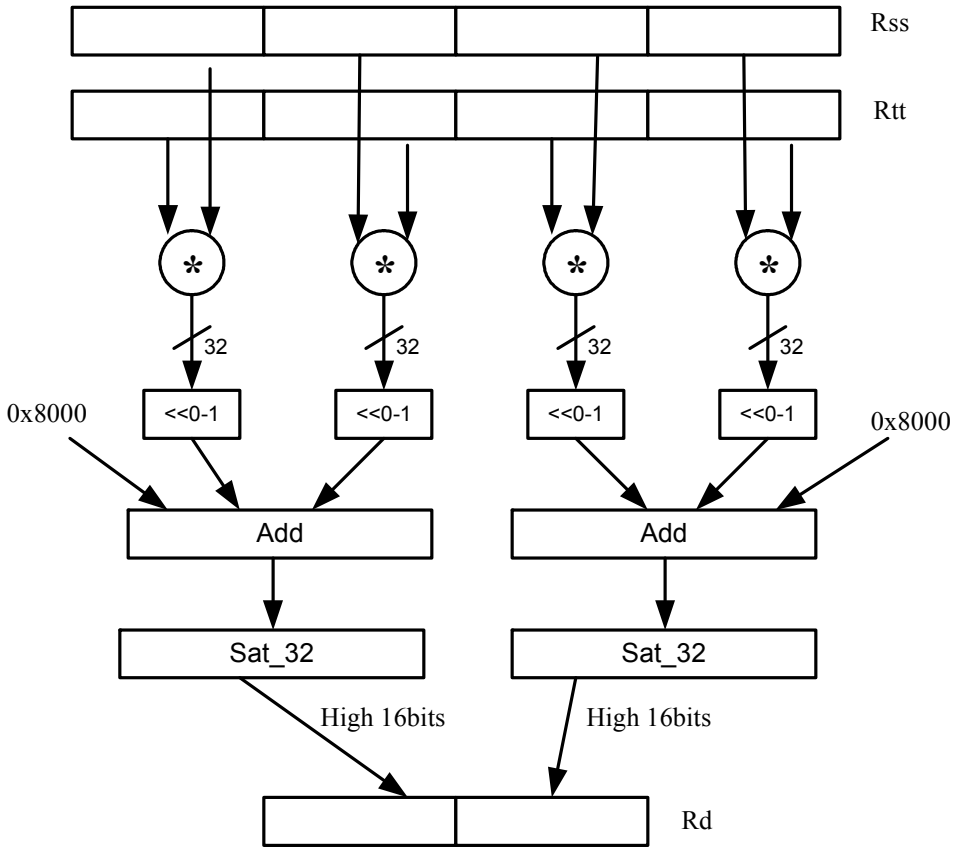
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vdmpy(Rss,Rtt)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vdmpy(Rss,Rtt)[:<<N]:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector dual multiply with round and pack

Multiply four 16-bit halfwords in Rss by the corresponding 16-bit halfwords in Rtt. The two lower results are scaled and added together with a rounding constant. This result is saturated to 32-bits, and the upper 16-bits of this result are stored in the lower 16-bits of the destination register. The same operation is performed on the upper two products and the result is stored in the upper 16-bit halfword of the destination.

Rd=vdmpy(Rss,Rtt):rnd:sat



Syntax	Behavior
`Rd=vdmpy(Rss,Rtt)[:<<1]:rnd:sat`	`Rd.h[0]=(sat_32((Rss.h[0] * Rtt.h[0])[<<1] + (Rss.h[1] * Rtt.h[1])[<<1] + 0x8000)).h[1];` `Rd.h[1]=(sat_32((Rss.h[2] * Rtt.h[2])[<<1] + (Rss.h[3] * Rtt.h[3])[<<1] + 0x8000)).h[1];`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vdmpy(Rss,Rtt):<<1:rnd:sat`	`Word32 Q6_R_vdmpy_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)`
`Rd=vdmpy(Rss,Rtt):rnd:sat`	`Word32 Q6_R_vdmpy_PP_rnd_sat(Word64 Rss, Word64 Rtt)`

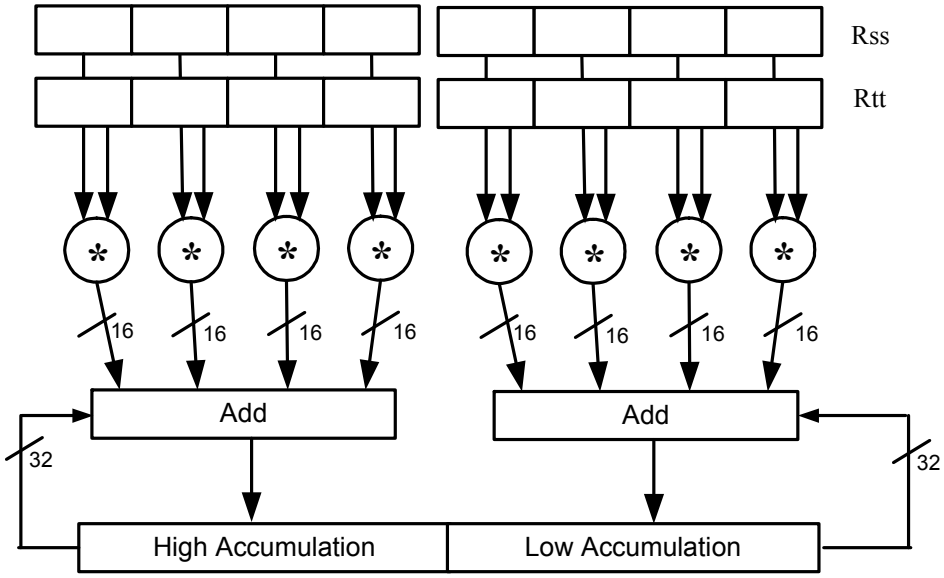
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	1	**N**	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	-	0	0	**d**	**d**	**d**	**d**	**d**	Rd=vdmpy(Rss,Rtt)[:<<N]:rnd:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector reduce multiply bytes

Multiply eight 8-bit bytes in Rss by the corresponding 8-bit bytes in Rtt. The four lower results are accumulated. The lower word of the accumulator is optionally added. This result is stored in the lower 32-bits of the accumulator. The same operation is performed on the upper four products using the upper word of the accumulator. The eight bytes of Rss can be treated as either signed or unsigned.



Syntax	Behavior
`Rdd=vrmpybsu(Rss,Rtt)`	`Rdd.w[0]=((Rss.b[0] * Rtt.ub[0]) + (Rss.b[1] * Rtt.ub[1]) + (Rss.b[2] * Rtt.ub[2]) + (Rss.b[3] * Rtt.ub[3]));` `Rdd.w[1]=((Rss.b[4] * Rtt.ub[4]) + (Rss.b[5] * Rtt.ub[5]) + (Rss.b[6] * Rtt.ub[6]) + (Rss.b[7] * Rtt.ub[7]));`
`Rdd=vrmpybu(Rss,Rtt)`	`Rdd.w[0]=((Rss.ub[0] * Rtt.ub[0]) + (Rss.ub[1] * Rtt.ub[1]) + (Rss.ub[2] * Rtt.ub[2]) + (Rss.ub[3] * Rtt.ub[3]));` `Rdd.w[1]=((Rss.ub[4] * Rtt.ub[4]) + (Rss.ub[5] * Rtt.ub[5]) + (Rss.ub[6] * Rtt.ub[6]) + (Rss.ub[7] * Rtt.ub[7]));`
`Rxx+=vrmpybsu(Rss,Rtt)`	`Rxx.w[0]=(Rxx.w[0] + (Rss.b[0] * Rtt.ub[0]) + (Rss.b[1] * Rtt.ub[1]) + (Rss.b[2] * Rtt.ub[2]) + (Rss.b[3] * Rtt.ub[3]));` `Rxx.w[1]=(Rxx.w[1] + (Rss.b[4] * Rtt.ub[4]) + (Rss.b[5] * Rtt.ub[5]) + (Rss.b[6] * Rtt.ub[6]) + (Rss.b[7] * Rtt.ub[7]));`
`Rxx+=vrmpybu(Rss,Rtt)`	`Rxx.w[0]=(Rxx.w[0] + (Rss.ub[0] * Rtt.ub[0]) + (Rss.ub[1] * Rtt.ub[1]) + (Rss.ub[2] * Rtt.ub[2]) + (Rss.ub[3] * Rtt.ub[3]));` `Rxx.w[1]=(Rxx.w[1] + (Rss.ub[4] * Rtt.ub[4]) + (Rss.ub[5] * Rtt.ub[5]) + (Rss.ub[6] * Rtt.ub[6]) + (Rss.ub[7] * Rtt.ub[7]));`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vrmpybsu(Rss,Rtt)`	`Word64 Q6_P_vrmpybsu_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vrmpybu(Rss,Rtt)`	`Word64 Q6_P_vrmpybu_PP(Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpybsu(Rss,Rtt)`	`Word64 Q6_P_vrmpybsuacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpybu(Rss,Rtt)`	`Word64 Q6_P_vrmpybuacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

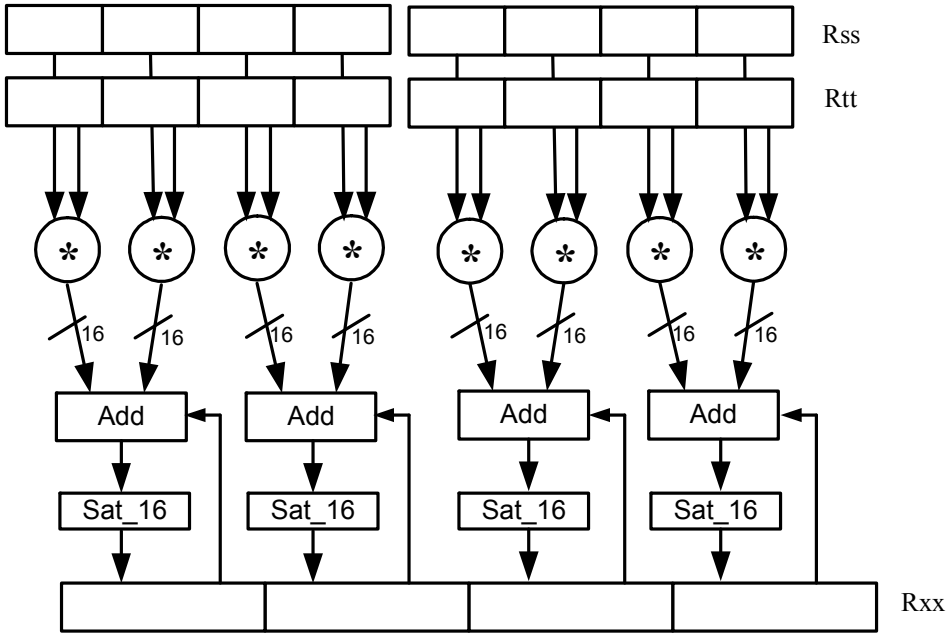
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vrmpybu(Rss,Rtt)
1	1	1	0	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vrmpybsu(Rss,Rtt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vrmpybu(Rss,Rtt)
1	1	1	0	1	0	1	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vrmpybsu(Rss,Rtt)

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Vector dual multiply signed by unsigned bytes

Multiply eight 8-bit signed bytes in Rss by the corresponding 8-bit unsigned bytes in Rtt. Add the results in pairs, and optionally add the accumulator. The results are saturated to signed 16-bits and stored in the four halfwords of the destination register.



Syntax	Behavior

`Rdd=vdmpybsu(Rss,Rtt):sat`	`Rdd.h[0]=sat_16(((Rss.b[0] * Rtt.ub[0]) + (Rss.b[1] * Rtt.ub[1])));`
	`Rdd.h[1]=sat_16(((Rss.b[2] * Rtt.ub[2]) + (Rss.b[3] * Rtt.ub[3])));`
	`Rdd.h[2]=sat_16(((Rss.b[4] * Rtt.ub[4]) + (Rss.b[5] * Rtt.ub[5])));`
	`Rdd.h[3]=sat_16(((Rss.b[6] * Rtt.ub[6]) + (Rss.b[7] * Rtt.ub[7])));`

`Rxx+=vdmpybsu(Rss,Rtt):sat`	`Rxx.h[0]=sat_16((Rxx.h[0] + (Rss.b[0] * Rtt.ub[0]) + (Rss.b[1] * Rtt.ub[1])));`
	`Rxx.h[1]=sat_16((Rxx.h[1] + (Rss.b[2] * Rtt.ub[2]) + (Rss.b[3] * Rtt.ub[3])));`
	`Rxx.h[2]=sat_16((Rxx.h[2] + (Rss.b[4] * Rtt.ub[4]) + (Rss.b[5] * Rtt.ub[5])));`
	`Rxx.h[3]=sat_16((Rxx.h[3] + (Rss.b[6] * Rtt.ub[6]) + (Rss.b[7] * Rtt.ub[7])));`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vdmpybsu(Rss,Rtt):sat`	`Word64 Q6_P_vdmpybsu_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+=vdmpybsu(Rss,Rtt):sat`	`Word64 Q6_P_vdmpybsuacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

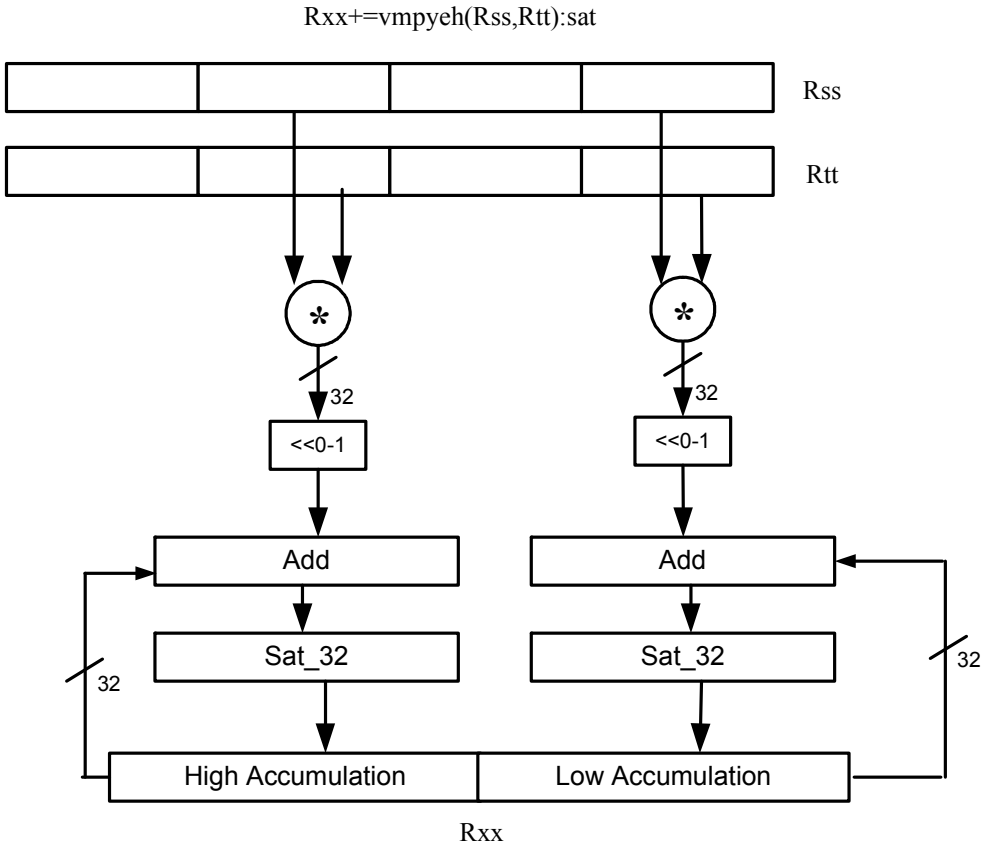
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vdmpybsu(Rss,Rtt):sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vdmpybsu(Rss,Rtt):sat

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Vector multiply even halfwords

Multiply the even 16-bit halfwords from Rss and Rtt separately. Optionally accumulate with the low and high words of the destination register pair and optionally saturate.



Syntax	Behavior
`Rdd=vmpyeh(Rss,Rtt):<<1:sat`	`Rdd.w[0]=sat_32((Rss.h[0] * Rtt.h[0])<<1);` `Rdd.w[1]=sat_32((Rss.h[2] * Rtt.h[2])<<1);`
`Rdd=vmpyeh(Rss,Rtt):sat`	`Rdd.w[0]=sat_32((Rss.h[0] * Rtt.h[0])<<0);` `Rdd.w[1]=sat_32((Rss.h[2] * Rtt.h[2])<<0);`
`Rxx+=vmpyeh(Rss,Rtt)`	`Rxx.w[0]=Rxx.w[0] + (Rss.h[0] * Rtt.h[0]);` `Rxx.w[1]=Rxx.w[1] + (Rss.h[2] * Rtt.h[2]);`
`Rxx+=vmpyeh(Rss,Rtt):<<1:sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0])<<1);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2])<<1);`
`Rxx+=vmpyeh(Rss,Rtt):sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rss.h[0] * Rtt.h[0])<<0);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rss.h[2] * Rtt.h[2])<<0);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vmpyeh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpyeh_PP_s1_sat(Word64 Rss, Word64 Rtt)`
`Rdd=vmpyeh(Rss,Rtt):sat`	`Word64 Q6_P_vmpyeh_PP_sat(Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyeh(Rss,Rtt)`	`Word64 Q6_P_vmpyehacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyeh(Rss,Rtt):<<1:sat`	`Word64 Q6_P_vmpyehacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`
`Rxx+=vmpyeh(Rss,Rtt):sat`	`Word64 Q6_P_vmpyehacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)`

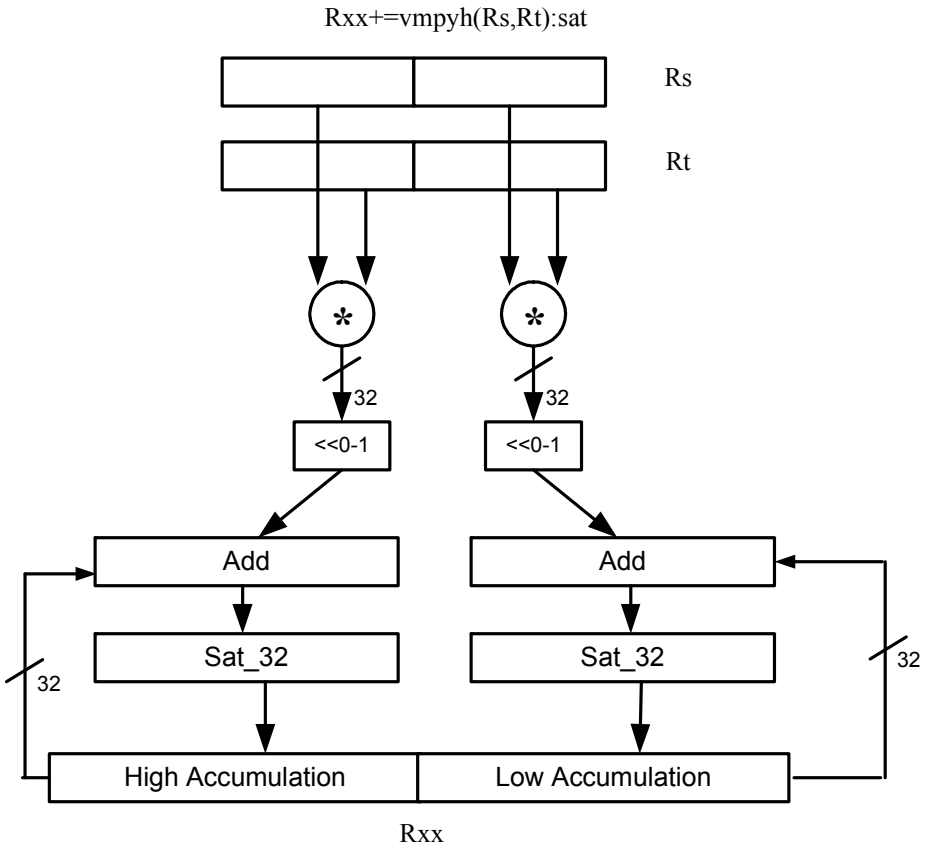
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyeh(Rss,Rtt)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyeh(Rss,Rtt)
1	1	1	0	1	0	1	0	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyeh(Rss,Rtt)[:<<N]:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector multiply halfwords

Multiply two 16-bit halfwords separately, and optionally accumulate with the low and high words of the destination. Optionally saturate, and store the results back to the destination register pair.



Syntax	Behavior
`Rdd=vmpyh(Rs,Rt)[:<<1]:sat`	`Rdd.w[0]=sat_32((Rs.h[0] * Rt.h[0])[<<1]);` `Rdd.w[1]=sat_32((Rs.h[1] * Rt.h[1])[<<1]);`
`Rxx+=vmpyh(Rs,Rt)`	`Rxx.w[0]=Rxx.w[0] + (Rs.h[0] * Rt.h[0]);` `Rxx.w[1]=Rxx.w[1] + (Rs.h[1] * Rt.h[1]);`
`Rxx+=vmpyh(Rs,Rt)[:<<1]:sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rs.h[0] * Rt.h[0])[<<1]);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rs.h[1] * Rt.h[1])[<<1]);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vmpyh(Rs,Rt):<<1:sat`	`Word64 Q6_P_vmpyh_RR_s1_sat(Word32 Rs, Word32 Rt)`
`Rdd=vmpyh(Rs,Rt):sat`	`Word64 Q6_P_vmpyh_RR_sat(Word32 Rs, Word32 Rt)`
`Rxx+=vmpyh(Rs,Rt)`	`Word64 Q6_P_vmpyhacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=vmpyh(Rs,Rt):<<1:sat`	`Word64 Q6_P_vmpyhacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=vmpyh(Rs,Rt):sat`	`Word64 Q6_P_vmpyhacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`

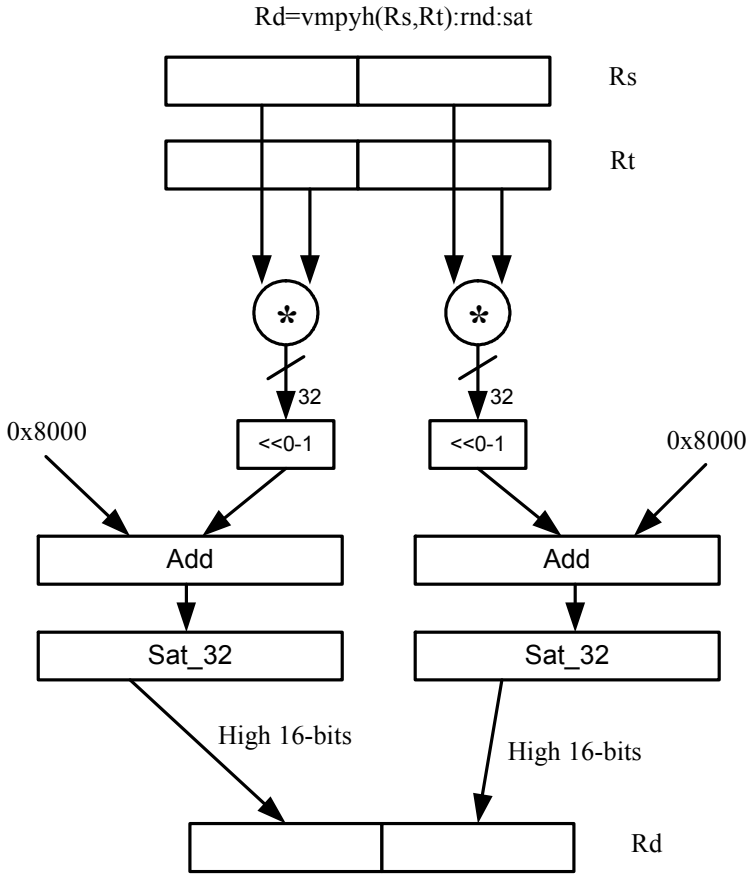
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyh(Rs,Rt)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyh(Rs,Rt)
1	1	1	0	0	1	1	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyh(Rs,Rt)[:<<N]:sat

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector multiply halfwords with round and pack

Multiply two 16-bit halfwords separately. Round the results, and store the high halfwords packed in a single register destination.



Syntax	Behavior
`Rd=vmpyh(Rs,Rt)[:<<1]:rnd:sat`	`Rd.h[1]=(sat_32((Rs.h[1] * Rt.h[1])[<<1] + 0x8000)).h[1];` `Rd.h[0]=(sat_32((Rs.h[0] * Rt.h[0])[<<1] + 0x8000)).h[1];`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vmpyh(Rs,Rt):<<1:rnd:sat`	`Word32 Q6_R_vmpyh_RR_s1_rnd_sat(Word32 Rs, Word32 Rt)`
`Rd=vmpyh(Rs,Rt):rnd:sat`	`Word32 Q6_R_vmpyh_RR_rnd_sat(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	1	0	1	**N**	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=vmpyh(Rs,Rt)[:<<N]:rnd:sat

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector multiply halfwords, signed by unsigned

Multiply two 16-bit halfwords. Rs is considered signed, Ru unsigned.

Syntax	Behavior
`Rdd=vmpyhsu(Rs,Rt)[:<<1]:sat`	`Rdd.w[0]=sat_32((Rs.h[0] * Rt.uh[0])[<<1]);` `Rdd.w[1]=sat_32((Rs.h[1] * Rt.uh[1])[<<1]);`
`Rxx+=vmpyhsu(Rs,Rt)[:<<1]:sat`	`Rxx.w[0]=sat_32(Rxx.w[0] + (Rs.h[0] * Rt.uh[0])[<<1]);` `Rxx.w[1]=sat_32(Rxx.w[1] + (Rs.h[1] * Rt.uh[1])[<<1]);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vmpyhsu(Rs,Rt):<<1:sat`	`Word64 Q6_P_vmpyhsu_RR_s1_sat(Word32 Rs, Word32 Rt)`
`Rdd=vmpyhsu(Rs,Rt):sat`	`Word64 Q6_P_vmpyhsu_RR_sat(Word32 Rs, Word32 Rt)`
`Rxx+=vmpyhsu(Rs,Rt):<<1:sat`	`Word64 Q6_P_vmpyhsuacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=vmpyhsu(Rs,Rt):sat`	`Word64 Q6_P_vmpyhsuacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	**N**	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpyhsu(Rs,Rt)[:<<N]:sat
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	**N**	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpyhsu(Rs,Rt)[:<<N]:sat

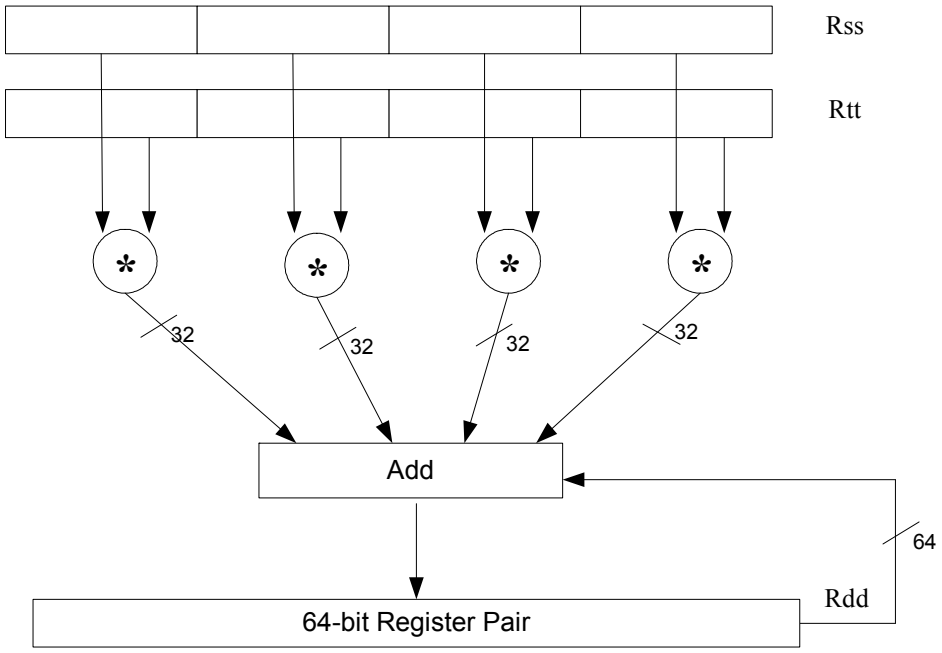
Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d

Field name	Description
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector reduce multiply halfwords

Multiply each halfword of Rss by the corresponding halfword in Rtt. Add the intermediate products together and then optionally add the accumulator. Store the full 64-bit result in the destination register pair.

This instruction is affectionately known as "big mac".



Syntax	Behavior
`Rdd=vrmpyh(Rss,Rtt)`	`Rdd = (Rss.h[0] * Rtt.h[0]) + (Rss.h[1] * Rtt.h[1]) + (Rss.h[2] * Rtt.h[2]) + (Rss.h[3] * Rtt.h[3]);`
`Rxx+=vrmpyh(Rss,Rtt)`	`Rxx = Rxx + (Rss.h[0] * Rtt.h[0]) + (Rss.h[1] * Rtt.h[1]) + (Rss.h[2] * Rtt.h[2]) + (Rss.h[3] * Rtt.h[3]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vrmpyh(Rss,Rtt)`	`Word64 Q6_P_vrmpyh_PP(Word64 Rss, Word64 Rtt)`
`Rxx+=vrmpyh(Rss,Rtt)`	`Word64 Q6_P_vrmpyhacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)`

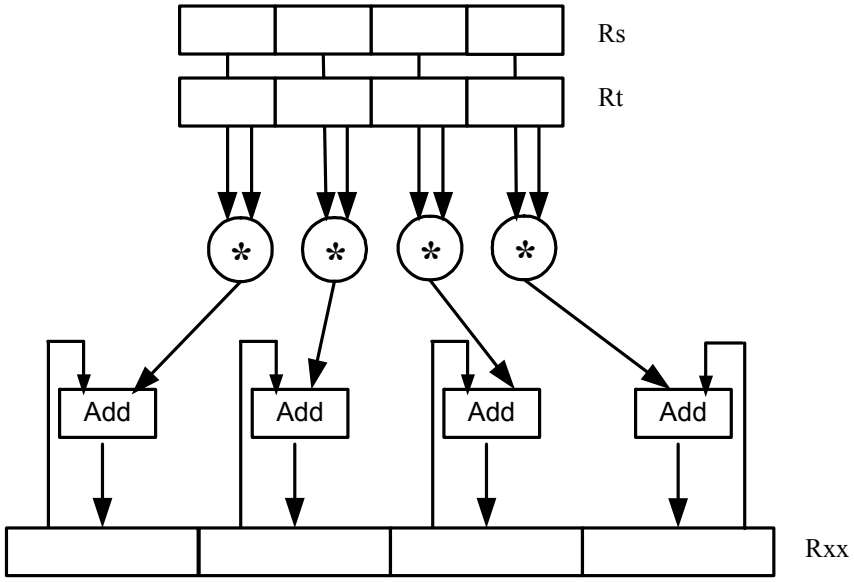
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vrmpyh(Rss,Rtt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	1	0	1	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	**x**	**x**	**x**	**x**	**x**	Rxx+=vrmpyh(Rss,Rtt)

Field name	Description
ICLASS	Instruction Class
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x

Vector multiply bytes

Four 8-bit bytes from register Rs are multiplied by four 8-bit bytes from Rt. The product is optionally accumulated with the 16-bit value from the destination register. The 16-bit results are packed in the destination register pair. The bytes of Rs can be treated as either signed or unsigned.



Syntax	Behavior

`Rdd=vmpybsu(Rs,Rt)`	`Rdd.h[0]=((Rs.b[0] * Rt.ub[0]));`
	`Rdd.h[1]=((Rs.b[1] * Rt.ub[1]));`
	`Rdd.h[2]=((Rs.b[2] * Rt.ub[2]));`
	`Rdd.h[3]=((Rs.b[3] * Rt.ub[3]));`

`Rdd=vmpybu(Rs,Rt)`	`Rdd.h[0]=((Rs.ub[0] * Rt.ub[0]));`
	`Rdd.h[1]=((Rs.ub[1] * Rt.ub[1]));`
	`Rdd.h[2]=((Rs.ub[2] * Rt.ub[2]));`
	`Rdd.h[3]=((Rs.ub[3] * Rt.ub[3]));`

`Rxx+=vmpybsu(Rs,Rt)`	`Rxx.h[0]=(Rxx.h[0]+(Rs.b[0] * Rt.ub[0]));`
	`Rxx.h[1]=(Rxx.h[1]+(Rs.b[1] * Rt.ub[1]));`
	`Rxx.h[2]=(Rxx.h[2]+(Rs.b[2] * Rt.ub[2]));`
	`Rxx.h[3]=(Rxx.h[3]+(Rs.b[3] * Rt.ub[3]));`

`Rxx+=vmpybu(Rs,Rt)`	`Rxx.h[0]=(Rxx.h[0]+(Rs.ub[0] * Rt.ub[0]));`
	`Rxx.h[1]=(Rxx.h[1]+(Rs.ub[1] * Rt.ub[1]));`
	`Rxx.h[2]=(Rxx.h[2]+(Rs.ub[2] * Rt.ub[2]));`
	`Rxx.h[3]=(Rxx.h[3]+(Rs.ub[3] * Rt.ub[3]));`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vmpybsu(Rs,Rt)`	`Word64 Q6_P_vmpybsu_RR(Word32 Rs, Word32 Rt)`
`Rdd=vmpybu(Rs,Rt)`	`Word64 Q6_P_vmpybu_RR(Word32 Rs, Word32 Rt)`
`Rxx+=vmpybsu(Rs,Rt)`	`Word64 Q6_P_vmpybsuacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`
`Rxx+=vmpybu(Rs,Rt)`	`Word64 Q6_P_vmpybuacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`

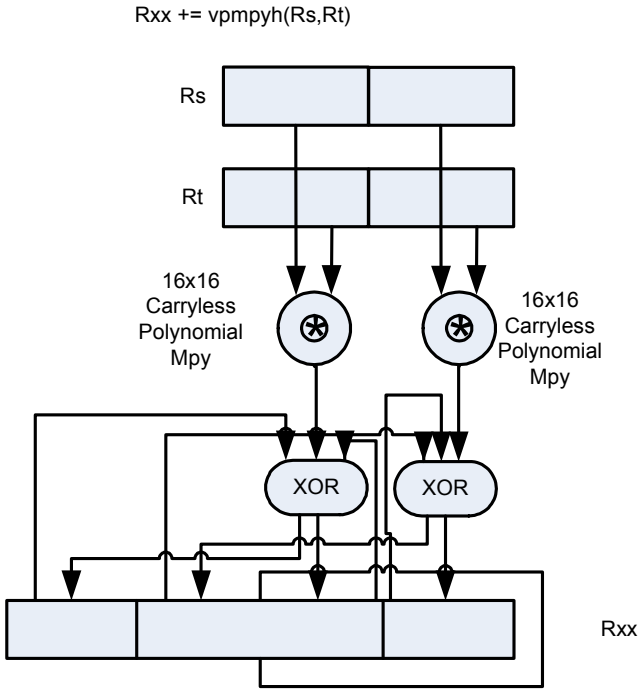
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpybsu(Rs,Rt)
1	1	1	0	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vmpybu(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpybu(Rs,Rt)
1	1	1	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	**x**	**x**	**x**	**x**	**x**	Rxx+=vmpybsu(Rs,Rt)

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

Vector polynomial multiply halfwords

Perform a vector 16x16 carryless polynomial multiply using 32-bit source registers Rs and Rt. The 64-bit result is stored in packed H,H,L,L format in the destination register. The destination register can also be optionally accumulated (XORed). Finite field multiply instructions are useful for many algorithms including scramble code generation, cryptographic algorithms, convolutional, and Reed Solomon codes.



Syntax	Behavior

`Rdd=vpmpyh(Rs,Rt)`	`x0 = Rs.uh[0];`
	`x1 = Rs.uh[1];`
	`y0 = Rt.uh[0];`
	`y1 = Rt.uh[1];`
	`prod0 = prod1 = 0;`
	`for(i=0; i < 16; i++) {`
	`  if((y0 >> i) & 1) prod0 ^= (x0 << i);`
	`  if((y1 >> i) & 1) prod1 ^= (x1 << i);`
	`};`
	`Rdd.h[0]=prod0.uh[0];`
	`Rdd.h[1]=prod1.uh[0];`
	`Rdd.h[2]=prod0.uh[1];`
	`Rdd.h[3]=prod1.uh[1];`

`Rxx^=vpmpyh(Rs,Rt)`	`x0 = Rs.uh[0];`
	`x1 = Rs.uh[1];`
	`y0 = Rt.uh[0];`
	`y1 = Rt.uh[1];`
	`prod0 = prod1 = 0;`
	`for(i=0; i < 16; i++) {`
	`  if((y0 >> i) & 1) prod0 ^= (x0 << i);`
	`  if((y1 >> i) & 1) prod1 ^= (x1 << i);`
	`};`
	`Rxx.h[0]=Rxx.uh[0] ^ prod0.uh[0];`
	`Rxx.h[1]=Rxx.uh[1] ^ prod1.uh[0];`
	`Rxx.h[2]=Rxx.uh[2] ^ prod0.uh[1];`
	`Rxx.h[3]=Rxx.uh[3] ^ prod1.uh[1];`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vpmpyh(Rs,Rt)`	`Word64 Q6_P_vpmpyh_RR(Word32 Rs, Word32 Rt)`
`Rxx^=vpmpyh(Rs,Rt)`	`Word64 Q6_P_vpmpyhxacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			d5					
1	1	1	0	0	1	0	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vpmpyh(Rs,Rt)
ICLASS				RegType				MajOp			s5					Parse			t5					MinOp			x5					
1	1	1	0	0	1	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	**x**	**x**	**x**	**x**	**x**	Rxx^=vpmpyh(Rs,Rt)

Field name	Description
`ICLASS`	Instruction Class
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x

11.12.6 XTYPE/PERM

The XTYPE/PERM instruction subclass includes instructions which perform permutations.

CABAC decode bin

This is a special-purpose instruction to support H.264 Context Adaptive Binary Arithmetic Coding (CABAC). See Section X.X for a complete description.

Syntax	Behavior

`Rdd=decbin(Rss,Rtt)`	`state = Rtt.w[1][5:0];`
	`valMPS = Rtt.w[1][8:8];`
	`bitpos = Rtt.w[0][4:0];`
	`range = Rss.w[0];`
	`offset = Rss.w[1];`
	`range <<= bitpos;`
	`offset <<= bitpos;`
	`rLPS = rLPS_table_64x4[state][ (range >>29)&3];`
	`rLPS = rLPS << 23;`
	`rMPS= (range&0xff800000) - rLPS;`
	`if (offset < rMPS) {`
	`  Rdd = AC_next_state_MPS_64[state];`
	`  Rdd[8:8]=valMPS;`
	`  Rdd[31:23]=(rMPS>>23);`
	`  Rdd.w[1]=offset;`
	`  P0=valMPS;`
	`} else {`
	`  Rdd = AC_next_state_LPS_64[state];`
	`  Rdd[8:8]=((!state)?(1-valMPS):(valMPS));`
	`  Rdd[31:23]=(rLPS>>23);`
	`  Rdd.w[1]=(offset-rMPS);`
	`  P0=(valMPS^1);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- The predicate generated by this instruction can not be used as a .new predicate, nor can it be automatically ANDed with another predicate.

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	1	1	-	s	s	s	s	s	P	P	-	t	t	t	t	t	1	1	-	d	d	d	d	d	Rdd=decbin(Rss,Rtt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

CABAC encode bin

This is a special-purpose instruction to support H.264 Context Adaptive Binary Arithmetic Coding (CABAC). See Section X.X for a complete description.

Syntax	Behavior

`Rdd=encbin(Rss,Rtt,Pu)`	`state = Rtt.w[1][5:0];`
	`valMPS = Rtt.w[1][8:8];`
	`bitpos = Rtt.w[0][4:0];`
	`range = Rss.w[0];`
	`low = Rss.w[1];`
	`bin = Pu[0];`
	`range <<= bitpos;`
	`range &= 0xFF800000U;`
	`rLPS = rLPS_table_64x4[state][(range>>29)&3];`
	`rLPS = rLPS << 23;`
	`rMPS = range - rLPS;`
	`if (bin == valMPS) {`
	`  Rdd = AC_next_state_MPS_64[state];`
	`  Rdd[8:8]=valMPS;`
	`  Rdd[31:23]=(rMPS>>23);`
	`  Rdd.w[1]=low;`
	`} else {`
	`  Rdd = AC_next_state_LPS_64[state];`
	`  Rdd[8:8]=((!state)?(1-valMPS):(valMPS));`
	`  Rdd[31:23]=(rLPS>>23);`
	`  Rdd.w[1]=(low+(rMPS >> bitpos));`
	`};`

Class: XTYPE (slots 2,3)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5						u2		d5						
1	1	0	0	0	0	1	0	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	**u**	**u**	**d**	**d**	**d**	**d**	**d**	Rdd=encbin(Rss,Rtt,Pu)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Saturate

Saturate a single scalar value.

sath saturates a signed 32-bit number to a signed 16-bit number, which is sign-extended back to 32 bits and placed in the destination register. The minimum negative value of the result is 0xffff8000 and the maximum positive value is 0x00007fff.

satuh saturates a signed 32-bit number to an unsigned 16-bit number, which is zero-extended back to 32 bits and placed in the destination register. The minimum value of the result is 0 and the maximum value is 0x0000ffff.

satb saturates a signed 32-bit number to an signed 8-bit number, which is sign-extended back to 32 bits and placed in the destination register. The minimum value of the result is 0xffffff80 and the maximum value is 0x0000007f.

satub saturates a signed 32-bit number to an unsigned 8-bit number, which is zero-extended back to 32 bits and placed in the destination register. The minimum value of the result is 0 and the maximum value is 0x000000ff.

Syntax	Behavior
`Rd=sat(Rss)`	`Rd = sat_32(Rss);`
`Rd=satb(Rs)`	`Rd = sat_8(Rs);`
`Rd=sath(Rs)`	`Rd = sat_16(Rs);`
`Rd=satub(Rs)`	`Rd = usat_8(Rs);`
`Rd=satuh(Rs)`	`Rd = usat_16(Rs);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=sat(Rss)`	`Word32 Q6_R_sat_P(Word64 Rss)`
`Rd=satb(Rs)`	`Word32 Q6_R_satb_R(Word32 Rs)`
`Rd=sath(Rs)`	`Word32 Q6_R_sath_R(Word32 Rs)`
`Rd=satub(Rs)`	`Word32 Q6_R_satub_R(Word32 Rs)`
`Rd=satuh(Rs)`	`Word32 Q6_R_satuh_R(Word32 Rs)`

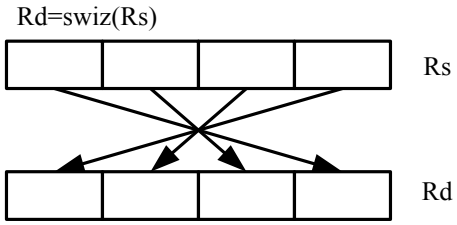
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sat(Rss)
1	0	0	0	1	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=sath(Rs)
1	0	0	0	1	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=satuh(Rs)
1	0	0	0	1	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=satub(Rs)
1	0	0	0	1	1	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=satb(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Swizzle bytes

Swizzle the bytes of a word. This instruction is useful in converting between little and big endian formats.



Syntax	Behavior

`Rd=swiz(Rs)`	`Rd.b[0]=Rs.b[3];`
	`Rd.b[1]=Rs.b[2];`
	`Rd.b[2]=Rs.b[1];`
	`Rd.b[3]=Rs.b[0];`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=swiz(Rs)`	`Word32 Q6_R_swiz_R(Word32 Rs)`

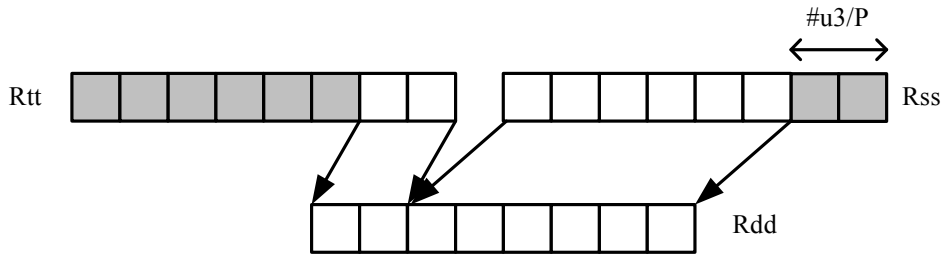
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	1	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=swiz(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Vector align

Align a vector. Use the immediate amount, or the least significant 3 bits of a Predicate register, as the number of bytes to align. Shift the Rss register pair right by this number of bytes. Fill the vacated positions with the least significant elements from Rtt.



Syntax	Behavior
`Rdd=valignb(Rtt,Rss,#u3)`	`Rdd = (Rss >>> #u*8)|(Rtt << ((8-#u)*8));`
`Rdd=valignb(Rtt,Rss,Pu)`	`Rdd = Rss >>> (Pu&0x7)*8|(Rtt << (8-(Pu&0x7))*8);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=valignb(Rtt,Rss,#u3)`	`Word64 Q6_P_valignb_PPI(Word64 Rtt, Word64 Rss, Word32 Iu3)`
`Rdd=valignb(Rtt,Rss,Pu)`	`Word64 Q6_P_valignb_PPp(Word64 Rtt, Word64 Rss, Byte Pu)`

Encoding

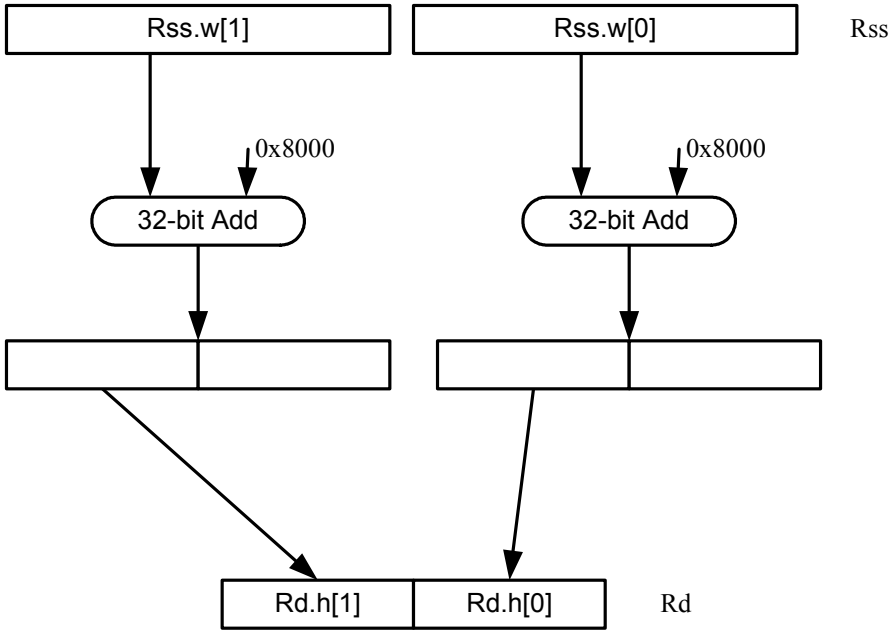
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	0	0	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	i	i	i	d	d	d	d	d	Rdd=valignb(Rtt,Rss,#u3)
ICLASS				RegType				Maj			s5					Parse			t5						u2		d5					
1	1	0	0	0	0	1	0	0	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	u	u	d	d	d	d	d	Rdd=valignb(Rtt,Rss,Pu)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u

Field name	Description
`Maj`	Major Opcode
`Min`	Minor Opcode
`RegType`	Register Type

Vector round and pack

Add the constant 0x00008000 to each word in the 64-bit source vector Rss. Optionally saturate this addition to 32bits. Pack the high halfwords of the result into the corresponding halfword of the 32-bit destination register.



Syntax	Behavior

`Rd=vrndwh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=(Rss.w[i]+0x08000).h[1];`
	`};`

`Rd=vrndwh(Rss):sat`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=sat_32(Rss.w[i]+0x08000).h[1];`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vrndwh(Rss)`	`Word32 Q6_R_vrndwh_P(Word64 Rss)`
`Rd=vrndwh(Rss):sat`	`Word32 Q6_R_vrndwh_P_sat(Word64 Rss)`

Encoding

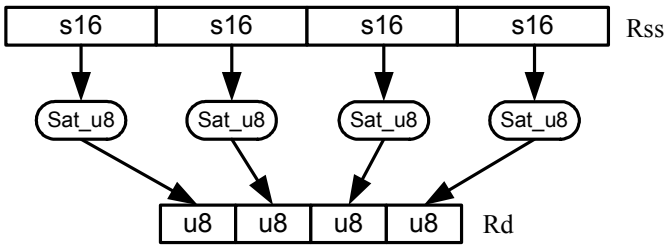
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**-**	**-**	**-**	**-**	**-**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=vrndwh(Rss)
1	0	0	0	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**-**	**-**	**-**	**-**	**-**	**-**	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=vrndwh(Rss):sat

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

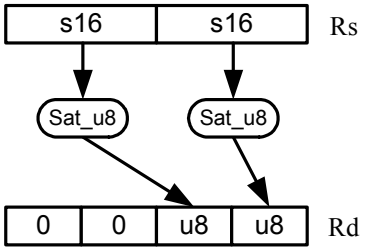
Vector saturate and pack

For each element in the vector, saturate the value to the next smaller size. VSATHUB saturates signed halfwords to unsigned bytes, while VSATHB saturates signed halfwords to signed bytes.

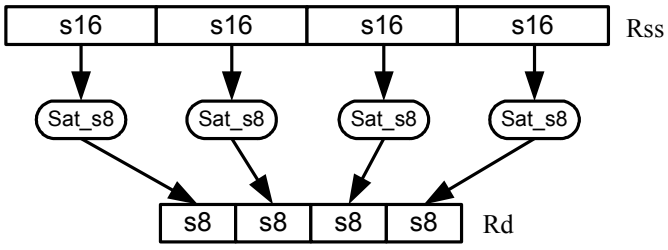
Rd=vsathub(Rss)



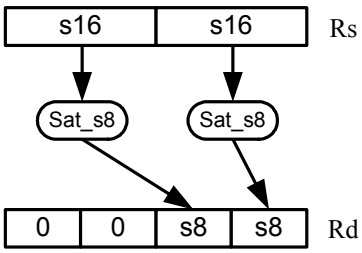
Rd=vsathub(Rs)



Rd=vsathb(Rss)

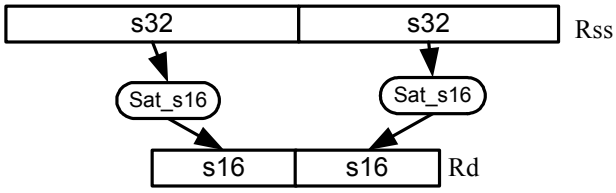


Rd=vsathb(Rs)

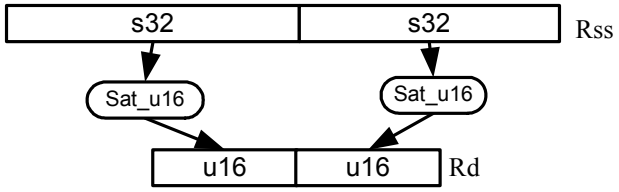


VSATWH saturates signed words to signed halfwords, while VSATWUH saturates signed words to unsigned halfwords. The resulting values are packed together into destination register Rd.

Rd=vsathwh(Rss)



Rd=vsathwuh(Rss)



Syntax	Behavior

`Rd=vsathb(Rs)`	`Rd.b[0]=sat_8(Rs.h[0]);`
	`Rd.b[1]=sat_8(Rs.h[1]);`
	`Rd.b[2]=0;`
	`Rd.b[3]=0;`

`Rd=vsathb(Rss)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=sat_8(Rss.h[i]);`
	`};`

`Rd=vsathub(Rs)`	`Rd.b[0]=usat_8(Rs.h[0]);`
	`Rd.b[1]=usat_8(Rs.h[1]);`
	`Rd.b[2]=0;`
	`Rd.b[3]=0;`

`Rd=vsathub(Rss)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=usat_8(Rss.h[i]);`
	`};`

`Rd=vsatwh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=sat_16(Rss.w[i]);`
	`};`

`Rd=vsatwuh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=usat_16(Rss.w[i]);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vsathb(Rs)`	`Word32 Q6_R_vsathb_R(Word32 Rs)`
`Rd=vsathb(Rss)`	`Word32 Q6_R_vsathb_P(Word64 Rss)`
`Rd=vsathub(Rs)`	`Word32 Q6_R_vsathub_R(Word32 Rs)`
`Rd=vsathub(Rss)`	`Word32 Q6_R_vsathub_P(Word64 Rss)`
`Rd=vsatwh(Rss)`	`Word32 Q6_R_vsatwh_P(Word64 Rss)`
`Rd=vsatwuh(Rss)`	`Word32 Q6_R_vsatwuh_P(Word64 Rss)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse							MinOp			d5					
1	0	0	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=vsathub(Rss)

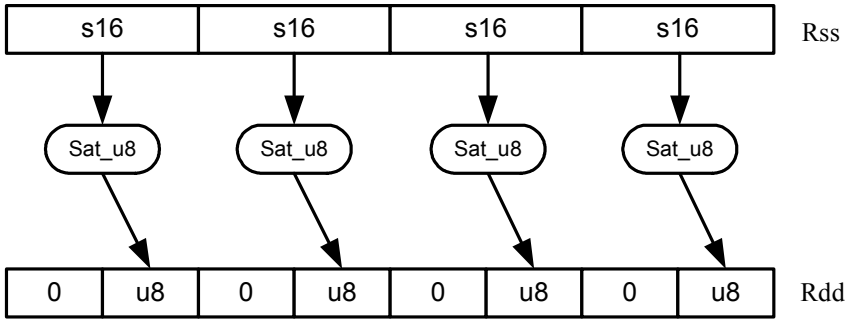
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=vsatwh(Rss)
1	0	0	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=vsatwuh(Rss)
1	0	0	0	1	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rd=vsathb(Rss)
1	0	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=vsathb(Rs)
1	0	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=vsathub(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

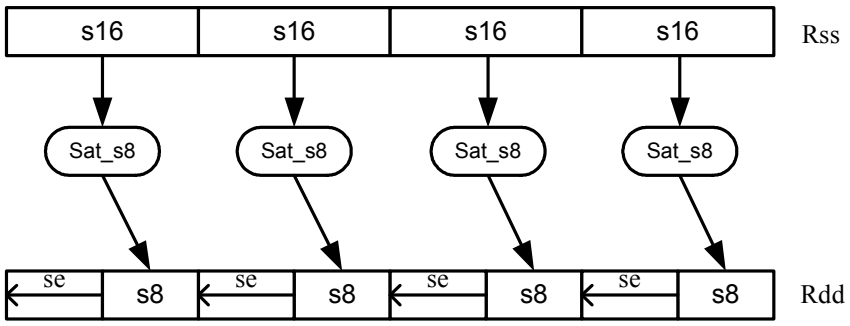
Vector saturate without pack

Saturate each element of source vector Rss to the next smaller size. VSATHUB saturates signed halfwords to unsigned bytes. VSATWH saturates signed words to signed halfwords, and VSATWUH saturates signed words to unsigned halfwords. The resulting values are placed in destination register Rdd in unpacked form.

Rdd=vsathub(Rss)



Rdd=vsathb(Rss)



Syntax	Behavior

`Rdd=vsathb(Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=sat_8(Rss.h[i]);`
	`};`

`Rdd=vsathub(Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=usat_8(Rss.h[i]);`
	`};`

`Rdd=vsatwh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=sat_16(Rss.w[i]);`
	`};`

`Rdd=vsatwuh(Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=usat_16(Rss.w[i]);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rdd=vsathb(Rss)`	`Word64 Q6_P_vsathb_P(Word64 Rss)`
`Rdd=vsathub(Rss)`	`Word64 Q6_P_vsathub_P(Word64 Rss)`
`Rdd=vsatwh(Rss)`	`Word64 Q6_P_vsatwh_P(Word64 Rss)`
`Rdd=vsatwuh(Rss)`	`Word64 Q6_P_vsatwuh_P(Word64 Rss)`

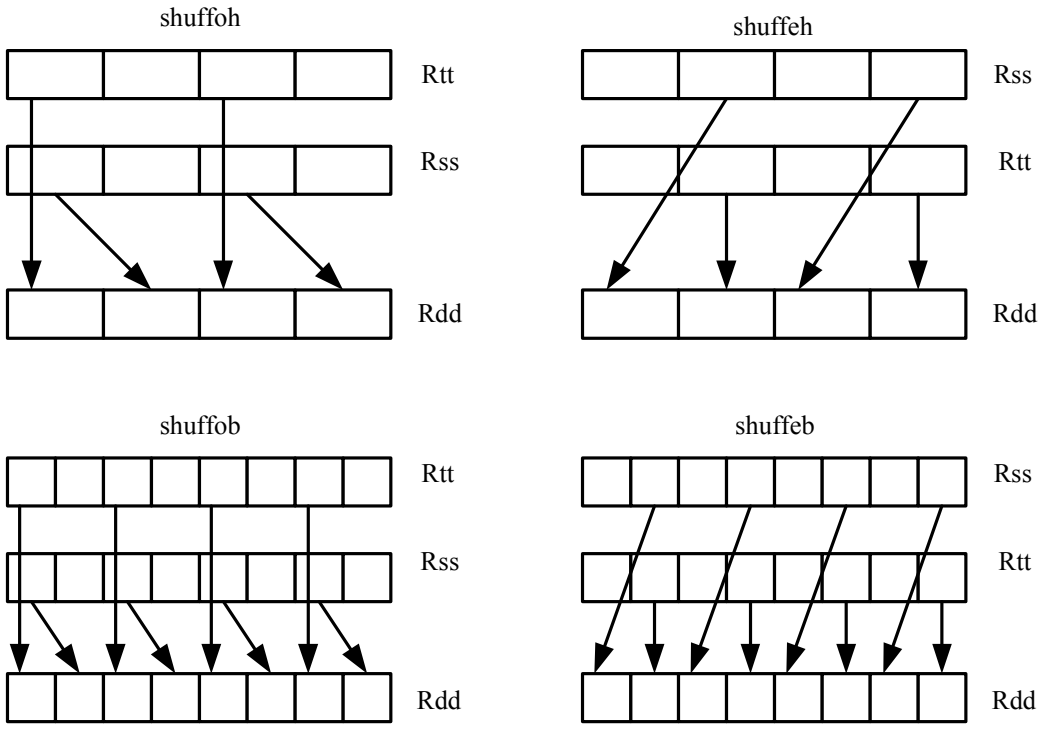
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsathub(Rss)
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsatwuh(Rss)
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vsatwh(Rss)
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vsathb(Rss)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Vector shuffle

Shuffle odd halfwords (shuffoh) takes the odd halfwords from Rtt and the odd halfwords from Rss and merges them together into vector Rdd. Shuffle even halfwords (shuffeh) performs the same operation on every even halfword in Rss and Rtt. The same operation is available for odd and even bytes.



Syntax	Behavior

`Rdd=shuffeb(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.b[i*2]=Rtt.b[i*2];`
	`  Rdd.b[i*2+1]=Rss.b[i*2];`
	`};`

`Rdd=shuffeh(Rss,Rtt)`	`for (i=0;i<2;i++) {`
	`  Rdd.h[i*2]=Rtt.h[i*2];`
	`  Rdd.h[i*2+1]=Rss.h[i*2];`
	`};`

`Rdd=shuffob(Rtt,Rss)`	`for (i=0;i<4;i++) {`
	`  Rdd.b[i*2]=Rss.b[i*2+1];`
	`  Rdd.b[i*2+1]=Rtt.b[i*2+1];`
	`};`

`Rdd=shuffoh(Rtt,Rss)`	`for (i=0;i<2;i++) {`
	`  Rdd.h[i*2]=Rss.h[i*2+1];`
	`  Rdd.h[i*2+1]=Rtt.h[i*2+1];`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=shuffeb(Rss,Rtt)`	`Word64 Q6_P_shuffeb_PP(Word64 Rss, Word64 Rtt)`
`Rdd=shuffeh(Rss,Rtt)`	`Word64 Q6_P_shuffeh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=shuffob(Rtt,Rss)`	`Word64 Q6_P_shuffob_PP(Word64 Rtt, Word64 Rss)`
`Rdd=shuffoh(Rtt,Rss)`	`Word64 Q6_P_shuffoh_PP(Word64 Rtt, Word64 Rss)`

Encoding

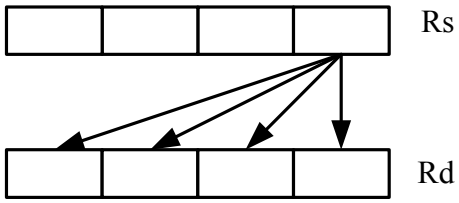
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=shuffeb(Rss,Rtt)
1	1	0	0	0	0	0	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=shuffob(Rtt,Rss)
1	1	0	0	0	0	0	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=shuffeh(Rss,Rtt)
1	1	0	0	0	0	0	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=shuffoh(Rtt,Rss)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major Opcode
`Min`	Minor Opcode
`RegType`	Register Type

Vector splat bytes

Replicate the low 8-bits from register Rs into each of the four bytes of destination register Rd.

Rd=vsplatb(Rs)



Syntax	Behavior

`Rd=vsplatb(Rs)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=Rs.b[0];`
	`};`

`Rdd=vsplatb(Rs)`	`for (i=0;i<8;i++) {`
	`  Rdd.b[i]=Rs.b[0];`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=vsplatb(Rs)`	`Word32 Q6_R_vsplatb_R(Word32 Rs)`
`Rdd=vsplatb(Rs)`	`Word64 Q6_P_vsplatb_R(Word32 Rs)`

Encoding

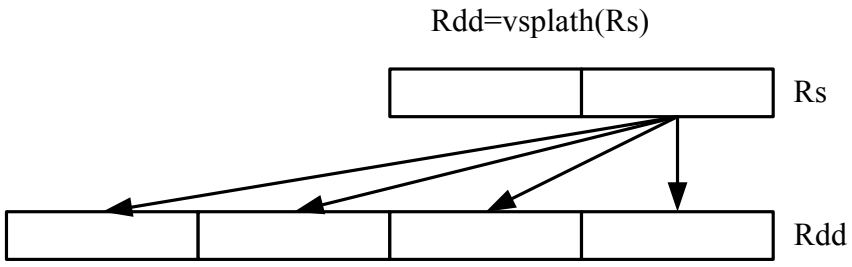
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vsplatb(Rs)
1	0	0	0	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	1	**d**	**d**	**d**	**d**	**d**	Rd=vsplatb(Rs)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

NOTE `Rdd=vsplatb(Rs)` is supported only in processor versions V61 or greater.

Vector splat halfwords

Replicate the low 16-bits from register Rs into each of the four halfwords of destination Rdd.



Syntax	Behavior

`Rdd=vsplath(Rs)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=Rs.h[0];`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vsplath(Rs)`	`Word64 Q6_P_vsplath_R(Word32 Rs)`

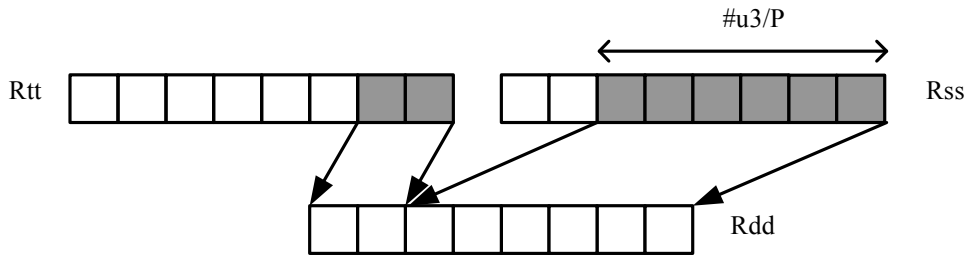
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse							MinOp			d5						
1	0	0	0	0	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vsplath(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Vector splice

Concatenate the low (8-N) bytes of vector Rtt with the low N bytes of vector Rss. This instruction is helpful to vectorize unaligned stores.



Syntax	Behavior
`Rdd=vspliceb(Rss,Rtt,#u3)`	`Rdd = Rtt << #u*8 | zxt`$_{\#u*8->64}$`(Rss);`
`Rdd=vspliceb(Rss,Rtt,Pu)`	`Rdd = Rtt << (Pu&7)*8 | zxt`$_{(Pu\&7)*8->64}$`(Rss);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vspliceb(Rss,Rtt,#u3)`	`Word64 Q6_P_vspliceb_PPI(Word64 Rss, Word64 Rtt, Word32 Iu3)`
`Rdd=vspliceb(Rss,Rtt,Pu)`	`Word64 Q6_P_vspliceb_PPp(Word64 Rss, Word64 Rtt, Byte Pu)`

Encoding

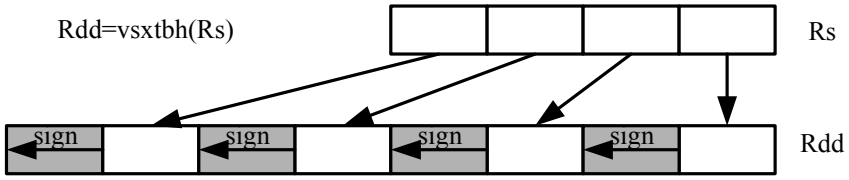
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	0	1	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	i	i	i	d	d	d	d	d	Rdd=vspliceb(Rss,Rtt,#u3)
ICLASS				RegType				Maj			s5					Parse			t5						u2		d5					
1	1	0	0	0	0	1	0	1	0	0	s	s	s	s	s	P	P	-	t	t	t	t	t	-	u	u	d	d	d	d	d	Rdd=vspliceb(Rss,Rtt,Pu)

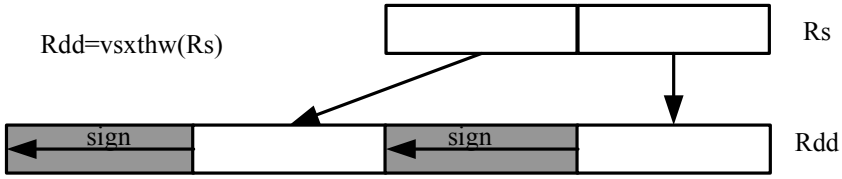
Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
u2	Field to encode register u
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Vector sign extend

vsxtbh sign-extends each byte of a single register source to halfwords, and places the result in the destination register pair.

vsxthw sign-extends each halfword of a single register source to words, and places the result in the destination register pair.





Syntax	Behavior

`Rdd=vsxtbh(Rs)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=Rs.b[i];`
	`};`

`Rdd=vsxthw(Rs)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=Rs.h[i];`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vsxtbh(Rs)`	`Word64 Q6_P_vsxtbh_R(Word32 Rs)`
`Rdd=vsxthw(Rs)`	`Word64 Q6_P_vsxthw_R(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vsxtbh(Rs)
1	0	0	0	0	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vsxthw(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

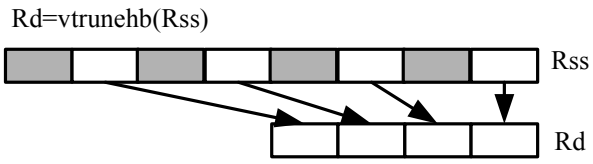
Vector truncate

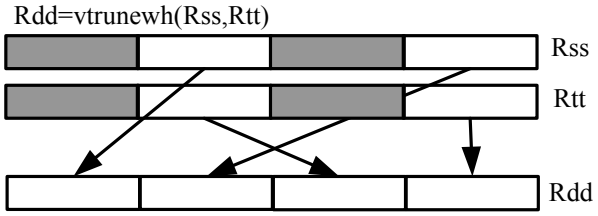
In vtrunehb, for each halfword in a vector, take the even (lower) byte and ignore the other byte. The resulting values are packed into destination register Rd.

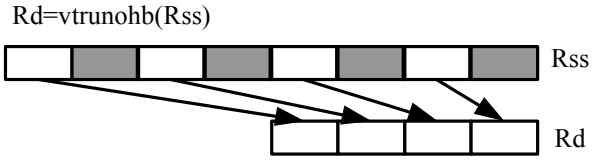
vtrunohb takes each odd byte of the source vector.

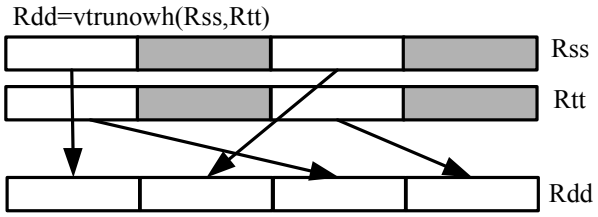
vtrunewh uses two source register pairs, Rss and Rtt. The even (lower) halfwords of Rss are packed in the upper word of Rdd, while the lower halfwords of Rtt are packed in the lower word of Rdd.

vtrunowh performs the same operation as vtrunewh, but uses the odd (upper) halfwords of the source vectors instead.









Syntax	Behavior

`Rd=vtrunehb(Rss)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=Rss.b[i*2];`
	`};`

`Rd=vtrunohb(Rss)`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=Rss.b[i*2+1];`
	`};`

`Rdd=vtrunehb(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.b[i]=Rtt.b[i*2];`
	`  Rdd.b[i+4]=Rss.b[i*2];`
	`};`

`Rdd=vtrunewh(Rss,Rtt)`	`Rdd.h[0]=Rtt.h[0];`
	`Rdd.h[1]=Rtt.h[2];`
	`Rdd.h[2]=Rss.h[0];`
	`Rdd.h[3]=Rss.h[2];`

`Rdd=vtrunohb(Rss,Rtt)`	`for (i=0;i<4;i++) {`
	`  Rdd.b[i]=Rtt.b[i*2+1];`
	`  Rdd.b[i+4]=Rss.b[i*2+1];`
	`};`

`Rdd=vtrunowh(Rss,Rtt)`	`Rdd.h[0]=Rtt.h[1];`
	`Rdd.h[1]=Rtt.h[3];`
	`Rdd.h[2]=Rss.h[1];`
	`Rdd.h[3]=Rss.h[3];`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=vtrunehb(Rss)`	`Word32 Q6_R_vtrunehb_P(Word64 Rss)`
`Rd=vtrunohb(Rss)`	`Word32 Q6_R_vtrunohb_P(Word64 Rss)`
`Rdd=vtrunehb(Rss,Rtt)`	`Word64 Q6_P_vtrunehb_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vtrunewh(Rss,Rtt)`	`Word64 Q6_P_vtrunewh_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vtrunohb(Rss,Rtt)`	`Word64 Q6_P_vtrunohb_PP(Word64 Rss, Word64 Rtt)`
`Rdd=vtrunowh(Rss,Rtt)`	`Word64 Q6_P_vtrunowh_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=vtrunohb(Rss)
1	0	0	0	1	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=vtrunehb(Rss)
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	0	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vtrunewh(Rss,Rtt)
1	1	0	0	0	0	0	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=vtrunehb(Rss,Rtt)
1	1	0	0	0	0	0	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vtrunowh(Rss,Rtt)
1	1	0	0	0	0	0	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vtrunohb(Rss,Rtt)

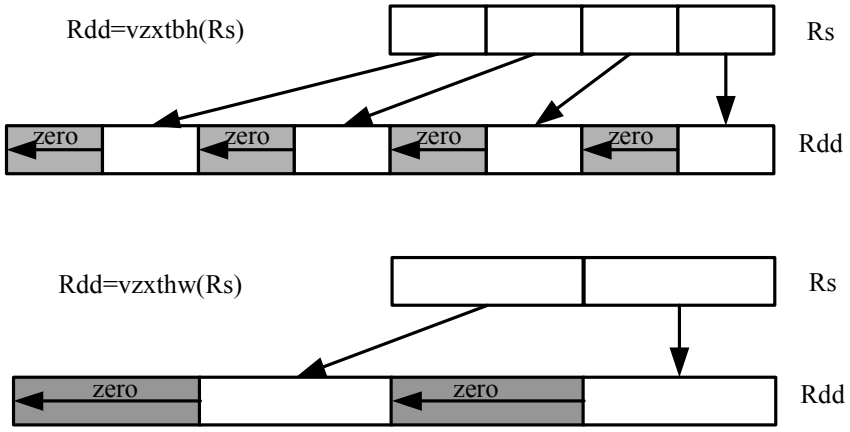
Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
MinOp	Minor Opcode
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type
RegType	Register Type

NOTE `Rdd=vtrunehb(Rss,Rtt)` and `Rdd=vtrunohb(Rss,Rtt)` are supported only in processor versions V61 or greater.

Vector zero extend

vzxtbh zero-extends each byte of a single register source to halfwords, and places the result in the destination register pair.

vzxthw zero-extends each halfword of a single register source to words, and places the result in the destination register pair.



Syntax	Behavior

`Rdd=vzxtbh(Rs)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=Rs.ub[i];`
	`};`

`Rdd=vzxthw(Rs)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=Rs.uh[i];`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vzxtbh(Rs)`	`Word64 Q6_P_vzxtbh_R(Word32 Rs)`
`Rdd=vzxthw(Rs)`	`Word64 Q6_P_vzxthw_R(Word32 Rs)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	0	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vzxtbh(Rs)
1	0	0	0	0	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vzxthw(Rs)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

11.12.7 XTYPE/PRED

The XTYPE/PRED instruction subclass includes instructions which perform miscellaneous operations on predicates, including mask generation, predicate transfers, and the Viterbi pack operation.

Bounds check

Determine if Rs falls in the range defined by Rtt.

Rtt.w0 is set by the user to the lower bound, and Rtt.w1 is set by the user to the upper bound.

All bits of the destination predicate are set if the value falls within the range, or all cleared otherwise.

Syntax	Behavior
`Pd=boundscheck(Rs,Rtt)`	`if ("Rs & 1") { Assembler mapped to: "Pd=boundscheck(Rss,Rtt):raw:hi"; } else { Assembler mapped to: "Pd=boundscheck(Rss,Rtt):raw:lo"; };`
`Pd=boundscheck(Rss,Rtt):raw:hi`	`src = Rss.uw[1]; Pd = (src.uw[0] >= Rtt.uw[0]) && (src.uw[0] < Rtt.uw[1]) ? 0xff : 0x00;`
`Pd=boundscheck(Rss,Rtt):raw:lo`	`src = Rss.uw[0]; Pd = (src.uw[0] >= Rtt.uw[0]) && (src.uw[0] < Rtt.uw[1]) ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=boundscheck(Rs,Rtt)`	`Byte Q6_p_boundscheck_RP(Word32 Rs, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5						MinOp						d2		
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	0	0	-	-	-	**d**	**d**	Pd=boundscheck(Rss,Rtt):raw:lo
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	1	0	1	-	-	-	**d**	**d**	Pd=boundscheck(Rss,Rtt):raw:hi

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Compare byte

These instructions sign- or zero-extend the low 8-bits of the source registers and perform 32-bit comparisons on the result. In the case of an extended 32-bit immediate operand, the full 32 immediate bits are used for the comparison.

Syntax	Behavior
`Pd=cmpb.eq(Rs,#u8)`	`Pd=Rs.ub[0] == #u ? 0xff : 0x00;`
`Pd=cmpb.eq(Rs,Rt)`	`Pd=Rs.b[0] == Rt.b[0] ? 0xff : 0x00;`
`Pd=cmpb.gt(Rs,#s8)`	`Pd=Rs.b[0] > #s ? 0xff : 0x00;`
`Pd=cmpb.gt(Rs,Rt)`	`Pd=Rs.b[0] > Rt.b[0] ? 0xff : 0x00;`
`Pd=cmpb.gtu(Rs,#u7)`	`apply_extension(#u);` `Pd=Rs.ub[0] > #u.uw[0] ? 0xff : 0x00;`
`Pd=cmpb.gtu(Rs,Rt)`	`Pd=Rs.ub[0] > Rt.ub[0] ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=cmpb.eq(Rs,#u8)`	`Byte Q6_p_cmpb_eq_RI(Word32 Rs, Word32 Iu8)`
`Pd=cmpb.eq(Rs,Rt)`	`Byte Q6_p_cmpb_eq_RR(Word32 Rs, Word32 Rt)`
`Pd=cmpb.gt(Rs,#s8)`	`Byte Q6_p_cmpb_gt_RI(Word32 Rs, Word32 Is8)`
`Pd=cmpb.gt(Rs,Rt)`	`Byte Q6_p_cmpb_gt_RR(Word32 Rs, Word32 Rt)`
`Pd=cmpb.gtu(Rs,#u7)`	`Byte Q6_p_cmpb_gtu_RI(Word32 Rs, Word32 Iu7)`
`Pd=cmpb.gtu(Rs,Rt)`	`Byte Q6_p_cmpb_gtu_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min						d2		
1	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	-	-	-	**d**	**d**	Pd=cmpb.gt(Rs,Rt)
1	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	0	-	-	-	**d**	**d**	Pd=cmpb.eq(Rs,Rt)
1	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	1	-	-	-	**d**	**d**	Pd=cmpb.gtu(Rs,Rt)
ICLASS				RegType							s5					Parse														d2		
1	1	0	1	1	1	0	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	-	**d**	**d**	Pd=cmpb.eq(Rs,#u8)
1	1	0	1	1	1	0	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	-	**d**	**d**	Pd=cmpb.gt(Rs,#s8)
1	1	0	1	1	1	0	1	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	-	**d**	**d**	Pd=cmpb.gtu(Rs,#u7)

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d

Field name	Description
s5	Field to encode register s
t5	Field to encode register t
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Compare half

These instructions sign- or zero-extend the low 16-bits of the source registers and perform 32-bit comparisons on the result. In the case of an extended 32-bit immediate operand, the full 32 immediate bits are used for the comparison.

Syntax	Behavior
`Pd=cmph.eq(Rs,#s8)`	`apply_extension(#s);` `Pd=Rs.h[0] == #s ? 0xff : 0x00;`
`Pd=cmph.eq(Rs,Rt)`	`Pd=Rs.h[0] == Rt.h[0] ? 0xff : 0x00;`
`Pd=cmph.gt(Rs,#s8)`	`apply_extension(#s);` `Pd=Rs.h[0] > #s ? 0xff : 0x00;`
`Pd=cmph.gt(Rs,Rt)`	`Pd=Rs.h[0] > Rt.h[0] ? 0xff : 0x00;`
`Pd=cmph.gtu(Rs,#u7)`	`apply_extension(#u);` `Pd=Rs.uh[0] > #u.uw[0] ? 0xff : 0x00;`
`Pd=cmph.gtu(Rs,Rt)`	`Pd=Rs.uh[0] > Rt.uh[0] ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=cmph.eq(Rs,#s8)`	`Byte Q6_p_cmph_eq_RI(Word32 Rs, Word32 Is8)`
`Pd=cmph.eq(Rs,Rt)`	`Byte Q6_p_cmph_eq_RR(Word32 Rs, Word32 Rt)`
`Pd=cmph.gt(Rs,#s8)`	`Byte Q6_p_cmph_gt_RI(Word32 Rs, Word32 Is8)`
`Pd=cmph.gt(Rs,Rt)`	`Byte Q6_p_cmph_gt_RR(Word32 Rs, Word32 Rt)`
`Pd=cmph.gtu(Rs,#u7)`	`Byte Q6_p_cmph_gtu_RI(Word32 Rs, Word32 Iu7)`
`Pd=cmph.gtu(Rs,Rt)`	`Byte Q6_p_cmph_gtu_RR(Word32 Rs, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min						d2		
1	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	1	-	-	-	**d**	**d**	Pd=cmph.eq(Rs,Rt)
1	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	-	-	-	**d**	**d**	Pd=cmph.gt(Rs,Rt)
1	1	0	0	0	1	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	1	-	-	-	**d**	**d**	Pd=cmph.gtu(Rs,Rt)
ICLASS				RegType							s5					Parse														d2		
1	1	0	1	1	1	0	1	-	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	-	**d**	**d**	Pd=cmph.eq(Rs,#s8)
1	1	0	1	1	1	0	1	-	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	-	**d**	**d**	Pd=cmph.gt(Rs,#s8)
1	1	0	1	1	1	0	1	-	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	-	**d**	**d**	Pd=cmph.gtu(Rs,#u7)

Field name	Description
`RegType`	Register Type
`MajOp`	Major Opcode
`ICLASS`	Instruction Class

Field name	Description
Parse	Packet/Loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Compare doublewords

Compare two 64-bit register pairs for unsigned greater than, greater than, or equal. The 8-bit predicate register Pd is set to all 1's or all 0's depending on the result.

Syntax	Behavior
`Pd=cmp.eq(Rss,Rtt)`	`Pd=Rss==Rtt ? 0xff : 0x00;`
`Pd=cmp.gt(Rss,Rtt)`	`Pd=Rss>Rtt ? 0xff : 0x00;`
`Pd=cmp.gtu(Rss,Rtt)`	`Pd=Rss.u64>Rtt.u64 ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=cmp.eq(Rss,Rtt)`	`Byte Q6_p_cmp_eq_PP(Word64 Rss, Word64 Rtt)`
`Pd=cmp.gt(Rss,Rtt)`	`Byte Q6_p_cmp_gt_PP(Word64 Rss, Word64 Rtt)`
`Pd=cmp.gtu(Rss,Rtt)`	`Byte Q6_p_cmp_gtu_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp						d2		
1	1	0	1	0	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	0	-	-	-	**d**	**d**	Pd=cmp.eq(Rss,Rtt)
1	1	0	1	0	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	-	-	-	**d**	**d**	Pd=cmp.gt(Rss,Rtt)
1	1	0	1	0	0	1	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	0	-	-	-	**d**	**d**	Pd=cmp.gtu(Rss,Rtt)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Compare bit mask

If all the bits in the mask in Rt or a short immediate are set (BITSSET) or clear (BITSCLEAR) in Rs, set the Pd to true. Otherwise, set the bits in Pd to false.

Syntax	Behavior
`Pd=[!]bitsclr(Rs,#u6)`	`Pd=(Rs&#u)[!]=0 ? 0xff : 0x00;`
`Pd=[!]bitsclr(Rs,Rt)`	`Pd=(Rs&Rt)[!]=0 ? 0xff : 0x00;`
`Pd=[!]bitsset(Rs,Rt)`	`Pd=(Rs&Rt)[!]=Rt ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=!bitsclr(Rs,#u6)`	`Byte Q6_p_not_bitsclr_RI(Word32 Rs, Word32 Iu6)`
`Pd=!bitsclr(Rs,Rt)`	`Byte Q6_p_not_bitsclr_RR(Word32 Rs, Word32 Rt)`
`Pd=!bitsset(Rs,Rt)`	`Byte Q6_p_not_bitsset_RR(Word32 Rs, Word32 Rt)`
`Pd=bitsclr(Rs,#u6)`	`Byte Q6_p_bitsclr_RI(Word32 Rs, Word32 Iu6)`
`Pd=bitsclr(Rs,Rt)`	`Byte Q6_p_bitsclr_RR(Word32 Rs, Word32 Rt)`
`Pd=bitsset(Rs,Rt)`	`Byte Q6_p_bitsset_RR(Word32 Rs, Word32 Rt)`

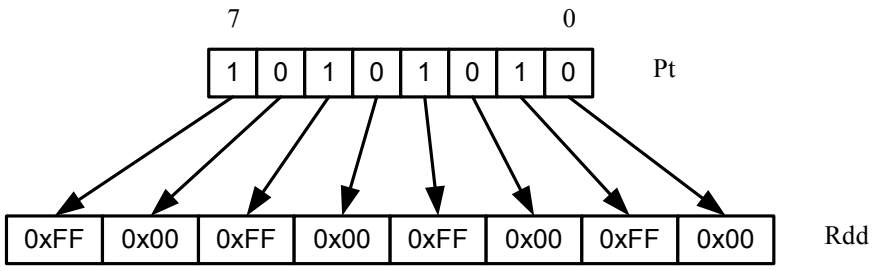
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse														d2		
1	0	0	0	0	1	0	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	-	-	-	-	-	-	**d**	**d**	Pd=bitsclr(Rs,#u6)
1	0	0	0	0	1	0	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	-	-	-	-	-	-	**d**	**d**	Pd=!bitsclr(Rs,#u6)
ICLASS				RegType				Maj			s5					Parse			t5										d2		
1	1	0	0	0	1	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=bitsset(Rs,Rt)
1	1	0	0	0	1	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=!bitsset(Rs,Rt)
1	1	0	0	0	1	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=bitsclr(Rs,Rt)
1	1	0	0	0	1	1	1	1	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=!bitsclr(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
Maj	Major Opcode
RegType	Register Type
RegType	Register Type

Mask generate from predicate

For each of the low 8 bits in predicate register Pt, if the bit is set then set the corresponding byte in 64-bit register pair Rdd to 0xff, otherwise, set the corresponding byte to 0x00.



Syntax	Behavior

`Rdd=mask(Pt)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=(Pt.i?(0xff):(0x00));`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=mask(Pt)`	`Word64 Q6_P_mask_p(Byte Pt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType												Parse						t2					d5					
1	0	0	0	0	1	1	0	-	-	-	-	-	-	-	-	**P**	**P**	-	-	-	-	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rdd=mask(Pt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
t2	Field to encode register t
RegType	Register Type

Check for TLB match

Determine if the TLB entry in Rss matches the ASID:PPN in Rt.

Syntax	Behavior

`Pd=tlbmatch(Rss,Rt)`	`MASK = 0x07ffffff;`
	`TLBLO = Rss.uw[0];`
	`TLBHI = Rss.uw[1];`
	`SIZE =`
	`min(6,count_leading_ones(~reverse_bits(TLBLO)));`
	`MASK &= (0xffffffff << 2*SIZE);`
	`Pd = TLBHI.31 && ((TLBHI & MASK) == (Rt & MASK))`
	`? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Notes

- The predicate generated by this instruction can not be used as a .new predicate, nor can it be automatically ANDed with another predicate.

Intrinsics

`Pd=tlbmatch(Rss,Rt)`	`Byte Q6_p_tlbmatch_PR(Word64 Rss, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp						d2		
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	1	1	-	-	-	**d**	**d**	Pd=tlbmatch(Rss,Rt)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Predicate transfer

Pd=Rs transfers a predicate to the 8 least-significant bits of a general register and zeros the other bits.

Rd=Ps transfers the 8 least-significant bits of a general register to a predicate.

Syntax	Behavior
`Pd=Rs`	`Pd = Rs.ub[0];`
`Rd=Ps`	`Rd = zxt`$_{8->32}$`(Ps);`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=Rs`	`Byte Q6_p_equals_R(Word32 Rs)`
`Rd=Ps`	`Word32 Q6_R_equals_p(Byte Ps)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse														d2		
1	0	0	0	0	1	0	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	-	-	-	**d**	**d**	Pd=Rs
ICLASS				RegType				MajOp						s2		Parse											d5					
1	0	0	0	1	0	0	1	-	1	-	-	-	-	**s**	**s**	**P**	**P**	-	-	-	-	-	-	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=Ps

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`d5`	Field to encode register d
`s2`	Field to encode register s
`s5`	Field to encode register s
`MajOp`	Major Opcode
`RegType`	Register Type

Test bit

Extract a bit from a register. If the bit is true (1), set all the bits of the predicate register destination to 1. If the bit is false (0), set all the bits of the predicate register destination to 0. The bit to be tested can be indicated using an immediate or register value.

If a register is used to indicate the bit to test, and the value specified is out of range, the predicate result is zero.

Syntax	Behavior
`Pd=[!]tstbit(Rs,#u5)`	`Pd = (Rs & (1<<#u)) == 0 ? 0xff : 0x00;`
`Pd=[!]tstbit(Rs,Rt)`	`Pd = (zxt32->64(Rs) & (sxt7->32(Rt)>0)?(zxt32->64(1)<<sxt7->32(Rt)):(zxt32->64(1)>>>sxt7->32(Rt))) == 0 ? 0xff : 0x00;`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=!tstbit(Rs,#u5)`	`Byte Q6_p_not_tstbit_RI(Word32 Rs, Word32 Iu5)`
`Pd=!tstbit(Rs,Rt)`	`Byte Q6_p_not_tstbit_RR(Word32 Rs, Word32 Rt)`
`Pd=tstbit(Rs,#u5)`	`Byte Q6_p_tstbit_RI(Word32 Rs, Word32 Iu5)`
`Pd=tstbit(Rs,Rt)`	`Byte Q6_p_tstbit_RR(Word32 Rs, Word32 Rt)`

Encoding

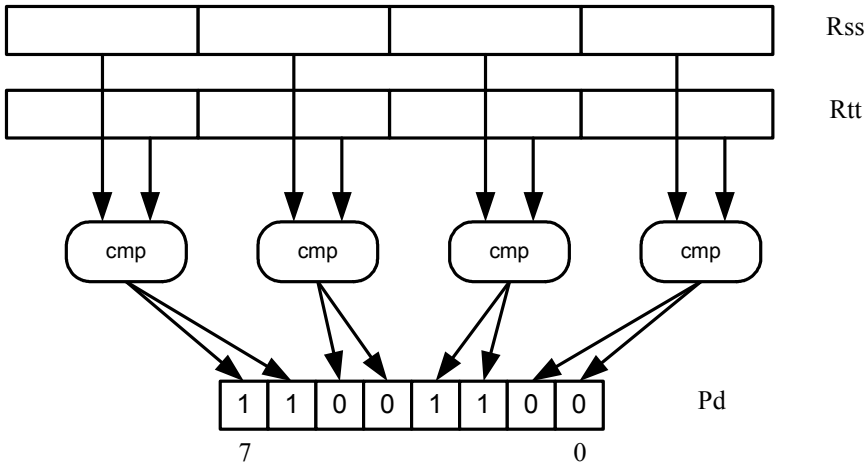
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse														d2		
1	0	0	0	0	1	0	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	-	-	-	-	-	-	**d**	**d**	Pd=tstbit(Rs,#u5)
1	0	0	0	0	1	0	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	-	-	-	-	-	-	**d**	**d**	Pd=!tstbit(Rs,#u5)
ICLASS				RegType				Maj			s5					Parse			t5										d2		
1	1	0	0	0	1	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=tstbit(Rs,Rt)
1	1	0	0	0	1	1	1	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	-	-	-	-	-	-	**d**	**d**	Pd=!tstbit(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
Maj	Major Opcode
RegType	Register Type
RegType	Register Type

Vector compare halfwords

Compare each of four 16-bit halfwords in two 64-bit vectors and set the corresponding bits in a predicate destination to '11' if true, '00' if false.

Halfword comparisons can be for equal, signed greater than, or unsigned greater than.



Syntax	Behavior

`Pd=vcmph.eq(Rss,#s8)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.h[i] == #s);`
	`  Pd.i*2+1 = (Rss.h[i] == #s);`
	`};`

`Pd=vcmph.eq(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.h[i] == Rtt.h[i]);`
	`  Pd.i*2+1 = (Rss.h[i] == Rtt.h[i]);`
	`};`

`Pd=vcmph.gt(Rss,#s8)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.h[i] > #s);`
	`  Pd.i*2+1 = (Rss.h[i] > #s);`
	`};`

`Pd=vcmph.gt(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.h[i] > Rtt.h[i]);`
	`  Pd.i*2+1 = (Rss.h[i] > Rtt.h[i]);`
	`};`

`Pd=vcmph.gtu(Rss,#u7)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.uh[i] > #u);`
	`  Pd.i*2+1 = (Rss.uh[i] > #u);`
	`};`

`Pd=vcmph.gtu(Rss,Rtt)`	`for (i = 0; i < 4; i++) {`
	`  Pd.i*2 = (Rss.uh[i] > Rtt.uh[i]);`
	`  Pd.i*2+1 = (Rss.uh[i] > Rtt.uh[i]);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

Pd=vcmph.eq(Rss,#s8)	Byte Q6_p_vcmph_eq_PI(Word64 Rss, Word32 Is8)
Pd=vcmph.eq(Rss,Rtt)	Byte Q6_p_vcmph_eq_PP(Word64 Rss, Word64 Rtt)
Pd=vcmph.gt(Rss,#s8)	Byte Q6_p_vcmph_gt_PI(Word64 Rss, Word32 Is8)
Pd=vcmph.gt(Rss,Rtt)	Byte Q6_p_vcmph_gt_PP(Word64 Rss, Word64 Rtt)
Pd=vcmph.gtu(Rss,#u7)	Byte Q6_p_vcmph_gtu_PI(Word64 Rss, Word32 Iu7)
Pd=vcmph.gtu(Rss,Rtt)	Byte Q6_p_vcmph_gtu_PP(Word64 Rss, Word64 Rtt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5						MinOp						d2		
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	1	-	-	-	**d**	**d**	Pd=vcmph.eq(Rss,Rtt)
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	0	-	-	-	**d**	**d**	Pd=vcmph.gt(Rss,Rtt)
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	0	1	-	-	-	**d**	**d**	Pd=vcmph.gtu(Rss,Rtt)
ICLASS				RegType							s5					Parse														d2		
1	1	0	1	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	-	**d**	**d**	Pd=vcmph.eq(Rss,#s8)
1	1	0	1	1	1	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	-	**d**	**d**	Pd=vcmph.gt(Rss,#s8)
1	1	0	1	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	-	**d**	**d**	Pd=vcmph.gtu(Rss,#u7)

Field name	Description
RegType	Register Type
MajOp	Major Opcode
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector compare bytes for any match

Compare each byte in two 64-bit source vectors and set a predicate if any of the 8 bytes are equal.

This instruction can be used to quickly find the null terminator in a string.

Syntax	Behavior

`Pd=any8(vcmpb.eq(Rss,Rtt))`	`Pd = 0;`
	`for (i = 0; i < 8; i++) {`
	`  if (Rss.b[i] == Rtt.b[i]) Pd = 0xff;`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=any8(vcmpb.eq(Rss,Rtt))`	`Byte Q6_p_any8_vcmpb_eq_PP(Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp						d2		
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	0	0	-	-	-	**d**	**d**	Pd=any8(vcmpb.eq(Rss,Rtt))

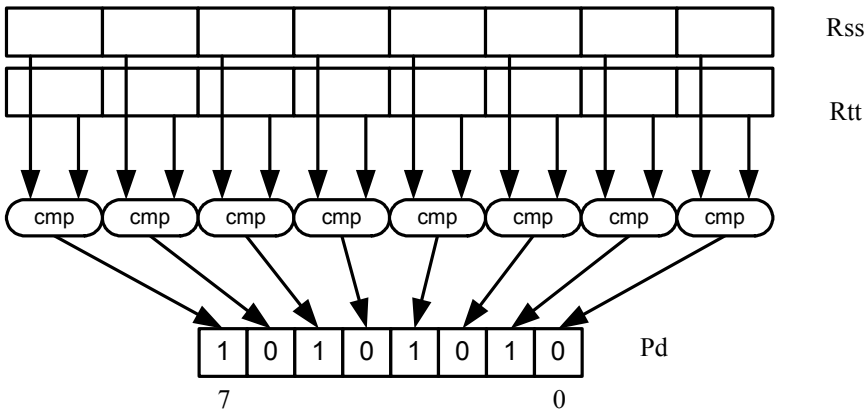
Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d2`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t

Vector compare bytes

Compare each of eight bytes in two 64-bit vectors and set the corresponding bit in a predicate destination to 1 if true, 0 if false.

Byte comparisons can be for equal or for unsigned greater than.

In the following example, every other comparison is true.



Syntax	Behavior

`Pd=vcmpb.eq(Rss,#u8)`	`for (i = 0; i < 8; i++) {`
	`  Pd.i = (Rss.ub[i] == #u);`
	`};`

`Pd=vcmpb.eq(Rss,Rtt)`	`for (i = 0; i < 8; i++) {`
	`  Pd.i = (Rss.b[i] == Rtt.b[i]);`
	`};`

`Pd=vcmpb.gt(Rss,#s8)`	`for (i = 0; i < 8; i++) {`
	`  Pd.i = (Rss.b[i] > #s);`
	`};`

`Pd=vcmpb.gt(Rss,Rtt)`	`for (i = 0; i < 8; i++) {`
	`  Pd.i = (Rss.b[i] > Rtt.b[i]);`
	`};`

`Pd=vcmpb.gtu(Rss,#u7)`	`for (i = 0; i < 8; i++) {`
	`  Pd.i = (Rss.ub[i] > #u);`
	`};`

`Pd=vcmpb.gtu(Rss,Rtt)`	`for (i = 0; i < 8; i++) {`
	`  Pd.i = (Rss.ub[i] > Rtt.ub[i]);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

Pd=vcmpb.eq(Rss,#u8)	Byte Q6_p_vcmpb_eq_PI(Word64 Rss, Word32 Iu8)
Pd=vcmpb.eq(Rss,Rtt)	Byte Q6_p_vcmpb_eq_PP(Word64 Rss, Word64 Rtt)
Pd=vcmpb.gt(Rss,#s8)	Byte Q6_p_vcmpb_gt_PI(Word64 Rss, Word32 Is8)
Pd=vcmpb.gt(Rss,Rtt)	Byte Q6_p_vcmpb_gt_PP(Word64 Rss, Word64 Rtt)
Pd=vcmpb.gtu(Rss,#u7)	Byte Q6_p_vcmpb_gtu_PI(Word64 Rss, Word32 Iu7)
Pd=vcmpb.gtu(Rss,Rtt)	Byte Q6_p_vcmpb_gtu_PP(Word64 Rss, Word64 Rtt)

Encoding

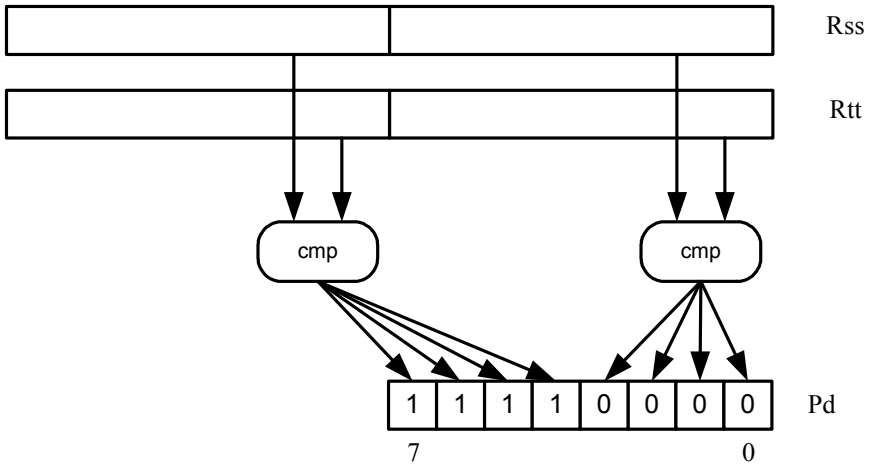
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5					MinOp						d2		
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	0	-	-	-	**d**	**d**	Pd=vcmpb.eq(Rss,Rtt)
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	1	1	1	-	-	-	**d**	**d**	Pd=vcmpb.gtu(Rss,Rtt)
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	1	**t**	**t**	**t**	**t**	**t**	0	1	0	-	-	-	**d**	**d**	Pd=vcmpb.gt(Rss,Rtt)
ICLASS				RegType							s5					Parse														d2		
1	1	0	1	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	-	**d**	**d**	Pd=vcmpb.eq(Rss,#u8)
1	1	0	1	1	1	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	-	**d**	**d**	Pd=vcmpb.gt(Rss,#s8)
1	1	0	1	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	-	**d**	**d**	Pd=vcmpb.gtu(Rss,#u7)

Field name	Description
RegType	Register Type
MajOp	Major Opcode
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Vector compare words

Compare each of two 32-bit words in two 64-bit vectors and set the corresponding bits in a predicate destination to '1111' if true, '0000' if false.

Word comparisons can be for equal, signed greater than, or unsigned greater than.



Syntax	Behavior
`Pd=vcmpw.eq(Rss,#s8)`	`Pd[3:0] = (Rss.w[0]==#s);` `Pd[7:4] = (Rss.w[1]==#s);`
`Pd=vcmpw.eq(Rss,Rtt)`	`Pd[3:0] = (Rss.w[0]==Rtt.w[0]);` `Pd[7:4] = (Rss.w[1]==Rtt.w[1]);`
`Pd=vcmpw.gt(Rss,#s8)`	`Pd[3:0] = (Rss.w[0]>#s);` `Pd[7:4] = (Rss.w[1]>#s);`
`Pd=vcmpw.gt(Rss,Rtt)`	`Pd[3:0] = (Rss.w[0]>Rtt.w[0]);` `Pd[7:4] = (Rss.w[1]>Rtt.w[1]);`
`Pd=vcmpw.gtu(Rss,#u7)`	`Pd[3:0] = (Rss.uw[0]>#u);` `Pd[7:4] = (Rss.uw[1]>#u);`
`Pd=vcmpw.gtu(Rss,Rtt)`	`Pd[3:0] = (Rss.uw[0]>Rtt.uw[0]);` `Pd[7:4] = (Rss.uw[1]>Rtt.uw[1]);`

Class: XTYPE (slots 2,3)

Intrinsics

`Pd=vcmpw.eq(Rss,#s8)`	`Byte Q6_p_vcmpw_eq_PI(Word64 Rss, Word32 Is8)`
`Pd=vcmpw.eq(Rss,Rtt)`	`Byte Q6_p_vcmpw_eq_PP(Word64 Rss, Word64 Rtt)`
`Pd=vcmpw.gt(Rss,#s8)`	`Byte Q6_p_vcmpw_gt_PI(Word64 Rss, Word32 Is8)`
`Pd=vcmpw.gt(Rss,Rtt)`	`Byte Q6_p_vcmpw_gt_PP(Word64 Rss, Word64 Rtt)`
`Pd=vcmpw.gtu(Rss,#u7)`	`Byte Q6_p_vcmpw_gtu_PI(Word64 Rss, Word32 Iu7)`
`Pd=vcmpw.gtu(Rss,Rtt)`	`Byte Q6_p_vcmpw_gtu_PP(Word64 Rss, Word64 Rtt)`

Encoding

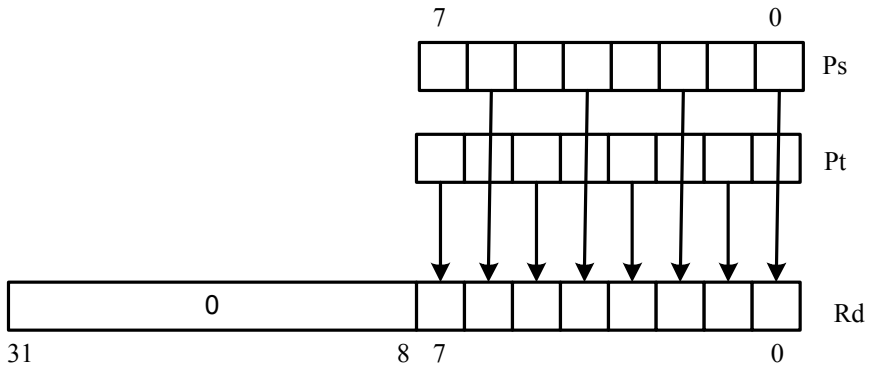
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5						MinOp					d2		
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	0	-	-	-	**d**	**d**	Pd=vcmpw.eq(Rss,Rtt)
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	0	1	-	-	-	**d**	**d**	Pd=vcmpw.gt(Rss,Rtt)
1	1	0	1	0	0	1	0	0	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	0	1	0	-	-	-	**d**	**d**	Pd=vcmpw.gtu(Rss,Rtt)
ICLASS				RegType							s5					Parse														d2		
1	1	0	1	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	1	0	-	**d**	**d**	Pd=vcmpw.eq(Rss,#s8)
1	1	0	1	1	1	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**i**	**i**	**i**	**i**	**i**	**i**	**i**	**i**	1	0	-	**d**	**d**	Pd=vcmpw.gt(Rss,#s8)
1	1	0	1	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	0	**i**	**i**	**i**	**i**	**i**	**i**	**i**	1	0	-	**d**	**d**	Pd=vcmpw.gtu(Rss,#u7)

Field name	Description
RegType	Register Type
MajOp	Major Opcode
MinOp	Minor Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d2	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Viterbi pack even and odd predicate bits

Pack the even and odd bits of two predicate registers into a single destination register. A variant of this instruction is the R3:2 |= vitpack(P1,P0). This places the packed predicate bits into the lower 8bits of the register pair which has been preshifted by 8bits.

This instruction is useful in Viterbi decoding. Repeated use of the push version enables a history to be stored for traceback, purposes.



Syntax	Behavior
`Rd=vitpack(Ps,Pt)`	`Rd = (Ps&0x55) | (Pt&0xAA);`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=vitpack(Ps,Pt)`	`Word32 Q6_R_vitpack_pp(Byte Ps, Byte Pt)`

Encoding

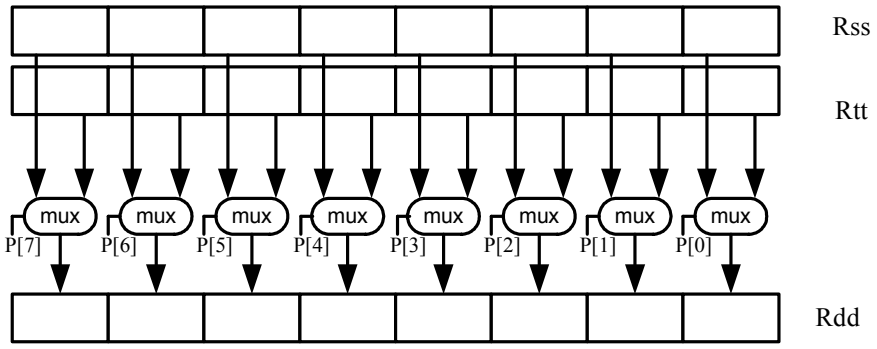
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp						s2		Parse						t2					d5					
1	0	0	0	1	0	0	1	-	0	0	-	-	-	**s**	**s**	**P**	**P**	-	-	-	-	**t**	**t**	-	-	-	**d**	**d**	**d**	**d**	**d**	Rd=vitpack(Ps,Pt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s2	Field to encode register s
t2	Field to encode register t
MajOp	Major Opcode
RegType	Register Type

Vector mux

Perform an element-wise byte selection between two vectors.

For each of the low 8 bits of predicate register Pu, if the bit is set, then the corresponding byte in Rdd is set to the corresponding byte from Rss. Otherwise, set the byte in Rdd to the byte from Rtt.



Syntax	Behavior

`Rdd=vmux(Pu,Rss,Rtt)`	`for (i = 0; i < 8; i++) {`
	`  Rdd.b[i]=(Pu.i?(Rss.b[i]):(Rtt.b[i]));`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vmux(Pu,Rss,Rtt)`	`Word64 Q6_P_vmux_pPP(Byte Pu, Word64 Rss, Word64 Rtt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType							s5					Parse			t5						u2		d5						
1	1	0	1	0	0	0	1	-	-	-	s	s	s	s	s	P	P	-	t	t	t	t	t	-	u	u	d	d	d	d	d	Rdd=vmux(Pu,Rss,Rtt)

Field name	Description
`RegType`	Register Type
`MinOp`	Minor Opcode
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`u2`	Field to encode register u

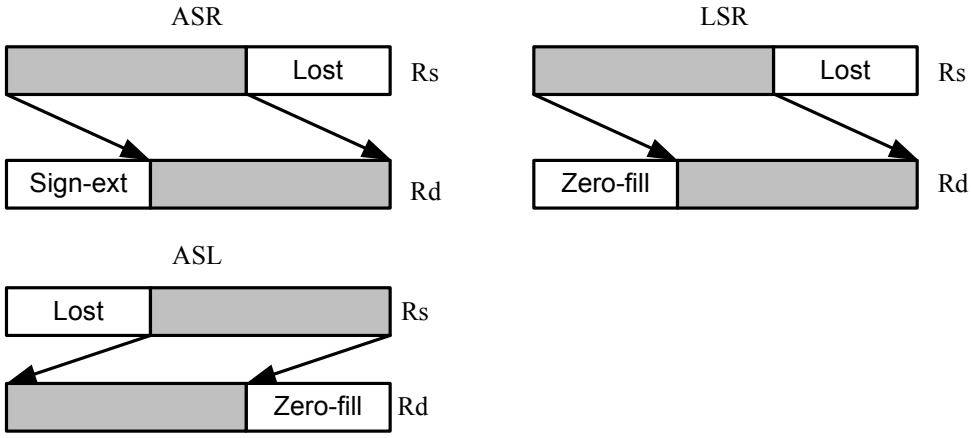
11.12.8 XTYPE/SHIFT

The XTYPE/SHIFT instruction subclass includes instructions which perform shifts.

Shift by immediate

Shift the source register value right or left based on the type of instruction. In these instructions, the shift amount is contained in an unsigned immediate (5 bits for 32-bit shifts, 6 bits for 64-bit shifts) and the shift instruction gives the shift direction.

Arithmetic right shifts place the sign bit of the source value in the vacated positions, while logical right shifts place zeros in the vacated positions. Left shifts always zero-fill the vacated bits.



Syntax	Behavior
`Rd=asl(Rs,#u5)`	`Rd = Rs << #u;`
`Rd=asr(Rs,#u5)`	`Rd = Rs >> #u;`
`Rd=lsr(Rs,#u5)`	`Rd = Rs >>> #u;`
`Rd=rol(Rs,#u5)`	`Rd = Rs <<R #u;`
`Rdd=asl(Rss,#u6)`	`Rdd = Rss << #u;`
`Rdd=asr(Rss,#u6)`	`Rdd = Rss >> #u;`
`Rdd=lsr(Rss,#u6)`	`Rdd = Rss >>> #u;`
`Rdd=rol(Rss,#u6)`	`Rdd = Rss <<R #u;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=asl(Rs,#u5)`	`Word32 Q6_R_asl_RI(Word32 Rs, Word32 Iu5)`
`Rd=asr(Rs,#u5)`	`Word32 Q6_R_asr_RI(Word32 Rs, Word32 Iu5)`
`Rd=lsr(Rs,#u5)`	`Word32 Q6_R_lsr_RI(Word32 Rs, Word32 Iu5)`

`Rd=rol(Rs,#u5)`	`Word32 Q6_R_rol_RI(Word32 Rs, Word32 Iu5)`
`Rdd=asl(Rss,#u6)`	`Word64 Q6_P_asl_PI(Word64 Rss, Word32 Iu6)`
`Rdd=asr(Rss,#u6)`	`Word64 Q6_P_asr_PI(Word64 Rss, Word32 Iu6)`
`Rdd=lsr(Rss,#u6)`	`Word64 Q6_P_lsr_PI(Word64 Rss, Word32 Iu6)`
`Rdd=rol(Rss,#u6)`	`Word64 Q6_P_rol_PI(Word64 Rss, Word32 Iu6)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=asr(Rss,#u6)
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=lsr(Rss,#u6)
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=asl(Rss,#u6)
1	0	0	0	0	0	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=rol(Rss,#u6)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=asr(Rs,#u5)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rd=lsr(Rs,#u5)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=asl(Rs,#u5)
1	0	0	0	1	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	1	**d**	**d**	**d**	**d**	**d**	Rd=rol(Rs,#u5)

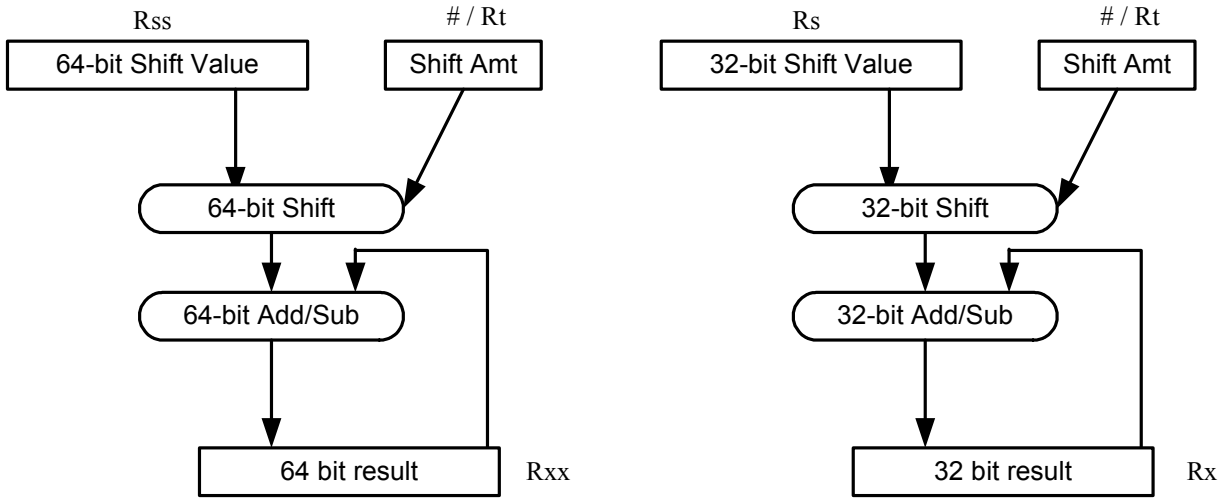
Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`MajOp`	Major Opcode
`MinOp`	Minor Opcode
`RegType`	Register Type

Shift by immediate and accumulate

Shift the source register value right or left based on the type of instruction. In these instructions, the shift amount is contained in an unsigned immediate (5 bits for 32-bit shifts, 6 bits for 64-bit shifts) and the shift instruction gives the shift direction.

Arithmetic right shifts place the sign bit of the source value in the vacated positions, while logical right shifts place zeros in the vacated positions. Left shifts always zero-fill the vacated bits.

After shifting, add or subtract the shifted value from the destination register or register pair.



Syntax	Behavior
`Rx=add(#u8,asl(Rx,#U5))`	`Rx=apply_extension(#u)+(Rx<<#U);`
`Rx=add(#u8,lsr(Rx,#U5))`	`Rx=apply_extension(#u)+(((unsigned int)Rx)>>#U);`
`Rx=sub(#u8,asl(Rx,#U5))`	`Rx=apply_extension(#u)-(Rx<<#U);`
`Rx=sub(#u8,lsr(Rx,#U5))`	`Rx=apply_extension(#u)-(((unsigned int)Rx)>>#U);`
`Rx[+-]=asl(Rs,#u5)`	`Rx = Rx [+-] Rs << #u;`
`Rx[+-]=asr(Rs,#u5)`	`Rx = Rx [+-] Rs >> #u;`
`Rx[+-]=lsr(Rs,#u5)`	`Rx = Rx [+-] Rs >>> #u;`
`Rx[+-]=rol(Rs,#u5)`	`Rx = Rx [+-] Rs <<R #u;`
`Rxx[+-]=asl(Rss,#u6)`	`Rxx = Rxx [+-] Rss << #u;`
`Rxx[+-]=asr(Rss,#u6)`	`Rxx = Rxx [+-] Rss >> #u;`
`Rxx[+-]=lsr(Rss,#u6)`	`Rxx = Rxx [+-] Rss >>> #u;`
`Rxx[+-]=rol(Rss,#u6)`	`Rxx = Rxx [+-] Rss <<R #u;`

Class: XTYPE (slots 2,3)

Intrinsics

Rx+=asl(Rs,#u5)	Word32 Q6_R_aslacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx+=asr(Rs,#u5)	Word32 Q6_R_asracc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx+=lsr(Rs,#u5)	Word32 Q6_R_lsracc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx+=rol(Rs,#u5)	Word32 Q6_R_rolacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx-=asl(Rs,#u5)	Word32 Q6_R_aslnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx-=asr(Rs,#u5)	Word32 Q6_R_asrnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx-=lsr(Rs,#u5)	Word32 Q6_R_lsrnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx-=rol(Rs,#u5)	Word32 Q6_R_rolnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx=add(#u8,asl(Rx,#U5))	Word32 Q6_R_add_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rx=add(#u8,lsr(Rx,#U5))	Word32 Q6_R_add_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rx=sub(#u8,asl(Rx,#U5))	Word32 Q6_R_sub_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rx=sub(#u8,lsr(Rx,#U5))	Word32 Q6_R_sub_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rxx+=asl(Rss,#u6)	Word64 Q6_P_aslacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx+=asr(Rss,#u6)	Word64 Q6_P_asracc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx+=lsr(Rss,#u6)	Word64 Q6_P_lsracc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx+=rol(Rss,#u6)	Word64 Q6_P_rolacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx-=asl(Rss,#u6)	Word64 Q6_P_aslnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx-=asr(Rss,#u6)	Word64 Q6_P_asrnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx-=lsr(Rss,#u6)	Word64 Q6_P_lsrnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx-=rol(Rss,#u6)	Word64 Q6_P_rolnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	0	x	x	x	x	x	Rxx-=asr(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	1	x	x	x	x	x	Rxx-=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	0	x	x	x	x	x	Rxx-=asl(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	1	x	x	x	x	x	Rxx-=rol(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	0	0	x	x	x	x	x	Rxx+=asr(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	0	1	x	x	x	x	x	Rxx+=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	1	0	x	x	x	x	x	Rxx+=asl(Rss,#u6)
1	0	0	0	0	0	1	0	0	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	1	1	x	x	x	x	x	Rxx+=rol(Rss,#u6)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	0	x	x	x	x	x	Rx-=asr(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	1	x	x	x	x	x	Rx-=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	0	x	x	x	x	x	Rx-=asl(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	1	x	x	x	x	x	Rx-=rol(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	0	0	x	x	x	x	x	Rx+=asr(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	0	1	x	x	x	x	x	Rx+=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	1	0	x	x	x	x	x	Rx+=asl(Rs,#u5)
1	0	0	0	1	1	1	0	0	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	1	1	x	x	x	x	x	Rx+=rol(Rs,#u5)
ICLASS				RegType							x5					Parse											MajOp					
1	1	0	1	1	1	1	0	i	i	i	x	x	x	x	x	P	P	i	l	l	l	l	l	i	i	i	0	i	1	0	-	Rx=add(#u8,asl(Rx,#U5))
1	1	0	1	1	1	1	0	i	i	i	x	x	x	x	x	P	P	i	l	l	l	l	l	i	i	i	0	i	1	1	-	Rx=sub(#u8,asl(Rx,#U5))
1	1	0	1	1	1	1	0	i	i	i	x	x	x	x	x	P	P	i	l	l	l	l	l	i	i	i	1	i	1	0	-	Rx=add(#u8,lsr(Rx,#U5))
1	1	0	1	1	1	1	0	i	i	i	x	x	x	x	x	P	P	i	l	l	l	l	l	i	i	i	1	i	1	1	-	Rx=sub(#u8,lsr(Rx,#U5))

Field name	Description
RegType	Register Type
MajOp	Major Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
x5	Field to encode register x
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Shift by immediate and add

Shift Rs left by 0-7 bits, add to Rt, and place the result in Rd.

This instruction is useful for calculating array pointers, where destruction of the base pointer is undesirable.

Syntax	Behavior
`Rd=addasl(Rt,Rs,#u3)`	`Rd = Rt + Rs << #u;`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=addasl(Rt,Rs,#u3)`	`Word32 Q6_R_addasl_RRI(Word32 Rt, Word32 Rs, Word32 Iu3)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	1	0	0	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**t**	**t**	**t**	**t**	**t**	**i**	**i**	**i**	**d**	**d**	**d**	**d**	**d**	Rd=addasl(Rt,Rs,#u3)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major Opcode
`Min`	Minor Opcode
`RegType`	Register Type

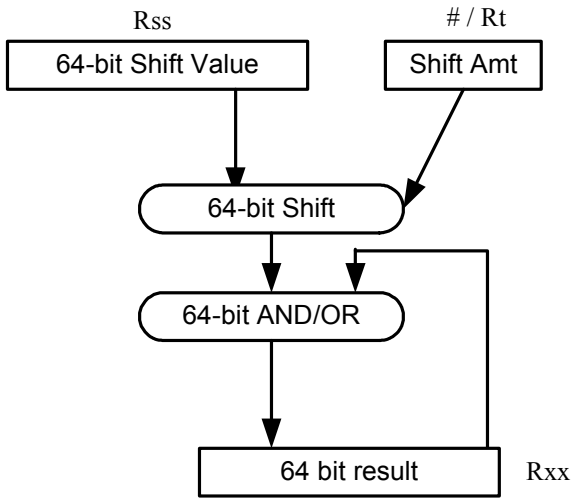
Shift by immediate and logical

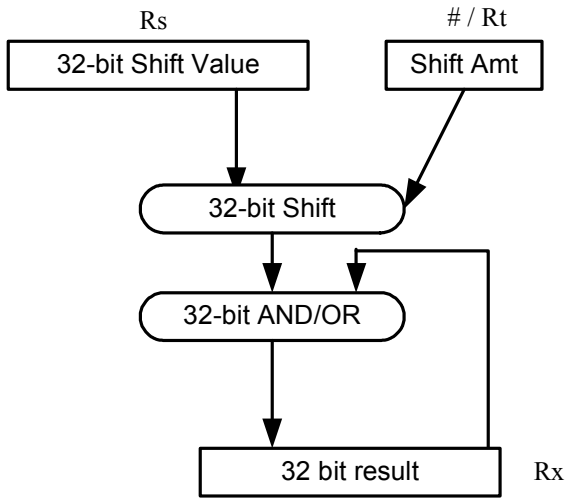
Shift the source register value right or left based on the type of instruction. In these instructions, the shift amount is contained in an unsigned immediate (5 bits for 32-bit shifts, 6 bits for 64-bit shifts) and the shift instruction gives the shift direction.

Arithmetic right shifts place the sign bit of the source value in the vacated positions, while logical right shifts place zeros in the vacated positions. Left shifts always zero-fill the vacated bits.

After shifting, take the logical AND, OR, or XOR of the shifted amount and the destination register or register pair, and place the result back in the destination register or register pair.

Saturation is not available for these instructions.





Syntax	Behavior
`Rx=and(#u8,asl(Rx,#U5))`	`Rx=apply_extension(#u)&(Rx<<#U);`
`Rx=and(#u8,lsr(Rx,#U5))`	`Rx=apply_extension(#u)&(((unsigned int)Rx)>>#U);`
`Rx=or(#u8,asl(Rx,#U5))`	`Rx=apply_extension(#u)|(Rx<<#U);`
`Rx=or(#u8,lsr(Rx,#U5))`	`Rx=apply_extension(#u)|(((unsigned int)Rx)>>#U);`
`Rx[&|]=asl(Rs,#u5)`	`Rx = Rx [|&] Rs << #u;`
`Rx[&|]=asr(Rs,#u5)`	`Rx = Rx [|&] Rs >> #u;`
`Rx[&|]=lsr(Rs,#u5)`	`Rx = Rx [|&] Rs >>> #u;`
`Rx[&|]=rol(Rs,#u5)`	`Rx = Rx [|&] Rs <<R #u;`
`Rx^=asl(Rs,#u5)`	`Rx = Rx ^ Rs << #u;`
`Rx^=lsr(Rs,#u5)`	`Rx = Rx ^ Rs >>> #u;`
`Rx^=rol(Rs,#u5)`	`Rx = Rx ^ Rs <<R #u;`
`Rxx[&|]=asl(Rss,#u6)`	`Rxx = Rxx [|&] Rss << #u;`
`Rxx[&|]=asr(Rss,#u6)`	`Rxx = Rxx [|&] Rss >> #u;`

Syntax	Behavior
Rxx[&|]=lsr(Rss,#u6)	Rxx = Rxx [|&] Rss >>> #u;
Rxx[&|]=rol(Rss,#u6)	Rxx = Rxx [|&] Rss <<$_R$ #u;
Rxx^=asl(Rss,#u6)	Rxx = Rxx ^ Rss << #u;
Rxx^=lsr(Rss,#u6)	Rxx = Rxx ^ Rss >>> #u;
Rxx^=rol(Rss,#u6)	Rxx = Rxx ^ Rss <<$_R$ #u;

Class: XTYPE (slots 2,3)

Intrinsics

Rx&=asl(Rs,#u5)	Word32 Q6_R_asland_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx&=asr(Rs,#u5)	Word32 Q6_R_asrand_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx&=lsr(Rs,#u5)	Word32 Q6_R_lsrand_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx&=rol(Rs,#u5)	Word32 Q6_R_roland_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx=and(#u8,asl(Rx,#U5))	Word32 Q6_R_and_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rx=and(#u8,lsr(Rx,#U5))	Word32 Q6_R_and_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rx=or(#u8,asl(Rx,#U5))	Word32 Q6_R_or_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rx=or(#u8,lsr(Rx,#U5))	Word32 Q6_R_or_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)
Rx^=asl(Rs,#u5)	Word32 Q6_R_aslxacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx^=lsr(Rs,#u5)	Word32 Q6_R_lsrxacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx^=rol(Rs,#u5)	Word32 Q6_R_rolxacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx|=asl(Rs,#u5)	Word32 Q6_R_aslor_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx|=asr(Rs,#u5)	Word32 Q6_R_asror_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx|=lsr(Rs,#u5)	Word32 Q6_R_lsror_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rx|=rol(Rs,#u5)	Word32 Q6_R_rolor_RI(Word32 Rx, Word32 Rs, Word32 Iu5)
Rxx&=asl(Rss,#u6)	Word64 Q6_P_asland_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx&=asr(Rss,#u6)	Word64 Q6_P_asrand_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)
Rxx&=lsr(Rss,#u6)	Word64 Q6_P_lsrand_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)

`Rxx&=rol(Rss,#u6)`	`Word64 Q6_P_roland_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx^=asl(Rss,#u6)`	`Word64 Q6_P_aslxacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx^=lsr(Rss,#u6)`	`Word64 Q6_P_lsrxacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx^=rol(Rss,#u6)`	`Word64 Q6_P_rolxacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx|=asl(Rss,#u6)`	`Word64 Q6_P_aslor_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx|=asr(Rss,#u6)`	`Word64 Q6_P_asror_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx|=lsr(Rss,#u6)`	`Word64 Q6_P_lsror_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`
`Rxx|=rol(Rss,#u6)`	`Word64 Q6_P_rolor_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)`

Encoding

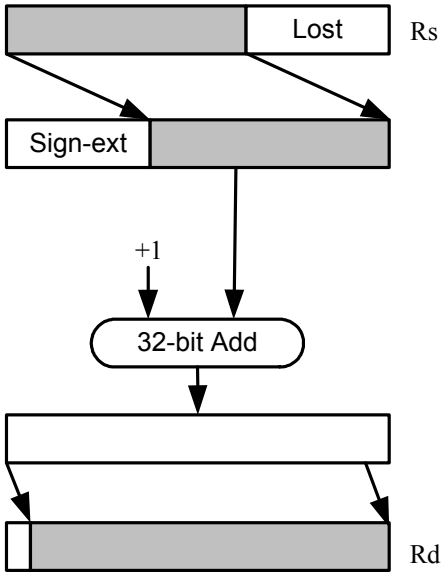
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			x5					
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	0	x	x	x	x	x	Rxx&=asr(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	1	x	x	x	x	x	Rxx&=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	0	x	x	x	x	x	Rxx&=asl(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	1	x	x	x	x	x	Rxx&=rol(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	0	0	x	x	x	x	x	Rxx|=asr(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	0	1	x	x	x	x	x	Rxx|=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	1	0	x	x	x	x	x	Rxx|=asl(Rss,#u6)
1	0	0	0	0	0	1	0	0	1	-	s	s	s	s	s	P	P	i	i	i	i	i	i	1	1	1	x	x	x	x	x	Rxx|=rol(Rss,#u6)
1	0	0	0	0	0	1	0	1	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	0	1	x	x	x	x	x	Rxx^=lsr(Rss,#u6)
1	0	0	0	0	0	1	0	1	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	0	x	x	x	x	x	Rxx^=asl(Rss,#u6)
1	0	0	0	0	0	1	0	1	0	-	s	s	s	s	s	P	P	i	i	i	i	i	i	0	1	1	x	x	x	x	x	Rxx^=rol(Rss,#u6)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	0	x	x	x	x	x	Rx&=asr(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	1	x	x	x	x	x	Rx&=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	0	x	x	x	x	x	Rx&=asl(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	1	x	x	x	x	x	Rx&=rol(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	0	0	x	x	x	x	x	Rx|=asr(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	0	1	x	x	x	x	x	Rx|=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	1	0	x	x	x	x	x	Rx|=asl(Rs,#u5)
1	0	0	0	1	1	1	0	0	1	-	s	s	s	s	s	P	P	0	i	i	i	i	i	1	1	1	x	x	x	x	x	Rx|=rol(Rs,#u5)
1	0	0	0	1	1	1	0	1	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	0	1	x	x	x	x	x	Rx^=lsr(Rs,#u5)
1	0	0	0	1	1	1	0	1	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	0	x	x	x	x	x	Rx^=asl(Rs,#u5)
1	0	0	0	1	1	1	0	1	0	-	s	s	s	s	s	P	P	0	i	i	i	i	i	0	1	1	x	x	x	x	x	Rx^=rol(Rs,#u5)
ICLASS				RegType							x5					Parse											MajOp					
1	1	0	1	1	1	1	0	i	i	i	x	x	x	x	x	P	P	i	l	l	l	l	l	i	i	i	0	i	0	0	-	Rx=and(#u8,asl(Rx,#U5))
1	1	0	1	1	1	1	0	i	i	i	x	x	x	x	x	P	P	i	l	l	l	l	l	i	i	i	0	i	0	1	-	Rx=or(#u8,asl(Rx,#U5))
1	1	0	1	1	1	1	0	i	i	i	x	x	x	x	x	P	P	i	l	l	l	l	l	i	i	i	1	i	0	0	-	Rx=and(#u8,lsr(Rx,#U5))
1	1	0	1	1	1	1	0	i	i	i	x	x	x	x	x	P	P	i	l	l	l	l	l	i	i	i	1	i	0	1	-	Rx=or(#u8,lsr(Rx,#U5))

Field name	Description
RegType	Register Type
MajOp	Major Opcode
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
x5	Field to encode register x
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Shift right by immediate with rounding

Perform an arithmetic right shift by an immediate amount, and then round the result. This instruction works by first shifting right, then adding the value +1 to the result, and finally shifting right again by one bit. The right shifts always inserts the sign-bit in the vacated position.

When using asrrnd, the assembler adjusts the immediate appropriately.



Syntax	Behavior
`Rd=asr(Rs,#u5):rnd`	`Rd = ((Rs >> #u)+1) >> 1;`

`Rd=asrrnd(Rs,#u5)`	`if ("#u5==0") {`
	`  Assembler mapped to: "Rd=Rs";`
	`} else {`
	`  Assembler mapped to: "Rd=asr(Rs,#u5-1):rnd";`
	`};`

`Rdd=asr(Rss,#u6):rnd`	`tmp = Rss >> #u;`
	`rnd = tmp & 1;`
	`Rdd = tmp >> 1 + rnd;`

`Rdd=asrrnd(Rss,#u6)`	`if ("#u6==0") {`
	`  Assembler mapped to: "Rdd=Rss";`
	`} else {`
	`  Assembler mapped to: "Rdd=asr(Rss,#u6-1):rnd";`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=asr(Rs,#u5):rnd`	`Word32 Q6_R_asr_RI_rnd(Word32 Rs, Word32 Iu5)`
`Rd=asrrnd(Rs,#u5)`	`Word32 Q6_R_asrrnd_RI(Word32 Rs, Word32 Iu5)`
`Rdd=asr(Rss,#u6):rnd`	`Word64 Q6_P_asr_PI_rnd(Word64 Rss, Word32 Iu6)`
`Rdd=asrrnd(Rss,#u6)`	`Word64 Q6_P_asrrnd_PI(Word64 Rss, Word32 Iu6)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	**i**	**i**	**i**	**i**	**i**	**i**	1	1	1	**d**	**d**	**d**	**d**	**d**	Rdd=asr(Rss,#u6):rnd
1	0	0	0	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rd=asr(Rs,#u5):rnd

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Shift left by immediate with saturation

Perform a left shift of the 32-bit source register value by an immediate amount and saturate.

Saturation works by first sign-extending the 32-bit Rs register to 64 bits. It is then left shifted by the immediate amount. If this 64-bit value cannot fit in a signed 32-bit number (the upper word is not the sign-extension of bit 31), then saturation is performed based on the sign of the original value. Saturation clamps the 32-bit result to the range 0x8000_0000 to 0x7fff_ffff.

Syntax	Behavior
`Rd=asl(Rs,#u5):sat`	`Rd = sat_32(sxt`$_{32->64}$`(Rs) << #u);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=asl(Rs,#u5):sat`	`Word32 Q6_R_asl_RI_sat(Word32 Rs, Word32 Iu5)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	1	1	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=asl(Rs,#u5):sat

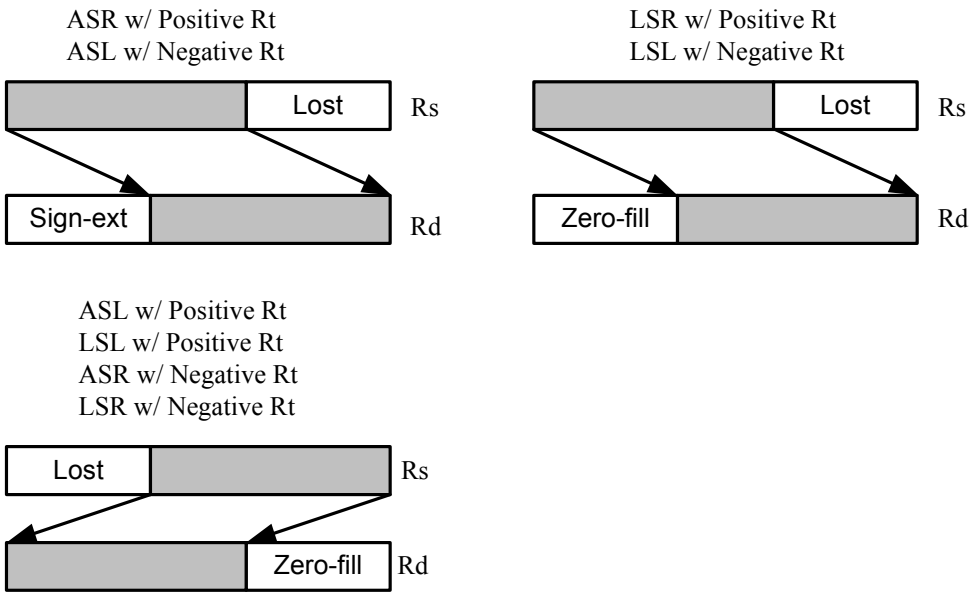
Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Shift by register

The shift amount is the least significant 7 bits of Rt, treated as a two's complement value. If the shift amount is negative (bit 6 of Rt is set), the direction of the shift indicted in the opcode is reversed (see Figure).

The source data to be shifted is always performed as a 64-bit shift. When the Rs source register is a 32-bit register, this register is first sign or zero-extended to 64-bits. Arithmetic shifts sign-extend the 32-bit source to 64-bits, while logical shifts zero extend.

The 64-bit source value is then right or left shifted based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior
`Rd=asl(Rs,Rt)`	`shamt=sxt7->32(Rt);` `Rd = (shamt>0)?(sxt32->64(Rs)<<shamt):(sxt32->64(Rs)>>shamt);`
`Rd=asr(Rs,Rt)`	`shamt=sxt7->32(Rt);` `Rd = (shamt>0)?(sxt32->64(Rs)>>shamt):(sxt32->64(Rs)<<shamt);`
`Rd=lsl(#s6,Rt)`	`shamt = sxt7->32(Rt);` `Rd = (shamt>0)?(zxt32->64(#s)<<shamt):(zxt32->64(#s)>>>shamt);`
`Rd=lsl(Rs,Rt)`	`shamt=sxt7->32(Rt);` `Rd = (shamt>0)?(zxt32->64(Rs)<<shamt):(zxt32->64(Rs)>>>shamt);`

Syntax	Behavior
Rd=lsr(Rs,Rt)	shamt=sxt$_{7->32}$(Rt); Rd = (shamt>0)?(zxt$_{32->64}$(Rs)>>>shamt):(zxt$_{32->64}$(Rs)<<shamt);
Rdd=asl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rdd = (shamt>0)?(Rss<<shamt):(Rss>>shamt);
Rdd=asr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rdd = (shamt>0)?(Rss>>shamt):(Rss<<shamt);
Rdd=lsl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rdd = (shamt>0)?(Rss<<shamt):(Rss>>>shamt);
Rdd=lsr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rdd = (shamt>0)?(Rss>>>shamt):(Rss<<shamt);

Class: XTYPE (slots 2,3)

Intrinsics

Rd=asl(Rs,Rt)	Word32 Q6_R_asl_RR(Word32 Rs, Word32 Rt)
Rd=asr(Rs,Rt)	Word32 Q6_R_asr_RR(Word32 Rs, Word32 Rt)
Rd=lsl(#s6,Rt)	Word32 Q6_R_lsl_IR(Word32 Is6, Word32 Rt)
Rd=lsl(Rs,Rt)	Word32 Q6_R_lsl_RR(Word32 Rs, Word32 Rt)
Rd=lsr(Rs,Rt)	Word32 Q6_R_lsr_RR(Word32 Rs, Word32 Rt)
Rdd=asl(Rss,Rt)	Word64 Q6_P_asl_PR(Word64 Rss, Word32 Rt)
Rdd=asr(Rss,Rt)	Word64 Q6_P_asr_PR(Word64 Rss, Word32 Rt)
Rdd=lsl(Rss,Rt)	Word64 Q6_P_lsl_PR(Word64 Rss, Word32 Rt)
Rdd=lsr(Rss,Rt)	Word64 Q6_P_lsr_PR(Word64 Rss, Word32 Rt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=asr(Rss,Rt)
1	1	0	0	0	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=lsr(Rss,Rt)
1	1	0	0	0	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=asl(Rss,Rt)
1	1	0	0	0	0	1	1	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=lsl(Rss,Rt)
1	1	0	0	0	1	1	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=asr(Rs,Rt)
1	1	0	0	0	1	1	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rd=lsr(Rs,Rt)
1	1	0	0	0	1	1	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=asl(Rs,Rt)
1	1	0	0	0	1	1	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rd=lsl(Rs,Rt)
ICLASS				RegType				Maj								Parse			t5					Min			d5					
1	1	0	0	0	1	1	0	1	0	-	**i**	**i**	**i**	**i**	**i**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	**i**	**d**	**d**	**d**	**d**	**d**	Rd=lsl(#s6,Rt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

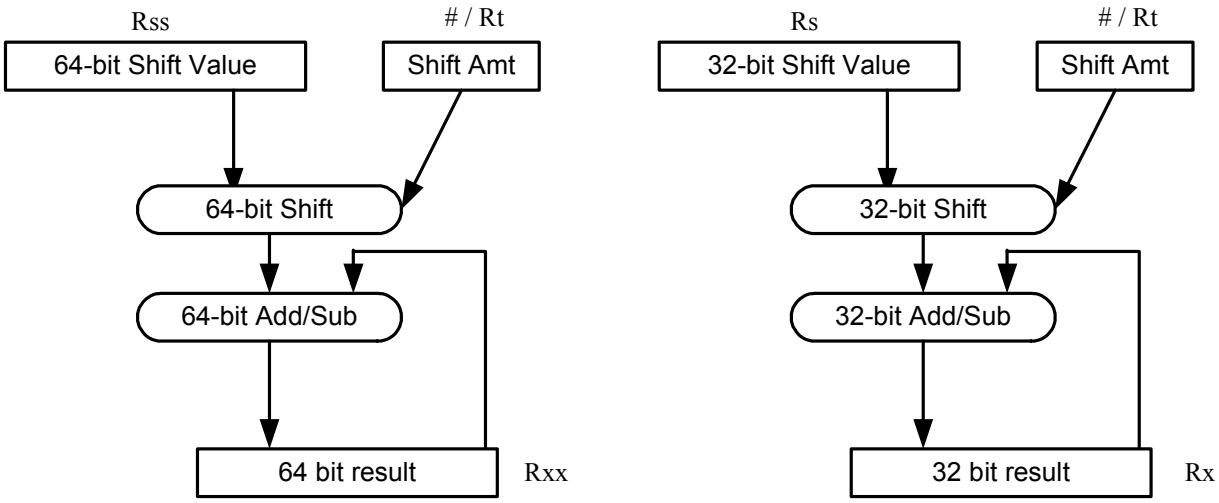
Shift by register and accumulate

The shift amount is the least significant 7 bits of Rt, treated as a two's complement value. If the shift amount is negative (bit 6 of Rt is set), the direction of the shift indicted in the opcode is reversed.

Shift the source register value right or left based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.

The shift operation is always performed as a 64-bit shift. When Rs is a 32-bit register, this register is first sign- or zero-extended to 64-bits. Arithmetic shifts sign-extend the 32-bit source to 64-bits, while logical shifts zero extend.

After shifting, add or subtract the 64-bit shifted amount from the destination register or register pair.



Syntax	Behavior
`Rx[+-]=asl(Rs,Rt)`	`shamt=sxt7->32(Rt); Rx = Rx [+-] (shamt>0)?(sxt32->64(Rs)<<shamt):(sxt32->64(Rs)>>shamt);`
`Rx[+-]=asr(Rs,Rt)`	`shamt=sxt7->32(Rt); Rx = Rx [+-] (shamt>0)?(sxt32->64(Rs)>>shamt):(sxt32->64(Rs)<<shamt);`
`Rx[+-]=lsl(Rs,Rt)`	`shamt=sxt7->32(Rt); Rx = Rx [+-] (shamt>0)?(zxt32->64(Rs)<<shamt):(zxt32->64(Rs)>>>shamt);`

Syntax	Behavior
Rx[+-]=lsr(Rs,Rt)	shamt=sxt$_{7->32}$(Rt); Rx = Rx [+-] (shamt>0)?(zxt$_{32->64}$(Rs)>>>shamt):(zxt$_{32->64}$(Rs)<<shamt);
Rxx[+-]=asl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [+-] (shamt>0)?(Rss<<shamt):(Rss>>shamt);
Rxx[+-]=asr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [+-] (shamt>0)?(Rss>>shamt):(Rss<<shamt);
Rxx[+-]=lsl(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [+-] (shamt>0)?(Rss<<shamt):(Rss>>>shamt);
Rxx[+-]=lsr(Rss,Rt)	shamt=sxt$_{7->32}$(Rt); Rxx = Rxx [+-] (shamt>0)?(Rss>>>shamt):(Rss<<shamt);

Class: XTYPE (slots 2,3)

Intrinsics

Rx+=asl(Rs,Rt)	Word32 Q6_R_aslacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=asr(Rs,Rt)	Word32 Q6_R_asracc_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=lsl(Rs,Rt)	Word32 Q6_R_lslacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx+=lsr(Rs,Rt)	Word32 Q6_R_lsracc_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=asl(Rs,Rt)	Word32 Q6_R_aslnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=asr(Rs,Rt)	Word32 Q6_R_asrnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=lsl(Rs,Rt)	Word32 Q6_R_lslnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx-=lsr(Rs,Rt)	Word32 Q6_R_lsrnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rxx+=asl(Rss,Rt)	Word64 Q6_P_aslacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx+=asr(Rss,Rt)	Word64 Q6_P_asracc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx+=lsl(Rss,Rt)	Word64 Q6_P_lslacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)

`Rxx+=lsr(Rss,Rt)`	`Word64 Q6_P_lsracc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx-=asl(Rss,Rt)`	`Word64 Q6_P_aslnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx-=asr(Rss,Rt)`	`Word64 Q6_P_asrnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx-=lsl(Rss,Rt)`	`Word64 Q6_P_lslnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`
`Rxx-=lsr(Rss,Rt)`	`Word64 Q6_P_lsrnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			x5					
1	1	0	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rxx-=asr(Rss,Rt)
1	1	0	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rxx-=lsr(Rss,Rt)
1	1	0	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rxx-=asl(Rss,Rt)
1	1	0	0	1	0	1	1	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rxx-=lsl(Rss,Rt)
1	1	0	0	1	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rxx+=asr(Rss,Rt)
1	1	0	0	1	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rxx+=lsr(Rss,Rt)
1	1	0	0	1	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rxx+=asl(Rss,Rt)
1	1	0	0	1	0	1	1	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rxx+=lsl(Rss,Rt)
1	1	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rx-=asr(Rs,Rt)
1	1	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rx-=lsr(Rs,Rt)
1	1	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rx-=asl(Rs,Rt)
1	1	0	0	1	1	0	0	1	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rx-=lsl(Rs,Rt)
1	1	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rx+=asr(Rs,Rt)
1	1	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rx+=lsr(Rs,Rt)
1	1	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rx+=asl(Rs,Rt)
1	1	0	0	1	1	0	0	1	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rx+=lsl(Rs,Rt)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`s5`	Field to encode register s
`t5`	Field to encode register t
`x5`	Field to encode register x
`Maj`	Major Opcode
`Min`	Minor Opcode
`RegType`	Register Type

Shift by register and logical

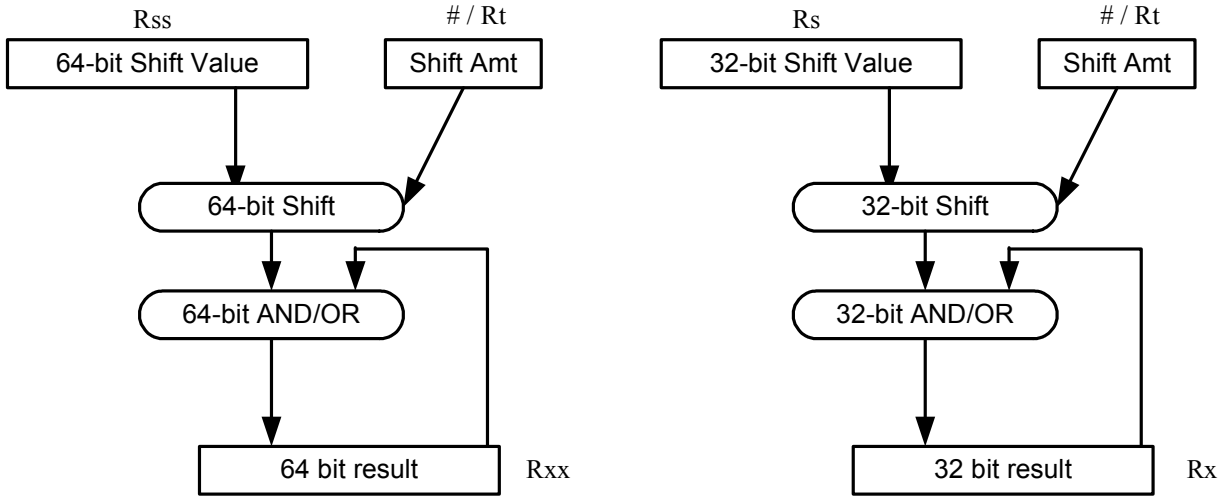
The shift amount is the least significant 7 bits of Rt, treated as a two's complement value. If the shift amount is negative (bit 6 of Rt is set), the direction of the shift indicted in the opcode is reversed.

Shift the source register value right or left based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.

The shift operation is always performed as a 64-bit shift. When the Rs source register is a 32-bit register, this register is first sign or zero-extended to 64-bits. Arithmetic shifts sign-extend the 32-bit source to 64-bits, while logical shifts zero extend.

After shifting, take the logical AND or OR of the shifted amount and the destination register or register pair, and place the result back in the destination register or register pair.

Saturation is not available for these instructions.



Syntax	Behavior
`Rx[&|]=asl(Rs,Rt)`	`shamt=sxt7->32(Rt); Rx = Rx [|&] (shamt>0)?(sxt32->64(Rs)<<shamt):(sxt32->64(Rs)>>shamt);`
`Rx[&|]=asr(Rs,Rt)`	`shamt=sxt7->32(Rt); Rx = Rx [|&] (shamt>0)?(sxt32->64(Rs)>>shamt):(sxt32->64(Rs)<<shamt);`
`Rx[&|]=lsl(Rs,Rt)`	`shamt=sxt7->32(Rt); Rx = Rx [|&] (shamt>0)?(zxt32->64(Rs)<<shamt):(zxt32->64(Rs)>>>shamt);`
`Rx[&|]=lsr(Rs,Rt)`	`shamt=sxt7->32(Rt); Rx = Rx [|&] (shamt>0)?(zxt32->64(Rs)>>>shamt):(zxt32->64(Rs)<<shamt);`
`Rxx[&|]=asl(Rss,Rt)`	`shamt=sxt7->32(Rt); Rxx = Rxx [|&] (shamt>0)?(Rss<<shamt):(Rss>>shamt);`
`Rxx[&|]=asr(Rss,Rt)`	`shamt=sxt7->32(Rt); Rxx = Rxx [|&] (shamt>0)?(Rss>>shamt):(Rss<<shamt);`
`Rxx[&|]=lsl(Rss,Rt)`	`shamt=sxt7->32(Rt); Rxx = Rxx [|&] (shamt>0)?(Rss<<shamt):(Rss>>>shamt);`
`Rxx[&|]=lsr(Rss,Rt)`	`shamt=sxt7->32(Rt); Rxx = Rxx [|&] (shamt>0)?(Rss>>>shamt):(Rss<<shamt);`
`Rxx^=asl(Rss,Rt)`	`shamt=sxt7->32(Rt); Rxx = Rxx ^ (shamt>0)?(Rss<<shamt):(Rss>>shamt);`
`Rxx^=asr(Rss,Rt)`	`shamt=sxt7->32(Rt); Rxx = Rxx ^ (shamt>0)?(Rss>>shamt):(Rss<<shamt);`
`Rxx^=lsl(Rss,Rt)`	`shamt=sxt7->32(Rt); Rxx = Rxx ^ (shamt>0)?(Rss<<shamt):(Rss>>>shamt);`
`Rxx^=lsr(Rss,Rt)`	`shamt=sxt7->32(Rt); Rxx = Rxx ^ (shamt>0)?(Rss>>>shamt):(Rss<<shamt);`

Class: XTYPE (slots 2,3)

Intrinsics

Rx&=asl(Rs,Rt)	Word32 Q6_R_asland_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx&=asr(Rs,Rt)	Word32 Q6_R_asrand_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx&=lsl(Rs,Rt)	Word32 Q6_R_lsland_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx&=lsr(Rs,Rt)	Word32 Q6_R_lsrand_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx|=asl(Rs,Rt)	Word32 Q6_R_aslor_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx|=asr(Rs,Rt)	Word32 Q6_R_asror_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx|=lsl(Rs,Rt)	Word32 Q6_R_lslor_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rx|=lsr(Rs,Rt)	Word32 Q6_R_lsror_RR(Word32 Rx, Word32 Rs, Word32 Rt)
Rxx&=asl(Rss,Rt)	Word64 Q6_P_asland_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx&=asr(Rss,Rt)	Word64 Q6_P_asrand_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx&=lsl(Rss,Rt)	Word64 Q6_P_lsland_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx&=lsr(Rss,Rt)	Word64 Q6_P_lsrand_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx^=asl(Rss,Rt)	Word64 Q6_P_aslxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx^=asr(Rss,Rt)	Word64 Q6_P_asrxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx^=lsl(Rss,Rt)	Word64 Q6_P_lslxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx^=lsr(Rss,Rt)	Word64 Q6_P_lsrxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx|=asl(Rss,Rt)	Word64 Q6_P_aslor_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx|=asr(Rss,Rt)	Word64 Q6_P_asror_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx|=lsl(Rss,Rt)	Word64 Q6_P_lslor_PR(Word64 Rxx, Word64 Rss, Word32 Rt)
Rxx|=lsr(Rss,Rt)	Word64 Q6_P_lsror_PR(Word64 Rxx, Word64 Rss, Word32 Rt)

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			x5					
1	1	0	0	1	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rxx|=asr(Rss,Rt)
1	1	0	0	1	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rxx|=lsr(Rss,Rt)
1	1	0	0	1	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rxx|=asl(Rss,Rt)
1	1	0	0	1	0	1	1	0	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rxx|=lsl(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rxx&=asr(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rxx&=lsr(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rxx&=asl(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rxx&=lsl(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rxx^=asr(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rxx^=lsr(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rxx^=asl(Rss,Rt)
1	1	0	0	1	0	1	1	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rxx^=lsl(Rss,Rt)
1	1	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rx|=asr(Rs,Rt)
1	1	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rx|=lsr(Rs,Rt)
1	1	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rx|=asl(Rs,Rt)
1	1	0	0	1	1	0	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rx|=lsl(Rs,Rt)
1	1	0	0	1	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**x**	**x**	**x**	**x**	**x**	Rx&=asr(Rs,Rt)
1	1	0	0	1	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**x**	**x**	**x**	**x**	**x**	Rx&=lsr(Rs,Rt)
1	1	0	0	1	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**x**	**x**	**x**	**x**	**x**	Rx&=asl(Rs,Rt)
1	1	0	0	1	1	0	0	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**x**	**x**	**x**	**x**	**x**	Rx&=lsl(Rs,Rt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
s5	Field to encode register s
t5	Field to encode register t
x5	Field to encode register x
Maj	Major Opcode
Min	Minor Opcode
RegType	Register Type

Shift by register with saturation

The shift amount is the least significant 7 bits of Rt, treated as a two's complement value. If the shift amount is negative (bit 6 of Rt is set), the direction of the shift indicted in the opcode is reversed.

Saturation is available for 32-bit arithmetic left shifts. This can be either an ASL instruction with positive Rt, or an ASR instruction with negative Rt. Saturation works by first sign-extending the 32-bit Rs register to 64 bits. It is then shifted by the shift amount. If this 64-bit value cannot fit in a signed 32-bit number (the upper word is not the sign-extension of bit 31), then saturation is performed based on the sign of the original value. Saturation clamps the 32-bit result to the range 0x80000000 to 0x7fffffff.

Syntax	Behavior
`Rd=asl(Rs,Rt):sat`	`shamt=sxt`$_{7->32}$`(Rt);` `Rd = bidir_shiftl(Rs,shamt);`
`Rd=asr(Rs,Rt):sat`	`shamt=sxt`$_{7->32}$`(Rt);` `Rd = bidir_shiftr(Rs,shamt);`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=asl(Rs,Rt):sat`	`Word32 Q6_R_asl_RR_sat(Word32 Rs, Word32 Rt)`
`Rd=asr(Rs,Rt):sat`	`Word32 Q6_R_asr_RR_sat(Word32 Rs, Word32 Rt)`

Encoding

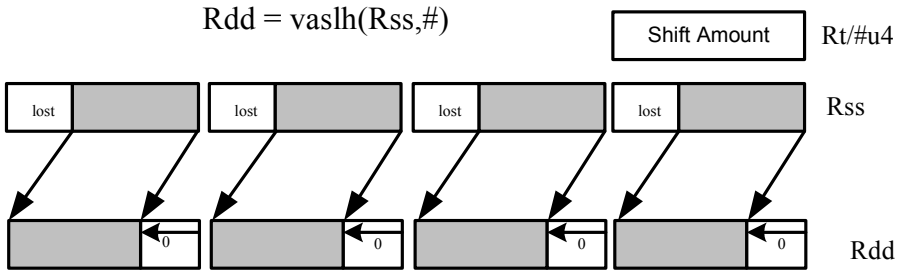
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	1	1	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rd=asr(Rs,Rt):sat
1	1	0	0	0	1	1	0	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rd=asl(Rs,Rt):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t

Field name	Description
`Maj`	Major Opcode
`Min`	Minor Opcode
`RegType`	Register Type

Vector shift halfwords by immediate

Shift individual halfwords of the source vector. Arithmetic right shifts place the sign bit of the source values in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior

`Rdd=vaslh(Rss,#u4)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.h[i]<<#u);`
	`};`

`Rdd=vasrh(Rss,#u4)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.h[i]>>#u);`
	`};`

`Rdd=vlsrh(Rss,#u4)`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=(Rss.uh[i]>>#u);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vaslh(Rss,#u4)`	`Word64 Q6_P_vaslh_PI(Word64 Rss, Word32 Iu4)`
`Rdd=vasrh(Rss,#u4)`	`Word64 Q6_P_vasrh_PI(Word64 Rss, Word32 Iu4)`
`Rdd=vlsrh(Rss,#u4)`	`Word64 Q6_P_vlsrh_PI(Word64 Rss, Word32 Iu4)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vasrh(Rss,#u4)
1	0	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vlsrh(Rss,#u4)
1	0	0	0	0	0	0	0	1	0	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vaslh(Rss,#u4)

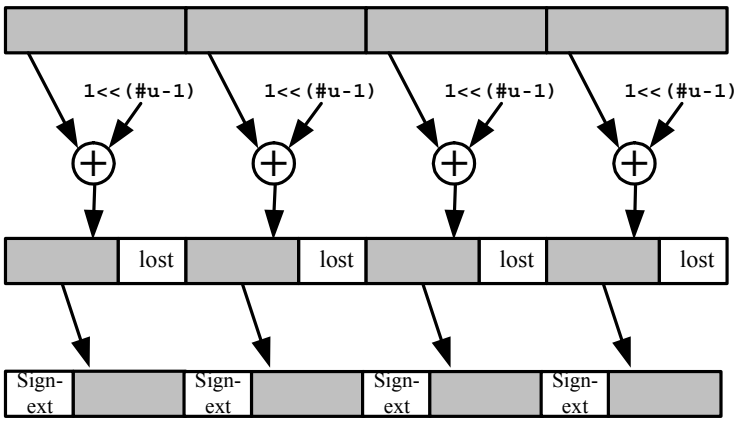
Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d

Field name	Description
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Vector arithmetic shift halfwords with round

For each halfword in the vector, round then arithmetic shift right by an immediate amount. The results are stored in the destination register.

Rdd = vasrh(Rss,#u):rnd



Syntax	Behavior

`Rdd=vasrh(Rss,#u4):raw`	`for (i=0;i<4;i++) {`
	`  Rdd.h[i]=( ((Rss.h[i] >> #u)+1)>>1 );`
	`};`

`Rdd=vasrh(Rss,#u4):rnd`	`if ("#u4==0") {`
	`  Assembler mapped to: "Rdd=Rss";`
	`} else {`
	`  Assembler mapped to: "Rdd=vasrh(Rss,#u4-1):raw";`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vasrh(Rss,#u4):rnd`	`Word64 Q6_P_vasrh_PI_rnd(Word64 Rss, Word32 Iu4)`

Encoding

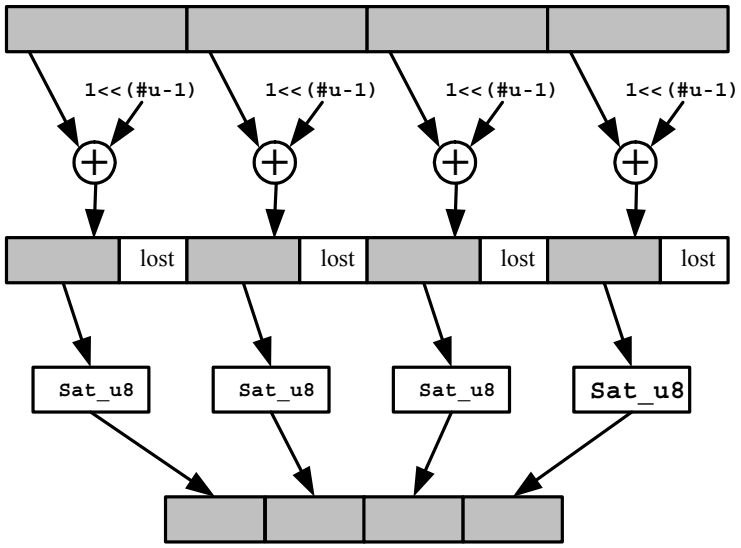
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse							MinOp			d5						
1	0	0	0	0	0	0	0	0	0	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vasrh(Rss,#u4):raw

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Vector arithmetic shift halfwords with saturate and pack

For each halfword in the vector, optionally round, then arithmetic shift right by an immediate amount. The results are saturated to unsigned [0-255] and then packed in the destination register.

Rd = vasrhub(Rss,#u):rnd:sat



Syntax	Behavior

`Rd=vasrhub(Rss,#u4):raw`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=usat_8(((Rss.h[i] >> #u )+1)>>1);`
	`};`

`Rd=vasrhub(Rss,#u4):rnd:sat`	`if ("#u4==0") {`
	`  Assembler mapped to: "Rd=vsathub(Rss)";`
	`} else {`
	`  Assembler mapped to: "Rd=vasrhub(Rss,#u4-1):raw";`
	`};`

`Rd=vasrhub(Rss,#u4):sat`	`for (i=0;i<4;i++) {`
	`  Rd.b[i]=usat_8(Rss.h[i] >> #u);`
	`};`

Class: XTYPE (slots 2,3)

Notes

- If saturation occurs during execution of this instruction (a result is clamped to either maximum or minimum values), then the OVF bit in the Status Register is set. OVF will remain set until explicitly cleared by a transfer to SR.

Intrinsics

`Rd=vasrhub(Rss,#u4):rnd:sat`	`Word32 Q6_R_vasrhub_PI_rnd_sat(Word64 Rss, Word32 Iu4)`
`Rd=vasrhub(Rss,#u4):sat`	`Word32 Q6_R_vasrhub_PI_sat(Word64 Rss, Word32 Iu4)`

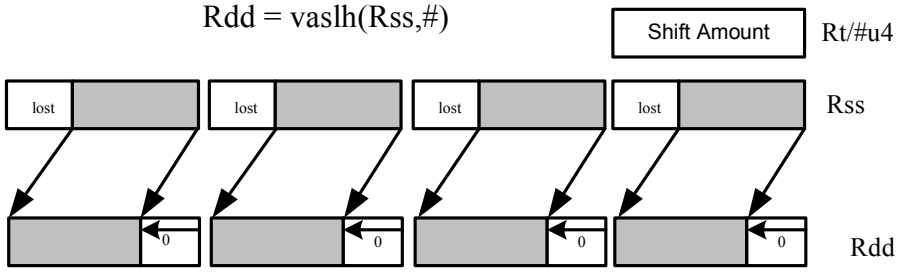
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	1	0	0	**d**	**d**	**d**	**d**	**d**	Rd=vasrhub(Rss,#u4):raw
1	0	0	0	1	0	0	0	0	1	1	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	0	**i**	**i**	**i**	**i**	1	0	1	**d**	**d**	**d**	**d**	**d**	Rd=vasrhub(Rss,#u4):sat

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Vector shift halfwords by register

The shift amount is the least significant 7 bits of Rt, treated as a two's complement value. If the shift amount is negative, the direction of the shift is reversed. Shift the source values right or left based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior
`Rdd=vaslh(Rss,Rt)`	`for (i=0;i<4;i++) { Rdd.h[i]=(sxt7->32(Rt)>0)?(sxt16->64(Rss.h[i])<<sxt7->32(Rt)):(sxt16->64(Rss.h[i])>>sxt7->32(Rt)); };`
`Rdd=vasrh(Rss,Rt)`	`for (i=0;i<4;i++) { Rdd.h[i]=(sxt7->32(Rt)>0)?(sxt16->64(Rss.h[i])>>sxt7->32(Rt)):(sxt16->64(Rss.h[i])<<sxt7->32(Rt)); };`
`Rdd=vlslh(Rss,Rt)`	`for (i=0;i<4;i++) { Rdd.h[i]=(sxt7->32(Rt)>0)?(zxt16->64(Rss.uh[i])<<sxt7->32(Rt)):(zxt16->64(Rss.uh[i])>>>sxt7->32(Rt)); };`
`Rdd=vlsrh(Rss,Rt)`	`for (i=0;i<4;i++) { Rdd.h[i]=(sxt7->32(Rt)>0)?(zxt16->64(Rss.uh[i])>>>sxt7->32(Rt)):(zxt16->64(Rss.uh[i])<<sxt7->32(Rt)); };`

Class: XTYPE (slots 2,3)

Notes

- If the number of bits to be shifted is greater than the width of the vector element, the result is either all sign-bits (for arithmetic right shifts) or all zeros for logical and left shifts.

Intrinsics

`Rdd=vaslh(Rss,Rt)`	`Word64 Q6_P_vaslh_PR(Word64 Rss, Word32 Rt)`
`Rdd=vasrh(Rss,Rt)`	`Word64 Q6_P_vasrh_PR(Word64 Rss, Word32 Rt)`
`Rdd=vlslh(Rss,Rt)`	`Word64 Q6_P_vlslh_PR(Word64 Rss, Word32 Rt)`
`Rdd=vlsrh(Rss,Rt)`	`Word64 Q6_P_vlsrh_PR(Word64 Rss, Word32 Rt)`

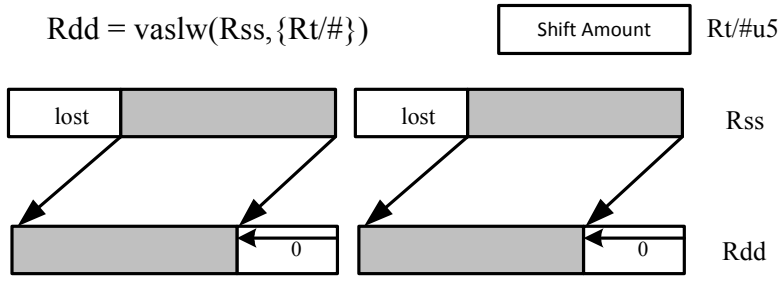
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vasrh(Rss,Rt)
1	1	0	0	0	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vlsrh(Rss,Rt)
1	1	0	0	0	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vaslh(Rss,Rt)
1	1	0	0	0	0	1	1	0	1	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vlslh(Rss,Rt)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major Opcode
`Min`	Minor Opcode
`RegType`	Register Type

Vector shift words by immediate

Shift individual words of the source vector. Arithmetic right shifts place the sign bit of the source values in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior

`Rdd=vaslw(Rss,#u5)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=(Rss.w[i]<<#u);`
	`};`

`Rdd=vasrw(Rss,#u5)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=(Rss.w[i]>>#u);`
	`};`

`Rdd=vlsrw(Rss,#u5)`	`for (i=0;i<2;i++) {`
	`  Rdd.w[i]=(Rss.uw[i]>>#u);`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rdd=vaslw(Rss,#u5)`	`Word64 Q6_P_vaslw_PI(Word64 Rss, Word32 Iu5)`
`Rdd=vasrw(Rss,#u5)`	`Word64 Q6_P_vasrw_PI(Word64 Rss, Word32 Iu5)`
`Rdd=vlsrw(Rss,#u5)`	`Word64 Q6_P_vlsrw_PI(Word64 Rss, Word32 Iu5)`

Encoding

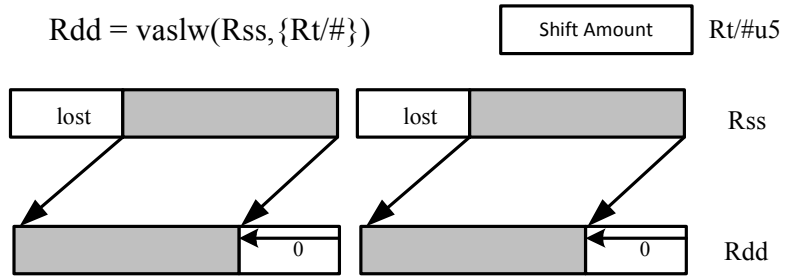
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse									MinOp			d5					
1	0	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	0	**d**	**d**	**d**	**d**	**d**	Rdd=vasrw(Rss,#u5)
1	0	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	0	1	**d**	**d**	**d**	**d**	**d**	Rdd=vlsrw(Rss,#u5)
1	0	0	0	0	0	0	0	0	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rdd=vaslw(Rss,#u5)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d

Field name	Description
s5	Field to encode register s
MajOp	Major Opcode
MinOp	Minor Opcode
RegType	Register Type

Vector shift words by register

The shift amount is the least significant 7 bits of Rt, treated as a two's complement value. If the shift amount is negative, the direction of the shift is reversed. Shift the source values right or left based on the shift amount and the type of instruction. Arithmetic right shifts place the sign bit of the source value in the vacated positions. Logical right shifts place zeros in the vacated positions.



Syntax	Behavior
`Rdd=vaslw(Rss,Rt)`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt7->32(Rt)>0)?(sxt32->64(Rss.w[i])<<sxt7->32(Rt)):(sxt32->64(Rss.w[i])>>sxt7->32(Rt)); };`
`Rdd=vasrw(Rss,Rt)`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt7->32(Rt)>0)?(sxt32->64(Rss.w[i])>>sxt7->32(Rt)):(sxt32->64(Rss.w[i])<<sxt7->32(Rt)); };`
`Rdd=vlslw(Rss,Rt)`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt7->32(Rt)>0)?(zxt32->64(Rss.uw[i])<<sxt7->32(Rt)):(zxt32->64(Rss.uw[i])>>>sxt7->32(Rt)); };`
`Rdd=vlsrw(Rss,Rt)`	`for (i=0;i<2;i++) { Rdd.w[i]=(sxt7->32(Rt)>0)?(zxt32->64(Rss.uw[i])>>>sxt7->32(Rt)):(zxt32->64(Rss.uw[i])<<sxt7->32(Rt)); };`

Class: XTYPE (slots 2,3)

Notes

- If the number of bits to be shifted is greater than the width of the vector element, the result is either all sign-bits (for arithmetic right shifts) or all zeros for logical and left shifts.

Intrinsics

`Rdd=vaslw(Rss,Rt)`	`Word64 Q6_P_vaslw_PR(Word64 Rss, Word32 Rt)`
`Rdd=vasrw(Rss,Rt)`	`Word64 Q6_P_vasrw_PR(Word64 Rss, Word32 Rt)`
`Rdd=vlslw(Rss,Rt)`	`Word64 Q6_P_vlslw_PR(Word64 Rss, Word32 Rt)`
`Rdd=vlsrw(Rss,Rt)`	`Word64 Q6_P_vlsrw_PR(Word64 Rss, Word32 Rt)`

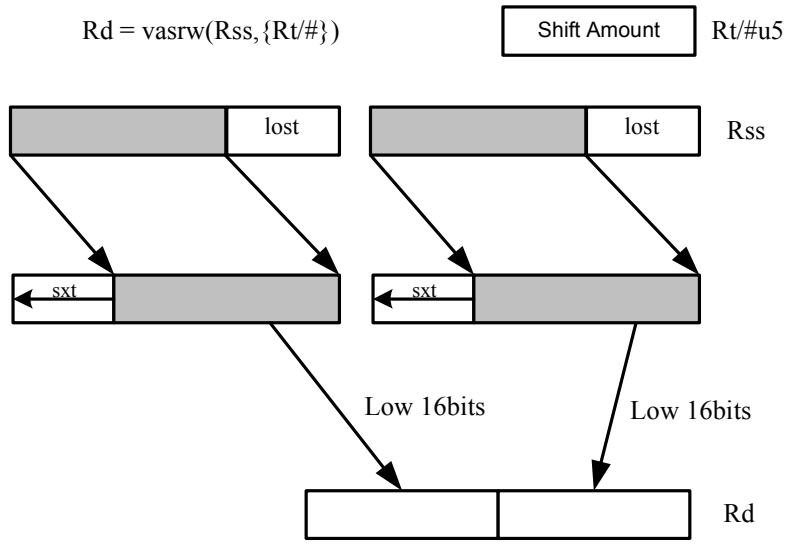
Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				Maj			s5					Parse			t5					Min			d5					
1	1	0	0	0	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vasrw(Rss,Rt)
1	1	0	0	0	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vlsrw(Rss,Rt)
1	1	0	0	0	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	0	-	**d**	**d**	**d**	**d**	**d**	Rdd=vaslw(Rss,Rt)
1	1	0	0	0	0	1	1	0	0	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	1	1	-	**d**	**d**	**d**	**d**	**d**	Rdd=vlslw(Rss,Rt)

Field name	Description
`ICLASS`	Instruction Class
`Parse`	Packet/Loop parse bits
`d5`	Field to encode register d
`s5`	Field to encode register s
`t5`	Field to encode register t
`Maj`	Major Opcode
`Min`	Minor Opcode
`RegType`	Register Type

Vector shift words with truncate and pack

Shift individual words of the source vector Rss right by a register or immediate amount. The low 16-bits of each word are packed into destination register Rd.



Syntax	Behavior

`Rd=vasrw(Rss,#u5)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=(Rss.w[i]>>#u).h[0];`
	`};`

`Rd=vasrw(Rss,Rt)`	`for (i=0;i<2;i++) {`
	`  Rd.h[i]=(sxt`$_{7->32}$`(Rt)>0)?(sxt`$_{32->64}$`(Rss.w[i])>>sxt`$_{7->32}$`(Rt)):(sxt`$_{32->64}$`(Rss.w[i])<<sxt`$_{7->32}$`(Rt)).h[0];`
	`};`

Class: XTYPE (slots 2,3)

Intrinsics

`Rd=vasrw(Rss,#u5)`	`Word32 Q6_R_vasrw_PI(Word64 Rss, Word32 Iu5)`
`Rd=vasrw(Rss,Rt)`	`Word32 Q6_R_vasrw_PR(Word64 Rss, Word32 Rt)`

Encoding

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
ICLASS				RegType				MajOp			s5					Parse								MinOp			d5					
1	0	0	0	1	0	0	0	1	1	0	**s**	**s**	**s**	**s**	**s**	**P**	**P**	0	**i**	**i**	**i**	**i**	**i**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=vasrw(Rss,#u5)
ICLASS				RegType							s5					Parse			t5					Min			d5					
1	1	0	0	0	1	0	1	-	-	-	**s**	**s**	**s**	**s**	**s**	**P**	**P**	-	**t**	**t**	**t**	**t**	**t**	0	1	0	**d**	**d**	**d**	**d**	**d**	Rd=vasrw(Rss,Rt)

Field name	Description
ICLASS	Instruction Class
Parse	Packet/Loop parse bits
d5	Field to encode register d
s5	Field to encode register s
t5	Field to encode register t
MajOp	Major Opcode
MinOp	Minor Opcode
Min	Minor Opcode
RegType	Register Type
RegType	Register Type

Instruction Index

A

abs
- Rd=abs(Rs)[:sat] 366
- Rdd=abs(Rss) 365

add
- if ([!]Pu[.new]) Rd=add(Rs,#s8) 208
- if ([!]Pu[.new]) Rd=add(Rs,Rt) 208
- Rd=add(#u6,mpyi(Rs,#U6)) 516
- Rd=add(#u6,mpyi(Rs,Rt)) 516
- Rd=add(Rs,#s16) 183
- Rd=add(Rs,add(Ru,#s6)) 367
- Rd=add(Rs,Rt) 183
- Rd=add(Rs,Rt):sat 183
- Rd=add(Rs,Rt):sat:deprecated 369
- Rd=add(Rt.[HL],Rs.[HL])[:sat]:<<16 371
- Rd=add(Rt.L,Rs.[HL])[:sat] 371
- Rd=add(Ru,mpyi(#u6:2,Rs)) 516
- Rd=add(Ru,mpyi(Rs,#u6)) 516
- Rdd=add(Rs,Rtt) 369
- Rdd=add(Rss,Rtt,Px):carry 374
- Rdd=add(Rss,Rtt) 369
- Rdd=add(Rss,Rtt):raw:hi 369
- Rdd=add(Rss,Rtt):raw:lo 369
- Rdd=add(Rss,Rtt):sat 369
- Rx+=add(Rs,#s8) 367
- Rx+=add(Rs,Rt) 367
- Rx-=add(Rs,#s8) 367
- Rx-=add(Rs,Rt) 367
- Ry=add(Ru,mpyi(Ry,Rs)) 517

addasl
- Rd=addasl(Rt,Rs,#u3) 626

all8
- Pd=all8(Ps) 227

allocframe
- allocframe(#u11:3) 345

and
- if ([!]Pu[.new]) Rd=and(Rs,Rt) 213
- Pd=and(Ps,and(Pt,[!]Pu)) 233
- Pd=and(Pt,[!]Ps) 233
- Rd=and(Rs,#s10) 185
- Rd=and(Rs,Rt) 185
- Rd=and(Rt,~Rs) 185
- Rdd=and(Rss,Rtt) 375
- Rdd=and(Rtt,~Rss) 375
- Rx[&|^]=and(Rs,~Rt) 378
- Rx[&|^]=and(Rs,Rt) 378
- Rx|=and(Rs,#s10) 378

any8
- Pd=any8(Ps) 227

asl
- Rd=asl(Rs,#u5) 621
- Rd=asl(Rs,#u5):sat 633
- Rd=asl(Rs,Rt) 634
- Rd=asl(Rs,Rt):sat 644
- Rdd=asl(Rss,#u6) 621
- Rdd=asl(Rss,Rt) 635
- Rx^=asl(Rs,#u5) 627
- Rx[&|]=asl(Rs,#u5) 627
- Rx[&|]=asl(Rs,Rt) 641
- Rx[+-]=asl(Rs,#u5) 623
- Rx[+-]=asl(Rs,Rt) 637
- Rx=add(#u8,asl(Rx,#U5)) 623
- Rx=and(#u8,asl(Rx,#U5)) 627
- Rx=or(#u8,asl(Rx,#U5)) 627
- Rx=sub(#u8,asl(Rx,#U5)) 623
- Rxx^=asl(Rss,#u6) 628
- Rxx^=asl(Rss,Rt) 641
- Rxx[&|]=asl(Rss,#u6) 627
- Rxx[&|]=asl(Rss,Rt) 641
- Rxx[+-]=asl(Rss,#u6) 623
- Rxx[+-]=asl(Rss,Rt) 638

aslh
- if ([!]Pu[.new]) Rd=aslh(Rs) 210
- Rd=aslh(Rs) 204

asr
- Rd=asr(Rs,#u5) 621
- Rd=asr(Rs,#u5):rnd 631
- Rd=asr(Rs,Rt) 634
- Rd=asr(Rs,Rt):sat 644
- Rdd=asr(Rss,#u6) 621
- Rdd=asr(Rss,#u6):rnd 631
- Rdd=asr(Rss,Rt) 635
- Rx[&|]=asr(Rs,#u5) 627
- Rx[&|]=asr(Rs,Rt) 641
- Rx[+-]=asr(Rs,#u5) 623
- Rx[+-]=asr(Rs,Rt) 637
- Rxx^=asr(Rss,Rt) 641
- Rxx[&|]=asr(Rss,#u6) 627
- Rxx[&|]=asr(Rss,Rt) 641
- Rxx[+-]=asr(Rss,#u6) 623
- Rxx[+-]=asr(Rss,Rt) 638

asrh
- if ([!]Pu[.new]) Rd=asrh(Rs) 210
- Rd=asrh(Rs) 204

asrrnd
- Rd=asrrnd(Rs,#u5) 631
- Rdd=asrrnd(Rss,#u6) 631

B

barrier
- barrier 351

bitsclr
Pd=[!]bitsclr(Rs,#u6) 606
Pd=[!]bitsclr(Rs,Rt) 606

bitsplit
Rdd=bitsplit(Rs,#u5) 455
Rdd=bitsplit(Rs,Rt) 455

bitsset
Pd=[!]bitsset(Rs,Rt) 606

boundscheck
Pd=boundscheck(Rs,Rtt) 600
Pd=boundscheck(Rss,Rtt):raw:hi 600
Pd=boundscheck(Rss,Rtt):raw:lo 600

brev
Rd=brev(Rs) 452
Rdd=brev(Rss) 452

brkpt
brkpt 352

C

call
call #r22:2 241
if ([!]Pu) call #r15:2 241

callr
callr Rs 237
if ([!]Pu) callr Rs 237

cl0
Rd=cl0(Rs) 439
Rd=cl0(Rss) 439

cl1
Rd=cl1(Rs) 439
Rd=cl1(Rss) 439

clb
Rd=add(clb(Rs),#s6) 439
Rd=add(clb(Rss),#s6) 439
Rd=clb(Rs) 439
Rd=clb(Rss) 439

clrbit
memb(Rs+#u6:0)=clrbit(#U5) 298
memh(Rs+#u6:1)=clrbit(#U5) 299
memw(Rs+#u6:2)=clrbit(#U5) 300
Rd=clrbit(Rs,#u5) 453
Rd=clrbit(Rs,Rt) 453

cmp.eq
if ([!]cmp.eq(Ns.new,#-1)) jump:<hint> #r9:2 302
if ([!]cmp.eq(Ns.new,#U5)) jump:<hint> #r9:2 302
if ([!]cmp.eq(Ns.new,Rt)) jump:<hint> #r9:2 302
p[01]=cmp.eq(Rs,#-1) 243
p[01]=cmp.eq(Rs,#U5) 243
p[01]=cmp.eq(Rs,Rt) 243
Pd=[!]cmp.eq(Rs,#s10) 221
Pd=[!]cmp.eq(Rs,Rt) 221
Pd=cmp.eq(Rss,Rtt) 605
Rd=[!]cmp.eq(Rs,#s8) 223
Rd=[!]cmp.eq(Rs,Rt) 223

cmp.ge
Pd=cmp.ge(Rs,#s8) 221

cmp.geu
Pd=cmp.geu(Rs,#u8) 221

cmp.gt
if ([!]cmp.gt(Ns.new,#-1)) jump:<hint> #r9:2 302
if ([!]cmp.gt(Ns.new,#U5)) jump:<hint> #r9:2 302
if ([!]cmp.gt(Ns.new,Rt)) jump:<hint> #r9:2 302
if ([!]cmp.gt(Rt,Ns.new)) jump:<hint> #r9:2 303
p[01]=cmp.gt(Rs,#-1) 243
p[01]=cmp.gt(Rs,#U5) 244
p[01]=cmp.gt(Rs,Rt) 244
Pd=[!]cmp.gt(Rs,#s10) 221
Pd=[!]cmp.gt(Rs,Rt) 221
Pd=cmp.gt(Rss,Rtt) 605

cmp.gtu
if ([!]cmp.gtu(Ns.new,#U5)) jump:<hint> #r9:2 303
if ([!]cmp.gtu(Ns.new,Rt)) jump:<hint> #r9:2 303
if ([!]cmp.gtu(Rt,Ns.new)) jump:<hint> #r9:2 303
p[01]=cmp.gtu(Rs,#U5) 244
p[01]=cmp.gtu(Rs,Rt) 244
Pd=[!]cmp.gtu(Rs,#u9) 221
Pd=[!]cmp.gtu(Rs,Rt) 221
Pd=cmp.gtu(Rss,Rtt) 605

cmp.lt
Pd=cmp.lt(Rs,Rt) 221

cmp.ltu
Pd=cmp.ltu(Rs,Rt) 221

cmpb.eq
Pd=cmpb.eq(Rs,#u8) 601
Pd=cmpb.eq(Rs,Rt) 601

cmpb.gt
Pd=cmpb.gt(Rs,#s8) 601
Pd=cmpb.gt(Rs,Rt) 601

cmpb.gtu
Pd=cmpb.gtu(Rs,#u7) 601
Pd=cmpb.gtu(Rs,Rt) 601

cmph.eq
Pd=cmph.eq(Rs,#s8) 603
Pd=cmph.eq(Rs,Rt) 603

cmph.gt
Pd=cmph.gt(Rs,#s8) 603
Pd=cmph.gt(Rs,Rt) 603

cmph.gtu
Pd=cmph.gtu(Rs,#u7) 603
Pd=cmph.gtu(Rs,Rt) 603

cmpy
Rd=cmpy(Rs,Rt)[:<<1]:rnd:sat 471
Rd=cmpy(Rs,Rt*)[:<<1]:rnd:sat 471
Rdd=cmpy(Rs,Rt)[:<<1]:sat 466
Rdd=cmpy(Rs,Rt*)[:<<1]:sat 467
Rxx+=cmpy(Rs,Rt)[:<<1]:sat 467
Rxx+=cmpy(Rs,Rt*)[:<<1]:sat 467
Rxx-=cmpy(Rs,Rt)[:<<1]:sat 467
Rxx-=cmpy(Rs,Rt*)[:<<1]:sat 467

cmpyi
Rdd=cmpyi(Rs,Rt) 469
Rxx+=cmpyi(Rs,Rt) 469

cmpyiwh
Rd=cmpyiwh(Rss,Rt):<<1:rnd:sat 473
Rd=cmpyiwh(Rss,Rt*):<<1:rnd:sat 473

cmpyr
Rdd=cmpyr(Rs,Rt) 469
Rxx+=cmpyr(Rs,Rt) 469

cmpyrwh
Rd=cmpyrwh(Rss,Rt):<<1:rnd:sat 473
Rd=cmpyrwh(Rss,Rt*):<<1:rnd:sat 473

combine
if ([!]Pu[.new]) Rdd=combine(Rs,Rt) 212
Rd=combine(Rt.[HL],Rs.[HL]) 200
Rdd=combine(#s8,#S8) 200
Rdd=combine(#s8,#U6) 200
Rdd=combine(#s8,Rs) 200
Rdd=combine(Rs,#s8) 200
Rdd=combine(Rs,Rt) 200

convert_d2df
Rdd=convert_d2df(Rss) 500

convert_d2sf
Rd=convert_d2sf(Rss) 500

convert_df2d
Rdd=convert_df2d(Rss) 502
Rdd=convert_df2d(Rss):chop 502

convert_df2sf
Rd=convert_df2sf(Rss) 499

convert_df2ud
Rdd=convert_df2ud(Rss) 502
Rdd=convert_df2ud(Rss):chop 502

convert_df2uw
Rd=convert_df2uw(Rss) 502
Rd=convert_df2uw(Rss):chop 502

convert_df2w
Rd=convert_df2w(Rss) 502
Rd=convert_df2w(Rss):chop 502

convert_sf2d
Rdd=convert_sf2d(Rs) 502
Rdd=convert_sf2d(Rs):chop 502

convert_sf2df
Rdd=convert_sf2df(Rs) 499

convert_sf2ud
Rdd=convert_sf2ud(Rs) 502
Rdd=convert_sf2ud(Rs):chop 502

convert_sf2uw
Rd=convert_sf2uw(Rs) 502
Rd=convert_sf2uw(Rs):chop 502

convert_sf2w
Rd=convert_sf2w(Rs) 502
Rd=convert_sf2w(Rs):chop 502

convert_ud2df
Rdd=convert_ud2df(Rss) 500

convert_ud2sf
Rd=convert_ud2sf(Rss) 500

convert_uw2df
Rdd=convert_uw2df(Rs) 500

convert_uw2sf
Rd=convert_uw2sf(Rs) 500

convert_w2df
Rdd=convert_w2df(Rs) 500

convert_w2sf
Rd=convert_w2sf(Rs) 500

cround
Rd=cround(Rs,#u5) 386
Rd=cround(Rs,Rt) 386

ct0
Rd=ct0(Rs) 442
Rd=ct0(Rss) 442

ct1
Rd=ct1(Rs) 442
Rd=ct1(Rss) 442

D

dccleana
dccleana(Rs) 354

dccleaninva
dccleaninva(Rs) 354

dcfetch
dcfetch(Rs) 353
dcfetch(Rs+#u11:3) 353

dcinva
dcinva(Rs) 354

dczeroa
dczeroa(Rs) 350

dealloc_return
dealloc_return 286
if ([!]Ps) dealloc_return 286
nt
if ([!]Ps.new) dealloc_return:nt 286
t
if ([!]Ps.new) dealloc_return:t 286

deallocframe
deallocframe 284

decbin
Rdd=decbin(Rss,Rtt) 574

deinterleave
Rdd=deinterleave(Rss) 449

dfclass
Pd=dfclass(Rss,#u5) 496

dfcmp.eq
Pd=dfcmp.eq(Rss,Rtt) 497

dfcmp.ge
Pd=dfcmp.ge(Rss,Rtt) 497

dfcmp.gt
Pd=dfcmp.gt(Rss,Rtt) 497

dfcmp.uo
Pd=dfcmp.uo(Rss,Rtt) 497

dfmake
Rdd=dfmake(#u10):neg 510
Rdd=dfmake(#u10):pos 510

E

encbin
Rdd=encbin(Rss,Rtt,Pu) 575

endloop0
endloop0 225

endloop01
endloop01 225

endloop1
endloop1 225

ext
ext-nop 188

extract
Rd=extract(Rs,#u5,#U5) 444
Rd=extract(Rs,Rtt) 444
Rdd=extract(Rss,#u6,#U6) 444
Rdd=extract(Rss,Rtt) 444

extractu
Rd=extractu(Rs,#u5,#U5) 444
Rd=extractu(Rs,Rtt) 444
Rdd=extractu(Rss,#u6,#U6) 444
Rdd=extractu(Rss,Rtt) 444

F

fastcorner9
Pd=[!]fastcorner9(Ps,Pt) 226

H

hintjr
hintjr(Rs) 238

I

icinva
icinva(Rs) 355

if ([!]p[01].new) jump:<hint> #r9:2 243, 243, 243, 243, 244, 244, 244, 244, 244

insert
Rx=insert(Rs,#u5,#U5) 447
Rx=insert(Rs,Rtt) 447
Rxx=insert(Rss,#u6,#U6) 447
Rxx=insert(Rss,Rtt) 447

interleave
Rdd=interleave(Rss) 449

isync
isync 356

J

jump
if ([!]Pu.new) jump:<hint> #r15:2 250
if ([!]Pu) jump #r15:2 248
if ([!]Pu) jump:<hint> #r15:2 248
jump #r22:2 248
nt
if (Rs!=#0) jump:nt #r13:2 251
if (Rs<=#0) jump:nt #r13:2 251
if (Rs==#0) jump:nt #r13:2 251
if (Rs>=#0) jump:nt #r13:2 251
Rd=#U6 253
Rd=Rs 253
t
if (Rs!=#0) jump:t #r13:2 251
if (Rs<=#0) jump:t #r13:2 251
if (Rs==#0) jump:t #r13:2 251
if (Rs>=#0) jump:t #r13:2 251

jump #r9:2 253, 253

jumpr
if ([!]Pu) jumpr Rs 239
if ([!]Pu[.new]) jumpr:<hint> Rs 239
jumpr Rs 239

L

l2fetch
l2fetch(Rs,Rt) 358
l2fetch(Rs,Rtt) 358

lfs
Rdd=lfs(Rss,Rtt) 450

loop0
loop0(#r7:2,#U10) 228
loop0(#r7:2,Rs) 228

loop1
loop1(#r7:2,#U10) 228
loop1(#r7:2,Rs) 228

lsl
Rd=lsl(#s6,Rt) 634
Rd=lsl(Rs,Rt) 634
Rdd=lsl(Rss,Rt) 635
Rx[&|]=lsl(Rs,Rt) 641
Rx[+-]=lsl(Rs,Rt) 637
Rxx^=lsl(Rss,Rt) 641
Rxx[&|]=lsl(Rss,Rt) 641
Rxx[+-]=lsl(Rss,Rt) 638

lsr
Rd=lsr(Rs,#u5) 621
Rd=lsr(Rs,Rt) 635
Rdd=lsr(Rss,#u6) 621
Rdd=lsr(Rss,Rt) 635
Rx^=lsr(Rs,#u5) 627
Rx[&|]=lsr(Rs,#u5) 627
Rx[&|]=lsr(Rs,Rt) 641
Rx[+-]=lsr(Rs,#u5) 623
Rx[+-]=lsr(Rs,Rt) 638
Rx=add(#u8,lsr(Rx,#U5)) 623
Rx=and(#u8,lsr(Rx,#U5)) 627
Rx=or(#u8,lsr(Rx,#U5)) 627
Rx=sub(#u8,lsr(Rx,#U5)) 623
Rxx^=lsr(Rss,#u6) 628
Rxx^=lsr(Rss,Rt) 641
Rxx[&|]=lsr(Rss,#u6) 628
Rxx[&|]=lsr(Rss,Rt) 641
Rxx[+-]=lsr(Rss,#u6) 623
Rxx[+-]=lsr(Rss,Rt) 638

M

mask
Rdd=mask(Pt) 607

max
Rd=max(Rs,Rt) 380
Rdd=max(Rss,Rtt) 381

maxu
Rd=maxu(Rs,Rt) 380
Rdd=maxu(Rss,Rtt) 381

memb
if ([!]Pt[.new]) Rd=memb(#u6) 260
if ([!]Pt[.new]) Rd=memb(Rs+#u6:0) 260
if ([!]Pt[.new]) Rd=memb(Rx++#s4:0) 260
if ([!]Pv[.new]) memb(#u6)=Nt.new 309
if ([!]Pv[.new]) memb(#u6)=Rt 330
if ([!]Pv[.new]) memb(Rs+#u6:0)=#S6 330
if ([!]Pv[.new]) memb(Rs+#u6:0)=Nt.new 309
if ([!]Pv[.new]) memb(Rs+#u6:0)=Rt 330
if ([!]Pv[.new]) memb(Rs+Ru<<#u2)=Nt.new 309
if ([!]Pv[.new]) memb(Rs+Ru<<#u2)=Rt 330
if ([!]Pv[.new]) memb(Rx++#s4:0)=Nt.new 309
if ([!]Pv[.new]) memb(Rx++#s4:0)=Rt 330
if ([!]Pv[.new]) Rd=memb(Rs+Rt<<#u2) 260
memb(gp+#u16:0)=Nt.new 306
memb(gp+#u16:0)=Rt 328
memb(Re=#U6)=Nt.new 306
memb(Re=#U6)=Rt 328
memb(Rs+#s11:0)=Nt.new 306
memb(Rs+#s11:0)=Rt 328
memb(Rs+#u6:0)[+-]=#U5 298
memb(Rs+#u6:0)[+-|&]=Rt 298
memb(Rs+#u6:0)=#S8 328
memb(Rs+Ru<<#u2)=Nt.new 306
memb(Rs+Ru<<#u2)=Rt 328
memb(Ru<<#u2+#U6)=Nt.new 306
memb(Ru<<#u2+#U6)=Rt 328
memb(Rx++#s4:0:circ(Mu))=Nt.new 306
memb(Rx++#s4:0:circ(Mu))=Rt 328
memb(Rx++#s4:0)=Nt.new 306
memb(Rx++#s4:0)=Rt 328
memb(Rx++I:circ(Mu))=Nt.new 306
memb(Rx++I:circ(Mu))=Rt 328
memb(Rx++Mu:brev)=Nt.new 306
memb(Rx++Mu:brev)=Rt 328
memb(Rx++Mu)=Nt.new 306
memb(Rx++Mu)=Rt 328
Rd=memb(gp+#u16:0) 258
Rd=memb(Re=#U6) 258
Rd=memb(Rs+#s11:0) 258
Rd=memb(Rs+Rt<<#u2) 258
Rd=memb(Rt<<#u2+#U6) 258
Rd=memb(Rx++#s4:0:circ(Mu)) 258
Rd=memb(Rx++#s4:0) 258
Rd=memb(Rx++I:circ(Mu)) 258
Rd=memb(Rx++Mu:brev) 258
Rd=memb(Rx++Mu) 258

memb_fifo
Ryy=memb_fifo(Re=#U6) 262
Ryy=memb_fifo(Rs) 262
Ryy=memb_fifo(Rs+#s11:0) 262
Ryy=memb_fifo(Rt<<#u2+#U6) 262
Ryy=memb_fifo(Rx++#s4:0:circ(Mu)) 263
Ryy=memb_fifo(Rx++#s4:0) 263
Ryy=memb_fifo(Rx++I:circ(Mu)) 263
Ryy=memb_fifo(Rx++Mu:brev) 263
Ryy=memb_fifo(Rx++Mu) 263

membh
- Rd=membh(Re=#U6) 290
- Rd=membh(Rs) 290
- Rd=membh(Rs+#s11:1) 290
- Rd=membh(Rt<<#u2+#U6) 290
- Rd=membh(Rx++#s4:1:circ(Mu)) 290
- Rd=membh(Rx++#s4:1) 290
- Rd=membh(Rx++I:circ(Mu)) 290
- Rd=membh(Rx++Mu:brev) 291
- Rd=membh(Rx++Mu) 291
- Rdd=membh(Re=#U6) 292
- Rdd=membh(Rs) 292
- Rdd=membh(Rs+#s11:2) 292
- Rdd=membh(Rt<<#u2+#U6) 293
- Rdd=membh(Rx++#s4:2:circ(Mu)) 293
- Rdd=membh(Rx++#s4:2) 293
- Rdd=membh(Rx++I:circ(Mu)) 293
- Rdd=membh(Rx++Mu:brev) 293
- Rdd=membh(Rx++Mu) 293

memd
- if ([!]Pt[.new]) Rdd=memd(#u6) 256
- if ([!]Pt[.new]) Rdd=memd(Rs+#u6:3) 256
- if ([!]Pt[.new]) Rdd=memd(Rx++#s4:3) 256
- if ([!]Pv[.new]) memd(#u6)=Rtt 326
- if ([!]Pv[.new]) memd(Rs+#u6:3)=Rtt 326
- if ([!]Pv[.new]) memd(Rs+Ru<<#u2)=Rtt 326
- if ([!]Pv[.new]) memd(Rx++#s4:3)=Rtt 326
- if ([!]Pv[.new]) Rdd=memd(Rs+Rt<<#u2) 256
- memd(gp+#u16:3)=Rtt 324
- memd(Re=#U6)=Rtt 324
- memd(Rs+#s11:3)=Rtt 324
- memd(Rs+Ru<<#u2)=Rtt 324
- memd(Ru<<#u2+#U6)=Rtt 324
- memd(Rx++#s4:3:circ(Mu))=Rtt 324
- memd(Rx++#s4:3)=Rtt 324
- memd(Rx++I:circ(Mu))=Rtt 324
- memd(Rx++Mu:brev)=Rtt 324
- memd(Rx++Mu)=Rtt 324
- Rdd=memd(gp+#u16:3) 254
- Rdd=memd(Re=#U6) 254
- Rdd=memd(Rs+#s11:3) 254
- Rdd=memd(Rs+Rt<<#u2) 254
- Rdd=memd(Rt<<#u2+#U6) 254
- Rdd=memd(Rx++#s4:3:circ(Mu)) 254
- Rdd=memd(Rx++#s4:3) 254
- Rdd=memd(Rx++I:circ(Mu)) 254
- Rdd=memd(Rx++Mu:brev) 254
- Rdd=memd(Rx++Mu) 254

memd_locked
- memd_locked(Rs,Pd)=Rtt 348
- Rdd=memd_locked(Rs) 347

memh
- if ([!]Pt[.new]) Rd=memh(#u6) 270
- if ([!]Pt[.new]) Rd=memh(Rs+#u6:1) 270
- if ([!]Pt[.new]) Rd=memh(Rx++#s4:1) 270
- if ([!]Pv[.new]) memh(#u6)=Nt.new 315
- if ([!]Pv[.new]) memh(#u6)=Rt 336
- if ([!]Pv[.new]) memh(#u6)=Rt.H 336
- if ([!]Pv[.new]) memh(Rs+#u6:1)=#S6 336
- if ([!]Pv[.new]) memh(Rs+#u6:1)=Nt.new 315
- if ([!]Pv[.new]) memh(Rs+#u6:1)=Rt 337
- if ([!]Pv[.new]) memh(Rs+#u6:1)=Rt.H 336
- if ([!]Pv[.new]) memh(Rs+Ru<<#u2)=Nt.new 315
- if ([!]Pv[.new]) memh(Rs+Ru<<#u2)=Rt 337
- if ([!]Pv[.new]) memh(Rs+Ru<<#u2)=Rt.H 337
- if ([!]Pv[.new]) memh(Rx++#s4:1)=Nt.new 315
- if ([!]Pv[.new]) memh(Rx++#s4:1)=Rt 337
- if ([!]Pv[.new]) memh(Rx++#s4:1)=Rt.H 337
- if ([!]Pv[.new]) Rd=memh(Rs+Rt<<#u2) 270
- memh(gp+#u16:1)=Nt.new 312
- memh(gp+#u16:1)=Rt 334
- memh(gp+#u16:1)=Rt.H 334
- memh(Re=#U6)=Nt.new 312
- memh(Re=#U6)=Rt 333
- memh(Re=#U6)=Rt.H 333
- memh(Rs+#s11:1)=Nt.new 312
- memh(Rs+#s11:1)=Rt 333
- memh(Rs+#s11:1)=Rt.H 333
- memh(Rs+#u6:1)[+-]=#U5 299
- memh(Rs+#u6:1)[+-|&]=Rt 299
- memh(Rs+#u6:1)=#S8 333
- memh(Rs+Ru<<#u2)=Nt.new 312
- memh(Rs+Ru<<#u2)=Rt 333
- memh(Rs+Ru<<#u2)=Rt.H 333
- memh(Ru<<#u2+#U6)=Nt.new 312
- memh(Ru<<#u2+#U6)=Rt 333
- memh(Ru<<#u2+#U6)=Rt.H 333
- memh(Rx++#s4:1:circ(Mu))=Nt.new 312
- memh(Rx++#s4:1:circ(Mu))=Rt 333
- memh(Rx++#s4:1:circ(Mu))=Rt.H 333
- memh(Rx++#s4:1)=Nt.new 312
- memh(Rx++#s4:1)=Rt 333
- memh(Rx++#s4:1)=Rt.H 333
- memh(Rx++I:circ(Mu))=Nt.new 312
- memh(Rx++I:circ(Mu))=Rt 333
- memh(Rx++I:circ(Mu))=Rt.H 333
- memh(Rx++Mu:brev)=Nt.new 312
- memh(Rx++Mu:brev)=Rt 334
- memh(Rx++Mu:brev)=Rt.H 334
- memh(Rx++Mu)=Nt.new 312
- memh(Rx++Mu)=Rt 334
- memh(Rx++Mu)=Rt.H 334
- Rd=memh(gp+#u16:1) 268
- Rd=memh(Re=#U6) 268
- Rd=memh(Rs+#s11:1) 268
- Rd=memh(Rs+Rt<<#u2) 268
- Rd=memh(Rt<<#u2+#U6) 268
- Rd=memh(Rx++#s4:1:circ(Mu)) 268
- Rd=memh(Rx++#s4:1) 268
- Rd=memh(Rx++I:circ(Mu)) 268
- Rd=memh(Rx++Mu:brev) 268
- Rd=memh(Rx++Mu) 268

memh_fifo
Ryy=memh_fifo(Re=#U6) 265
Ryy=memh_fifo(Rs) 265
Ryy=memh_fifo(Rs+#s11:1) 265
Ryy=memh_fifo(Rt<<#u2+#U6) 265
Ryy=memh_fifo(Rx++#s4:1:circ(Mu)) 266
Ryy=memh_fifo(Rx++#s4:1) 266
Ryy=memh_fifo(Rx++I:circ(Mu)) 266
Ryy=memh_fifo(Rx++Mu:brev) 266
Ryy=memh_fifo(Rx++Mu) 266

memub
if ([!]Pt[.new]) Rd=memub(#u6) 274
if ([!]Pt[.new]) Rd=memub(Rs+#u6:0) 274
if ([!]Pt[.new]) Rd=memub(Rx++#s4:0) 274
if ([!]Pv[.new]) Rd=memub(Rs+Rt<<#u2) 274
Rd=memub(gp+#u16:0) 272
Rd=memub(Re=#U6) 272
Rd=memub(Rs+#s11:0) 272
Rd=memub(Rs+Rt<<#u2) 272
Rd=memub(Rt<<#u2+#U6) 272
Rd=memub(Rx++#s4:0:circ(Mu)) 272
Rd=memub(Rx++#s4:0) 272
Rd=memub(Rx++I:circ(Mu)) 272
Rd=memub(Rx++Mu:brev) 272
Rd=memub(Rx++Mu) 272

memubh
Rd=memubh(Re=#U6) 291
Rd=memubh(Rs+#s11:1) 291
Rd=memubh(Rt<<#u2+#U6) 291
Rd=memubh(Rx++#s4:1:circ(Mu)) 292
Rd=memubh(Rx++#s4:1) 291
Rd=memubh(Rx++I:circ(Mu)) 292
Rd=memubh(Rx++Mu:brev) 292
Rd=memubh(Rx++Mu) 292
Rdd=memubh(Re=#U6) 294
Rdd=memubh(Rs+#s11:2) 294
Rdd=memubh(Rt<<#u2+#U6) 294
Rdd=memubh(Rx++#s4:2:circ(Mu)) 295
Rdd=memubh(Rx++#s4:2) 295
Rdd=memubh(Rx++I:circ(Mu)) 295
Rdd=memubh(Rx++Mu:brev) 295
Rdd=memubh(Rx++Mu) 295

memuh
if ([!]Pt[.new]) Rd=memuh(#u6) 278
if ([!]Pt[.new]) Rd=memuh(Rs+#u6:1) 278
if ([!]Pt[.new]) Rd=memuh(Rx++#s4:1) 278
if ([!]Pv[.new]) Rd=memuh(Rs+Rt<<#u2) 278
Rd=memuh(gp+#u16:1) 276
Rd=memuh(Re=#U6) 276
Rd=memuh(Rs+#s11:1) 276
Rd=memuh(Rs+Rt<<#u2) 276
Rd=memuh(Rt<<#u2+#U6) 276
Rd=memuh(Rx++#s4:1:circ(Mu)) 276
Rd=memuh(Rx++#s4:1) 276
Rd=memuh(Rx++I:circ(Mu)) 276
Rd=memuh(Rx++Mu:brev) 276
Rd=memuh(Rx++Mu) 276

memw
if ([!]Pt[.new]) Rd=memw(#u6) 282
if ([!]Pt[.new]) Rd=memw(Rs+#u6:2) 282
if ([!]Pt[.new]) Rd=memw(Rx++#s4:2) 282
if ([!]Pv[.new]) memw(#u6)=Nt.new 321
if ([!]Pv[.new]) memw(#u6)=Rt 342
if ([!]Pv[.new]) memw(Rs+#u6:2)=#S6 342
if ([!]Pv[.new]) memw(Rs+#u6:2)=Nt.new 321
if ([!]Pv[.new]) memw(Rs+#u6:2)=Rt 342
if ([!]Pv[.new]) memw(Rs+Ru<<#u2)=Nt.new 321
if ([!]Pv[.new]) memw(Rs+Ru<<#u2)=Rt 342
if ([!]Pv[.new]) memw(Rx++#s4:2)=Nt.new 321
if ([!]Pv[.new]) memw(Rx++#s4:2)=Rt 342
if ([!]Pv[.new]) Rd=memw(Rs+Rt<<#u2) 282
memw(gp+#u16:2)=Nt.new 318
memw(gp+#u16:2)=Rt 340
memw(Re=#U6)=Nt.new 318
memw(Re=#U6)=Rt 340
memw(Rs+#s11:2)=Nt.new 318
memw(Rs+#s11:2)=Rt 340
memw(Rs+#u6:2)[+-]=#U5 300
memw(Rs+#u6:2)[+-|&]=Rt 300
memw(Rs+#u6:2)=#S8 340
memw(Rs+Ru<<#u2)=Nt.new 318
memw(Rs+Ru<<#u2)=Rt 340
memw(Ru<<#u2+#U6)=Nt.new 318
memw(Ru<<#u2+#U6)=Rt 340
memw(Rx++#s4:2:circ(Mu))=Nt.new 318
memw(Rx++#s4:2:circ(Mu))=Rt 340
memw(Rx++#s4:2)=Nt.new 318
memw(Rx++#s4:2)=Rt 340
memw(Rx++I:circ(Mu))=Nt.new 318
memw(Rx++I:circ(Mu))=Rt 340
memw(Rx++Mu:brev)=Nt.new 318
memw(Rx++Mu:brev)=Rt 340
memw(Rx++Mu)=Nt.new 318
memw(Rx++Mu)=Rt 340
Rd=memw(gp+#u16:2) 280
Rd=memw(Re=#U6) 280
Rd=memw(Rs+#s11:2) 280
Rd=memw(Rs+Rt<<#u2) 280
Rd=memw(Rt<<#u2+#U6) 280
Rd=memw(Rx++#s4:2:circ(Mu)) 280
Rd=memw(Rx++#s4:2) 280
Rd=memw(Rx++I:circ(Mu)) 280
Rd=memw(Rx++Mu:brev) 280
Rd=memw(Rx++Mu) 280

memw_locked
memw_locked(Rs,Pd)=Rt 348
Rd=memw_locked(Rs) 347

min
Rd=min(Rt,Rs) 382
Rdd=min(Rtt,Rss) 383

minu
Rd=minu(Rt,Rs) 382
Rdd=minu(Rtt,Rss) 383

modwrap
Rd=modwrap(Rs,Rt) 384

mpy
Rd=mpy(Rs,Rt.H):<<1:rnd:sat 545
Rd=mpy(Rs,Rt.H):<<1:sat 545
Rd=mpy(Rs,Rt.L):<<1:rnd:sat 545
Rd=mpy(Rs,Rt.L):<<1:sat 545
Rd=mpy(Rs,Rt) 545
Rd=mpy(Rs,Rt):<<1 545
Rd=mpy(Rs,Rt):<<1:sat 545
Rd=mpy(Rs,Rt):rnd 545
Rd=mpy(Rs.[HL],Rt.[HL])[:<<1][:rnd][:sat] 529
Rdd=mpy(Rs,Rt) 548
Rdd=mpy(Rs.[HL],Rt.[HL])[:<<1][:rnd] 529
Rx+=mpy(Rs,Rt):<<1:sat 545
Rx+=mpy(Rs.[HL],Rt.[HL])[:<<1][:sat] 529
Rx-=mpy(Rs,Rt):<<1:sat 545
Rx-=mpy(Rs.[HL],Rt.[HL])[:<<1][:sat] 529
Rxx[+-]=mpy(Rs,Rt) 548
Rxx+=mpy(Rs.[HL],Rt.[HL])[:<<1] 529
Rxx-=mpy(Rs.[HL],Rt.[HL])[:<<1] 529

mpyi
Rd=+mpyi(Rs,#u8) 516
Rd=mpyi(Rs,#m9) 516
Rd=-mpyi(Rs,#u8) 516
Rd=mpyi(Rs,Rt) 516
Rx+=mpyi(Rs,#u8) 517
Rx+=mpyi(Rs,Rt) 517
Rx-=mpyi(Rs,#u8) 517

mpysu
Rd=mpysu(Rs,Rt) 545

mpyu
Rd=mpyu(Rs,Rt) 545
Rd=mpyu(Rs.[HL],Rt.[HL])[:<<1] 536
Rdd=mpyu(Rs,Rt) 548
Rdd=mpyu(Rs.[HL],Rt.[HL])[:<<1] 536
Rx+=mpyu(Rs.[HL],Rt.[HL])[:<<1] 536
Rx-=mpyu(Rs.[HL],Rt.[HL])[:<<1] 536
Rxx[+-]=mpyu(Rs,Rt) 548
Rxx+=mpyu(Rs.[HL],Rt.[HL])[:<<1] 536
Rxx-=mpyu(Rs.[HL],Rt.[HL])[:<<1] 536

mpyui
Rd=mpyui(Rs,Rt) 517

mux
Rd=mux(Pu,#s8,#S8) 202
Rd=mux(Pu,#s8,Rs) 202
Rd=mux(Pu,Rs,#s8) 202
Rd=mux(Pu,Rs,Rt) 202

N

neg
Rd=neg(Rs) 187
Rd=neg(Rs):sat 385
Rdd=neg(Rss) 385

no mnemonic
Cd=Rs 235
Cdd=Rss 235
if ([!]Pu[.new]) Rd=#s12 218
if ([!]Pu[.new]) Rd=Rs 218
if ([!]Pu[.new]) Rdd=Rss 218
Pd=Ps 233
Pd=Rs 609
Rd=#s16 192
Rd=Cs 235
Rd=Ps 609
Rd=Rs 194
Rdd=#s8 192
Rdd=Css 235
Rdd=Rss 194
Rx.[HL]=#u16 192

nop
nop 188

normamt
Rd=normamt(Rs) 439
Rd=normamt(Rss) 439

not
Pd=not(Ps) 233
Rd=not(Rs) 185
Rdd=not(Rss) 375

O

or
if ([!]Pu[.new]) Rd=or(Rs,Rt) 213
Pd=and(Ps,or(Pt,[!]Pu)) 233
Pd=or(Ps,and(Pt,[!]Pu)) 233
Pd=or(Ps,or(Pt,[!]Pu)) 233
Pd=or(Pt,[!]Ps) 233
Rd=or(Rs,#s10) 185
Rd=or(Rs,Rt) 185
Rd=or(Rt,~Rs) 185
Rdd=or(Rss,Rtt) 375
Rdd=or(Rtt,~Rss) 375
Rx[&|^]=or(Rs,Rt) 378
Rx=or(Ru,and(Rx,#s10)) 378
Rx|=or(Rs,#s10) 378

P

packhl
Rdd=packhl(Rs,Rt) 206

parity
Rd=parity(Rs,Rt) 451
Rd=parity(Rss,Rtt) 451

pause
pause(#u8) 360

pc
Rd=add(pc,#u6) 230

pmpyw
Rdd=pmpyw(Rs,Rt) 541
Rxx^=pmpyw(Rs,Rt) 541

popcount
Rd=popcount(Rss) 441

R

rol
Rd=rol(Rs,#u5) 621
Rdd=rol(Rss,#u6) 621
Rx^=rol(Rs,#u5) 627
Rx[&|]=rol(Rs,#u5) 627
Rx[+-]=rol(Rs,#u5) 623
Rxx^=rol(Rss,#u6) 628
Rxx[&|]=rol(Rss,#u6) 628
Rxx[+-]=rol(Rss,#u6) 623

round
Rd=round(Rs,#u5)[:sat] 386
Rd=round(Rs,Rt)[:sat] 386
Rd=round(Rss):sat 386

S

sat
Rd=sat(Rss) 576

satb
Rd=satb(Rs) 576

sath
Rd=sath(Rs) 576

satub
Rd=satub(Rs) 576

satuh
Rd=satuh(Rs) 576

setbit
memb(Rs+#u6:0)=setbit(#U5) 298
memh(Rs+#u6:1)=setbit(#U5) 299
memw(Rs+#u6:2)=setbit(#U5) 300
Rd=setbit(Rs,#u5) 453
Rd=setbit(Rs,Rt) 453

sfadd
Rd=sfadd(Rs,Rt) 495

sfclass
Pd=sfclass(Rs,#u5) 496

sfcmp.eq
Pd=sfcmp.eq(Rs,Rt) 497

sfcmp.ge
Pd=sfcmp.ge(Rs,Rt) 497

sfcmp.gt
Pd=sfcmp.gt(Rs,Rt) 497

sfcmp.uo
Pd=sfcmp.uo(Rs,Rt) 497

sffixupd
Rd=sffixupd(Rs,Rt) 504

sffixupn
Rd=sffixupn(Rs,Rt) 504

sffixupr
Rd=sffixupr(Rs) 504

sfinvsqrta
Rd,Pe=sfinvsqrta(Rs) 507

sfmake
Rd=sfmake(#u10):neg 510
Rd=sfmake(#u10):pos 510

sfmax
Rd=sfmax(Rs,Rt) 511

sfmin
Rd=sfmin(Rs,Rt) 512

sfmpy
Rd=sfmpy(Rs,Rt) 513
Rx+=sfmpy(Rs,Rt,Pu):scale 506
Rx+=sfmpy(Rs,Rt) 505
Rx+=sfmpy(Rs,Rt):lib 508
Rx-=sfmpy(Rs,Rt) 505
Rx-=sfmpy(Rs,Rt):lib 508

sfrecipa
Rd,Pe=sfrecipa(Rs,Rt) 514

sfsub
Rd=sfsub(Rs,Rt) 515

shuffeb
Rdd=shuffeb(Rss,Rtt) 588

shuffeh
Rdd=shuffeh(Rss,Rtt) 588

shuffob
Rdd=shuffob(Rtt,Rss) 588

shuffoh
Rdd=shuffoh(Rtt,Rss) 588

sp1loop0
p3=sp1loop0(#r7:2,#U10) 231
p3=sp1loop0(#r7:2,Rs) 231

sp2loop0
p3=sp2loop0(#r7:2,#U10) 231
p3=sp2loop0(#r7:2,Rs) 231

sp3loop0
p3=sp3loop0(#r7:2,#U10) 231
p3=sp3loop0(#r7:2,Rs) 231

sub
if ([!]Pu[.new]) Rd=sub(Rt,Rs) 215
Rd=add(Rs,sub(#s6,Ru)) 367
Rd=sub(#s10,Rs) 189
Rd=sub(Rt,Rs) 189
Rd=sub(Rt,Rs):sat 189
Rd=sub(Rt,Rs):sat:deprecated 388
Rd=sub(Rt.[HL],Rs.[HL])[:sat]:<<16 390
Rd=sub(Rt.L,Rs.[HL])[:sat] 390
Rdd=sub(Rss,Rtt,Px):carry 374
Rdd=sub(Rtt,Rss) 388
Rx+=sub(Rt,Rs) 389

swiz
Rd=swiz(Rs) 578

sxtb
if ([!]Pu[.new]) Rd=sxtb(Rs) 216
Rd=sxtb(Rs) 191

sxth
if ([!]Pu[.new]) Rd=sxth(Rs) 216
Rd=sxth(Rs) 191

sxtw
Rdd=sxtw(Rs) 393

syncht
syncht 361

T

tableidxb
Rx=tableidxb(Rs,#u4,#S6):raw 458
Rx=tableidxb(Rs,#u4,#U5) 458

tableidxd
Rx=tableidxd(Rs,#u4,#S6):raw 458
Rx=tableidxd(Rs,#u4,#U5) 458

tableidxh
Rx=tableidxh(Rs,#u4,#S6):raw 458
Rx=tableidxh(Rs,#u4,#U5) 458

tableidxw
Rx=tableidxw(Rs,#u4,#S6):raw 458
Rx=tableidxw(Rs,#u4,#U5) 458

tlbmatch
Pd=tlbmatch(Rss,Rt) 608

togglebit
Rd=togglebit(Rs,#u5) 453
Rd=togglebit(Rs,Rt) 453

trace
trace(Rs) 362

trap0
trap0(#u8) 363

trap1
trap1(#u8) 363

tstbit
if ([!]tstbit(Ns.new,#0)) jump:<hint> #r9:2 303
p[01]=tstbit(Rs,#0) 244
Pd=[!]tstbit(Rs,#u5) 610
Pd=[!]tstbit(Rs,Rt) 610

V

vabsdiffb
Rdd=vabsdiffb(Rtt,Rss) 396

vabsdiffh
Rdd=vabsdiffh(Rtt,Rss) 397

vabsdiffub
Rdd=vabsdiffub(Rtt,Rss) 396

vabsdiffw
Rdd=vabsdiffw(Rtt,Rss) 398

vabsh
Rdd=vabsh(Rss) 394
Rdd=vabsh(Rss):sat 394

vabsw
Rdd=vabsw(Rss) 395
Rdd=vabsw(Rss):sat 395

vacsh
Rxx,Pe=vacsh(Rss,Rtt) 400

vaddb
Rdd=vaddb(Rss,Rtt) 409

vaddh
Rd=vaddh(Rs,Rt)[:sat] 195
Rdd=vaddh(Rss,Rtt)[:sat] 402

vaddhub
Rd=vaddhub(Rss,Rtt):sat 404

vaddub
Rdd=vaddub(Rss,Rtt)[:sat] 409

vadduh
Rd=vadduh(Rs,Rt):sat 195
Rdd=vadduh(Rss,Rtt):sat 402

vaddw
Rdd=vaddw(Rss,Rtt)[:sat] 410

valignb
Rdd=valignb(Rtt,Rss,#u3) 579
Rdd=valignb(Rtt,Rss,Pu) 579

vaslh
Rdd=vaslh(Rss,#u4) 646
Rdd=vaslh(Rss,Rt) 652

vaslw
Rdd=vaslw(Rss,#u5) 654
Rdd=vaslw(Rss,Rt) 656

vasrh
Rdd=vasrh(Rss,#u4) 646
Rdd=vasrh(Rss,#u4):raw 648
Rdd=vasrh(Rss,#u4):rnd 648
Rdd=vasrh(Rss,Rt) 652

vasrhub
Rd=vasrhub(Rss,#u4):raw 650
Rd=vasrhub(Rss,#u4):rnd:sat 650
Rd=vasrhub(Rss,#u4):sat 650

vasrw
Rd=vasrw(Rss,#u5) 658
Rd=vasrw(Rss,Rt) 658
Rdd=vasrw(Rss,#u5) 654
Rdd=vasrw(Rss,Rt) 656

vavgh
Rd=vavgh(Rs,Rt) 196
Rd=vavgh(Rs,Rt):rnd 196
Rdd=vavgh(Rss,Rtt) 411
Rdd=vavgh(Rss,Rtt):crnd 411
Rdd=vavgh(Rss,Rtt):rnd 411

vavgub
Rdd=vavgub(Rss,Rtt) 413
Rdd=vavgub(Rss,Rtt):rnd 413

vavguh
Rdd=vavguh(Rss,Rtt) 411
Rdd=vavguh(Rss,Rtt):rnd 411

vavguw
Rdd=vavguw(Rss,Rtt)[:rnd] 414

vavgw
Rdd=vavgw(Rss,Rtt):crnd 414
Rdd=vavgw(Rss,Rtt)[:rnd] 414

vcmpb.eq
Pd=any8(vcmpb.eq(Rss,Rtt)) 613
Pd=vcmpb.eq(Rss,#u8) 614
Pd=vcmpb.eq(Rss,Rtt) 614

vcmpb.gt
Pd=vcmpb.gt(Rss,#s8) 614
Pd=vcmpb.gt(Rss,Rtt) 614

vcmpb.gtu
Pd=vcmpb.gtu(Rss,#u7) 614
Pd=vcmpb.gtu(Rss,Rtt) 614

vcmph.eq
Pd=vcmph.eq(Rss,#s8) 611
Pd=vcmph.eq(Rss,Rtt) 611

vcmph.gt
Pd=vcmph.gt(Rss,#s8) 611
Pd=vcmph.gt(Rss,Rtt) 611

vcmph.gtu
Pd=vcmph.gtu(Rss,#u7) 611
Pd=vcmph.gtu(Rss,Rtt) 611

vcmpw.eq
Pd=vcmpw.eq(Rss,#s8) 616
Pd=vcmpw.eq(Rss,Rtt) 616

vcmpw.gt
Pd=vcmpw.gt(Rss,#s8) 616
Pd=vcmpw.gt(Rss,Rtt) 616

vcmpw.gtu
Pd=vcmpw.gtu(Rss,#u7) 616
Pd=vcmpw.gtu(Rss,Rtt) 616

vcmpyi
Rdd=vcmpyi(Rss,Rtt)[:<<1]:sat 476
Rxx+=vcmpyi(Rss,Rtt):sat 476

vcmpyr
Rdd=vcmpyr(Rss,Rtt)[:<<1]:sat 476
Rxx+=vcmpyr(Rss,Rtt):sat 476

vcnegh
Rdd=vcnegh(Rss,Rt) 416

vconj
Rdd=vconj(Rss):sat 478

vcrotate
Rdd=vcrotate(Rss,Rt) 479

vdmpy
Rd=vdmpy(Rss,Rtt)[:<<1]:rnd:sat 553
Rdd=vdmpy(Rss,Rtt):<<1:sat 551
Rdd=vdmpy(Rss,Rtt):sat 551
Rxx+=vdmpy(Rss,Rtt):<<1:sat 551
Rxx+=vdmpy(Rss,Rtt):sat 551

vdmpybsu
Rdd=vdmpybsu(Rss,Rtt):sat 557
Rxx+=vdmpybsu(Rss,Rtt):sat 557

vitpack
Rd=vitpack(Ps,Pt) 618

vlslh
Rdd=vlslh(Rss,Rt) 652

vlslw
Rdd=vlslw(Rss,Rt) 656

vlsrh
Rdd=vlsrh(Rss,#u4) 646
Rdd=vlsrh(Rss,Rt) 652

vlsrw
Rdd=vlsrw(Rss,#u5) 654
Rdd=vlsrw(Rss,Rt) 656

vmaxb
Rdd=vmaxb(Rtt,Rss) 418

vmaxh
Rdd=vmaxh(Rtt,Rss) 419

vmaxub
Rdd=vmaxub(Rtt,Rss) 418

vmaxuh
Rdd=vmaxuh(Rtt,Rss) 419

vmaxuw
Rdd=vmaxuw(Rtt,Rss) 424

vmaxw
Rdd=vmaxw(Rtt,Rss) 424

vminb
Rdd=vminb(Rtt,Rss) 425

vminh
Rdd=vminh(Rtt,Rss) 427

vminub
Rdd,Pe=vminub(Rtt,Rss) 425
Rdd=vminub(Rtt,Rss) 425

vminuh
Rdd=vminuh(Rtt,Rss) 427

vminuw
Rdd=vminuw(Rtt,Rss) 432

vminw
Rdd=vminw(Rtt,Rss) 432

vmpybsu
Rdd=vmpybsu(Rs,Rt) 569
Rxx+=vmpybsu(Rs,Rt) 569

vmpybu
Rdd=vmpybu(Rs,Rt) 569
Rxx+=vmpybu(Rs,Rt) 569

vmpyeh
Rdd=vmpyeh(Rss,Rtt):<<1:sat 559
Rdd=vmpyeh(Rss,Rtt):sat 559
Rxx+=vmpyeh(Rss,Rtt) 559
Rxx+=vmpyeh(Rss,Rtt):<<1:sat 559
Rxx+=vmpyeh(Rss,Rtt):sat 559

vmpyh
Rd=vmpyh(Rs,Rt)[:<<1]:rnd:sat 563
Rdd=vmpyh(Rs,Rt)[:<<1]:sat 561
Rxx+=vmpyh(Rs,Rt) 561
Rxx+=vmpyh(Rs,Rt)[:<<1]:sat 561

vmpyhsu
Rdd=vmpyhsu(Rs,Rt)[:<<1]:sat 565
Rxx+=vmpyhsu(Rs,Rt)[:<<1]:sat 565

vmpyweh
Rdd=vmpyweh(Rss,Rtt)[:<<1]:rnd:sat 521
Rdd=vmpyweh(Rss,Rtt)[:<<1]:sat 521
Rxx+=vmpyweh(Rss,Rtt)[:<<1]:rnd:sat 521
Rxx+=vmpyweh(Rss,Rtt)[:<<1]:sat 521

vmpyweuh
Rdd=vmpyweuh(Rss,Rtt)[:<<1]:rnd:sat 526
Rdd=vmpyweuh(Rss,Rtt)[:<<1]:sat 526
Rxx+=vmpyweuh(Rss,Rtt)[:<<1]:rnd:sat 526
Rxx+=vmpyweuh(Rss,Rtt)[:<<1]:sat 526

vmpywoh
Rdd=vmpywoh(Rss,Rtt)[:<<1]:rnd:sat 521
Rdd=vmpywoh(Rss,Rtt)[:<<1]:sat 521
Rxx+=vmpywoh(Rss,Rtt)[:<<1]:rnd:sat 521
Rxx+=vmpywoh(Rss,Rtt)[:<<1]:sat 521

vmpywouh
Rdd=vmpywouh(Rss,Rtt)[:<<1]:rnd:sat 526
Rdd=vmpywouh(Rss,Rtt)[:<<1]:sat 526
Rxx+=vmpywouh(Rss,Rtt)[:<<1]:rnd:sat 526
Rxx+=vmpywouh(Rss,Rtt)[:<<1]:sat 526

vmux
Rdd=vmux(Pu,Rss,Rtt) 619

vnavgh
Rd=vnavgh(Rt,Rs) 196
Rdd=vnavgh(Rtt,Rss) 411
Rdd=vnavgh(Rtt,Rss):crnd:sat 411
Rdd=vnavgh(Rtt,Rss):rnd:sat 411

vnavgw
Rdd=vnavgw(Rtt,Rss) 414
Rdd=vnavgw(Rtt,Rss):crnd:sat 414
Rdd=vnavgw(Rtt,Rss):rnd:sat 414

vpmpyh
Rdd=vpmpyh(Rs,Rt) 572
Rxx^=vpmpyh(Rs,Rt) 572

vraddh
Rd=vraddh(Rss,Rtt) 407

vraddub
Rdd=vraddub(Rss,Rtt) 405
Rxx+=vraddub(Rss,Rtt) 405

vradduh
Rd=vradduh(Rss,Rtt) 407

vrcmpyi
Rdd=vrcmpyi(Rss,Rtt) 481
Rdd=vrcmpyi(Rss,Rtt*) 481
Rxx+=vrcmpyi(Rss,Rtt) 482
Rxx+=vrcmpyi(Rss,Rtt*) 482

vrcmpyr
Rdd=vrcmpyr(Rss,Rtt) 481
Rdd=vrcmpyr(Rss,Rtt*) 482
Rxx+=vrcmpyr(Rss,Rtt) 482
Rxx+=vrcmpyr(Rss,Rtt*) 482

vrcmpys
Rd=vrcmpys(Rss,Rt):<<1:rnd:sat 489
Rd=vrcmpys(Rss,Rtt):<<1:rnd:sat:raw:hi 489
Rd=vrcmpys(Rss,Rtt):<<1:rnd:sat:raw:lo 489
Rdd=vrcmpys(Rss,Rt):<<1:sat 486
Rdd=vrcmpys(Rss,Rtt):<<1:sat:raw:hi 486
Rdd=vrcmpys(Rss,Rtt):<<1:sat:raw:lo 486
Rxx+=vrcmpys(Rss,Rt):<<1:sat 486
Rxx+=vrcmpys(Rss,Rtt):<<1:sat:raw:hi 486
Rxx+=vrcmpys(Rss,Rtt):<<1:sat:raw:lo 486

vrcnegh
Rxx+=vrcnegh(Rss,Rt) 416

vrcrotate
Rdd=vrcrotate(Rss,Rt,#u2) 493
Rxx+=vrcrotate(Rss,Rt,#u2) 493

vrmaxh
Rxx=vrmaxh(Rss,Ru) 420

vrmaxuh
Rxx=vrmaxuh(Rss,Ru) 420

vrmaxuw
Rxx=vrmaxuw(Rss,Ru) 422

vrmaxw
Rxx=vrmaxw(Rss,Ru) 422

vrminh
Rxx=vrminh(Rss,Ru) 428

vrminuh
Rxx=vrminuh(Rss,Ru) 428

vrminuw
Rxx=vrminuw(Rss,Ru) 430

vrminw
Rxx=vrminw(Rss,Ru) 430

vrmpybsu
Rdd=vrmpybsu(Rss,Rtt) 555
Rxx+=vrmpybsu(Rss,Rtt) 555

vrmpybu
Rdd=vrmpybu(Rss,Rtt) 555
Rxx+=vrmpybu(Rss,Rtt) 555

vrmpyh
Rdd=vrmpyh(Rss,Rtt) 567
Rxx+=vrmpyh(Rss,Rtt) 567

vrmpyweh
Rdd=vrmpyweh(Rss,Rtt)[:<<1] 543
Rxx+=vrmpyweh(Rss,Rtt)[:<<1] 543

vrmpywoh
Rdd=vrmpywoh(Rss,Rtt)[:<<1] 543
Rxx+=vrmpywoh(Rss,Rtt)[:<<1] 543

vrndwh
Rd=vrndwh(Rss) 581
Rd=vrndwh(Rss):sat 581

vrsadub
Rdd=vrsadub(Rss,Rtt) 433
Rxx+=vrsadub(Rss,Rtt) 433

vsathb
Rd=vsathb(Rs) 584
Rd=vsathb(Rss) 584
Rdd=vsathb(Rss) 586

vsathub
Rd=vsathub(Rs) 584
Rd=vsathub(Rss) 584
Rdd=vsathub(Rss) 586

vsatwh
Rd=vsatwh(Rss) 584
Rdd=vsatwh(Rss) 586

vsatwuh
Rd=vsatwuh(Rss) 584
Rdd=vsatwuh(Rss) 586

vsplatb
Rd=vsplatb(Rs) 590
Rdd=vsplatb(Rs) 590

vsplath
Rdd=vsplath(Rs) 591

vspliceb
Rdd=vspliceb(Rss,Rtt,#u3) 592
Rdd=vspliceb(Rss,Rtt,Pu) 592

vsubb
Rdd=vsubb(Rss,Rtt) 437

vsubh
Rd=vsubh(Rt,Rs)[:sat] 197
Rdd=vsubh(Rtt,Rss)[:sat] 435

vsubub
Rdd=vsubub(Rtt,Rss)[:sat] 437

vsubuh
Rd=vsubuh(Rt,Rs):sat 197
Rdd=vsubuh(Rtt,Rss):sat 435

vsubw
Rdd=vsubw(Rtt,Rss)[:sat] 438

vsxtbh
Rdd=vsxtbh(Rs) 593

vsxthw
Rdd=vsxthw(Rs) 593

vtrunehb
Rd=vtrunehb(Rss) 596
Rdd=vtrunehb(Rss,Rtt) 596

vtrunewh
Rdd=vtrunewh(Rss,Rtt) 596

vtrunohb
Rd=vtrunohb(Rss) 596
Rdd=vtrunohb(Rss,Rtt) 596

vtrunowh
Rdd=vtrunowh(Rss,Rtt) 596

vxaddsubh
Rdd=vxaddsubh(Rss,Rtt):rnd:>>1:sat 461
Rdd=vxaddsubh(Rss,Rtt):sat 461

vxaddsubw
Rdd=vxaddsubw(Rss,Rtt):sat 463

vxsubaddh
Rdd=vxsubaddh(Rss,Rtt):rnd:>>1:sat 461
Rdd=vxsubaddh(Rss,Rtt):sat 461

vxsubaddw
Rdd=vxsubaddw(Rss,Rtt):sat 463

vzxtbh
Rdd=vzxtbh(Rs) 598

vzxthw
Rdd=vzxthw(Rs) 598

X

xor
if ([!]Pu[.new]) Rd=xor(Rs,Rt) 213
Pd=xor(Ps,Pt) 233
Rd=xor(Rs,Rt) 185
Rdd=xor(Rss,Rtt) 375
Rx[&|^]=xor(Rs,Rt) 378
Rxx^=xor(Rss,Rtt) 377

Z

zxtb
if ([!]Pu[.new]) Rd=zxtb(Rs) 219
Rd=zxtb(Rs) 199

zxth
if ([!]Pu[.new]) Rd=zxth(Rs) 219
Rd=zxth(Rs) 199

Intrinsics Index

A

abs

Rd=abs(Rs)	Word32 Q6_R_abs_R(Word32 Rs)	366
Rd=abs(Rs):sat	Word32 Q6_R_abs_R_sat(Word32 Rs)	366
Rdd=abs(Rss)	Word64 Q6_P_abs_P(Word64 Rss)	365

add

Rd=add(#u6,mpyi(Rs,#U6))	Word32 Q6_R_add_mpyi_IRI(Word32 Iu6, Word32 Rs, Word32 IU6)	517
Rd=add(#u6,mpyi(Rs,Rt))	Word32 Q6_R_add_mpyi_IRR(Word32 Iu6, Word32 Rs, Word32 Rt)	517
Rd=add(Rs,#s16)	Word32 Q6_R_add_RI(Word32 Rs, Word32 Is16)	183
Rd=add(Rs,add(Ru,#s6))	Word32 Q6_R_add_add_RRI(Word32 Rs, Word32 Ru, Word32 Is6)	367
Rd=add(Rs,Rt)	Word32 Q6_R_add_RR(Word32 Rs, Word32 Rt)	183
Rd=add(Rs,Rt):sat	Word32 Q6_R_add_RR_sat(Word32 Rs, Word32 Rt)	183
Rd=add(Rt.H,Rs.H):<<16	Word32 Q6_R_add_RhRh_s16(Word32 Rt, Word32 Rs)	372
Rd=add(Rt.H,Rs.H):sat:<<16	Word32 Q6_R_add_RhRh_sat_s16(Word32 Rt, Word32 Rs)	372
Rd=add(Rt.H,Rs.L):<<16	Word32 Q6_R_add_RhRl_s16(Word32 Rt, Word32 Rs)	372
Rd=add(Rt.H,Rs.L):sat:<<16	Word32 Q6_R_add_RhRl_sat_s16(Word32 Rt, Word32 Rs)	372
Rd=add(Rt.L,Rs.H)	Word32 Q6_R_add_RlRh(Word32 Rt, Word32 Rs)	372
Rd=add(Rt.L,Rs.H):<<16	Word32 Q6_R_add_RlRh_s16(Word32 Rt, Word32 Rs)	372
Rd=add(Rt.L,Rs.H):sat	Word32 Q6_R_add_RlRh_sat(Word32 Rt, Word32 Rs)	372
Rd=add(Rt.L,Rs.H):sat:<<16	Word32 Q6_R_add_RlRh_sat_s16(Word32 Rt, Word32 Rs)	372
Rd=add(Rt.L,Rs.L)	Word32 Q6_R_add_RlRl(Word32 Rt, Word32 Rs)	372
Rd=add(Rt.L,Rs.L):<<16	Word32 Q6_R_add_RlRl_s16(Word32 Rt, Word32 Rs)	372
Rd=add(Rt.L,Rs.L):sat	Word32 Q6_R_add_RlRl_sat(Word32 Rt, Word32 Rs)	372
Rd=add(Rt.L,Rs.L):sat:<<16	Word32 Q6_R_add_RlRl_sat_s16(Word32 Rt, Word32 Rs)	372
Rd=add(Ru,mpyi(#u6:2,Rs))	Word32 Q6_R_add_mpyi_RIR(Word32 Ru, Word32 Iu6_2, Word32 Rs)	517
Rd=add(Ru,mpyi(Rs,#u6))	Word32 Q6_R_add_mpyi_RRI(Word32 Ru, Word32 Rs, Word32 Iu6)	517
Rdd=add(Rs,Rtt)	Word64 Q6_P_add_RP(Word32 Rs, Word64 Rtt)	369
Rdd=add(Rss,Rtt)	Word64 Q6_P_add_PP(Word64 Rss, Word64 Rtt)	369
Rdd=add(Rss,Rtt):sat	Word64 Q6_P_add_PP_sat(Word64 Rss, Word64 Rtt)	369
Rx+=add(Rs,#s8)	Word32 Q6_R_addacc_RI(Word32 Rx, Word32 Rs, Word32 Is8)	367
Rx+=add(Rs,Rt)	Word32 Q6_R_addacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)	367
Rx-=add(Rs,#s8)	Word32 Q6_R_addnac_RI(Word32 Rx, Word32 Rs, Word32 Is8)	367
Rx-=add(Rs,Rt)	Word32 Q6_R_addnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)	367
Ry=add(Ru,mpyi(Ry,Rs))	Word32 Q6_R_add_mpyi_RRR(Word32 Ru, Word32 Ry, Word32 Rs)	517

addasl

Rd=addasl(Rt,Rs,#u3)	Word32 Q6_R_addasl_RRI(Word32 Rt, Word32 Rs, Word32 Iu3)	626

all8

Pd=all8(Ps)	Byte Q6_p_all8_p(Byte Ps)	227

and

Pd=and(Ps,and(Pt,!Pu))	Byte Q6_p_and_and_ppnp(Byte Ps, Byte Pt, Byte Pu)	233
Pd=and(Ps,and(Pt,Pu))	Byte Q6_p_and_and_ppp(Byte Ps, Byte Pt, Byte Pu)	233
Pd=and(Pt,!Ps)	Byte Q6_p_and_pnp(Byte Pt, Byte Ps)	233
Pd=and(Pt,Ps)	Byte Q6_p_and_pp(Byte Pt, Byte Ps)	233
Rd=and(Rs,#s10)	Word32 Q6_R_and_RI(Word32 Rs, Word32 Is10)	185
Rd=and(Rs,Rt)	Word32 Q6_R_and_RR(Word32 Rs, Word32 Rt)	185
Rd=and(Rt,~Rs)	Word32 Q6_R_and_RnR(Word32 Rt, Word32 Rs)	185
Rdd=and(Rss,Rtt)	Word64 Q6_P_and_PP(Word64 Rss, Word64 Rtt)	375
Rdd=and(Rtt,~Rss)	Word64 Q6_P_and_PnP(Word64 Rtt, Word64 Rss)	375
Rx^=and(Rs,~Rt)	Word32 Q6_R_andxacc_RnR(Word32 Rx, Word32 Rs, Word32 Rt)	378
Rx^=and(Rs,Rt)	Word32 Q6_R_andxacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)	378
Rx&=and(Rs,~Rt)	Word32 Q6_R_andand_RnR(Word32 Rx, Word32 Rs, Word32 Rt)	378
Rx&=and(Rs,Rt)	Word32 Q6_R_andand_RR(Word32 Rx, Word32 Rs, Word32 Rt)	378
Rx|=and(Rs,#s10)	Word32 Q6_R_andor_RI(Word32 Rx, Word32 Rs, Word32 Is10)	378
Rx|=and(Rs,~Rt)	Word32 Q6_R_andor_RnR(Word32 Rx, Word32 Rs, Word32 Rt)	379
Rx|=and(Rs,Rt)	Word32 Q6_R_andor_RR(Word32 Rx, Word32 Rs, Word32 Rt)	379

any8

Pd=any8(Ps)	Byte Q6_p_any8_p(Byte Ps)	227

asl

Rd=asl(Rs,#u5)	Word32 Q6_R_asl_RI(Word32 Rs, Word32 Iu5)	621
Rd=asl(Rs,#u5):sat	Word32 Q6_R_asl_RI_sat(Word32 Rs, Word32 Iu5)	633
Rd=asl(Rs,Rt)	Word32 Q6_R_asl_RR(Word32 Rs, Word32 Rt)	635
Rd=asl(Rs,Rt):sat	Word32 Q6_R_asl_RR_sat(Word32 Rs, Word32 Rt)	644
Rdd=asl(Rss,#u6)	Word64 Q6_P_asl_PI(Word64 Rss, Word32 Iu6)	622
Rdd=asl(Rss,Rt)	Word64 Q6_P_asl_PR(Word64 Rss, Word32 Rt)	635
Rx^=asl(Rs,#u5)	Word32 Q6_R_aslxacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	628
Rx&=asl(Rs,#u5)	Word32 Q6_R_asland_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	628
Rx&=asl(Rs,Rt)	Word32 Q6_R_asland_RR(Word32 Rx, Word32 Rs, Word32 Rt)	642
Rx+=asl(Rs,#u5)	Word32 Q6_R_aslacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	624
Rx+=asl(Rs,Rt)	Word32 Q6_R_aslacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)	638
Rx=add(#u8,asl(Rx,#U5))	Word32 Q6_R_add_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)	624
Rx=and(#u8,asl(Rx,#U5))	Word32 Q6_R_and_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)	628
Rx-=asl(Rs,#u5)	Word32 Q6_R_aslnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	624
Rx-=asl(Rs,Rt)	Word32 Q6_R_aslnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)	638
Rx=or(#u8,asl(Rx,#U5))	Word32 Q6_R_or_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)	628
Rx=sub(#u8,asl(Rx,#U5))	Word32 Q6_R_sub_asl_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)	624
Rx|=asl(Rs,#u5)	Word32 Q6_R_aslor_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	628
Rx|=asl(Rs,Rt)	Word32 Q6_R_aslor_RR(Word32 Rx, Word32 Rs, Word32 Rt)	642
Rxx^=asl(Rss,#u6)	Word64 Q6_P_aslxacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	629
Rxx^=asl(Rss,Rt)	Word64 Q6_P_aslxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	642
Rxx&=asl(Rss,#u6)	Word64 Q6_P_asland_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	628
Rxx&=asl(Rss,Rt)	Word64 Q6_P_asland_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	642
Rxx+=asl(Rss,#u6)	Word64 Q6_P_aslacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	624
Rxx+=asl(Rss,Rt)	Word64 Q6_P_aslacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	638
Rxx-=asl(Rss,#u6)	Word64 Q6_P_aslnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	624
Rxx-=asl(Rss,Rt)	Word64 Q6_P_aslnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	639
Rxx|=asl(Rss,#u6)	Word64 Q6_P_aslor_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	629
Rxx|=asl(Rss,Rt)	Word64 Q6_P_aslor_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	642

aslh

Rd=aslh(Rs)	Word32 Q6_R_aslh_R(Word32 Rs)	204

asr

Rd=asr(Rs,#u5)	Word32 Q6_R_asr_RI(Word32 Rs, Word32 Iu5)	621
Rd=asr(Rs,#u5):rnd	Word32 Q6_R_asr_RI_rnd(Word32 Rs, Word32 Iu5)	632
Rd=asr(Rs,Rt)	Word32 Q6_R_asr_RR(Word32 Rs, Word32 Rt)	635
Rd=asr(Rs,Rt):sat	Word32 Q6_R_asr_RR_sat(Word32 Rs, Word32 Rt)	644
Rdd=asr(Rss,#u6)	Word64 Q6_P_asr_PI(Word64 Rss, Word32 Iu6)	622
Rdd=asr(Rss,#u6):rnd	Word64 Q6_P_asr_PI_rnd(Word64 Rss, Word32 Iu6)	632
Rdd=asr(Rss,Rt)	Word64 Q6_P_asr_PR(Word64 Rss, Word32 Rt)	635
Rx&=asr(Rs,#u5)	Word32 Q6_R_asrand_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	628
Rx&=asr(Rs,Rt)	Word32 Q6_R_asrand_RR(Word32 Rx, Word32 Rs, Word32 Rt)	642
Rx+=asr(Rs,#u5)	Word32 Q6_R_asracc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	624
Rx+=asr(Rs,Rt)	Word32 Q6_R_asracc_RR(Word32 Rx, Word32 Rs, Word32 Rt)	638
Rx-=asr(Rs,#u5)	Word32 Q6_R_asrnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	624
Rx-=asr(Rs,Rt)	Word32 Q6_R_asrnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)	638
Rx|=asr(Rs,#u5)	Word32 Q6_R_asror_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	628
Rx|=asr(Rs,Rt)	Word32 Q6_R_asror_RR(Word32 Rx, Word32 Rs, Word32 Rt)	642
Rxx^=asr(Rss,Rt)	Word64 Q6_P_asrxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	642
Rxx&=asr(Rss,#u6)	Word64 Q6_P_asrand_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	628
Rxx&=asr(Rss,Rt)	Word64 Q6_P_asrand_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	642
Rxx+=asr(Rss,#u6)	Word64 Q6_P_asracc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	624
Rxx+=asr(Rss,Rt)	Word64 Q6_P_asracc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	638
Rxx-=asr(Rss,#u6)	Word64 Q6_P_asrnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	624
Rxx-=asr(Rss,Rt)	Word64 Q6_P_asrnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	639
Rxx|=asr(Rss,#u6)	Word64 Q6_P_asror_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	629
Rxx|=asr(Rss,Rt)	Word64 Q6_P_asror_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	642

asrh

Rd=asrh(Rs)	Word32 Q6_R_asrh_R(Word32 Rs)	204

asrrnd

Rd=asrrnd(Rs,#u5)	Word32 Q6_R_asrrnd_RI(Word32 Rs, Word32 Iu5)	632
Rdd=asrrnd(Rss,#u6)	Word64 Q6_P_asrrnd_PI(Word64 Rss, Word32 Iu6)	632

B

bitsclr

Pd=!bitsclr(Rs,#u6)	Byte Q6_p_not_bitsclr_RI(Word32 Rs, Word32 Iu6)	606
Pd=!bitsclr(Rs,Rt)	Byte Q6_p_not_bitsclr_RR(Word32 Rs, Word32 Rt)	606
Pd=bitsclr(Rs,#u6)	Byte Q6_p_bitsclr_RI(Word32 Rs, Word32 Iu6)	606
Pd=bitsclr(Rs,Rt)	Byte Q6_p_bitsclr_RR(Word32 Rs, Word32 Rt)	606

bitsplit

Rdd=bitsplit(Rs,#u5)	Word64 Q6_P_bitsplit_RI(Word32 Rs, Word32 Iu5)	455
Rdd=bitsplit(Rs,Rt)	Word64 Q6_P_bitsplit_RR(Word32 Rs, Word32 Rt)	455

bitsset

Pd=!bitsset(Rs,Rt)	Byte Q6_p_not_bitsset_RR(Word32 Rs, Word32 Rt)	606
Pd=bitsset(Rs,Rt)	Byte Q6_p_bitsset_RR(Word32 Rs, Word32 Rt)	606

boundscheck

Pd=boundscheck(Rs,Rtt)	Byte Q6_p_boundscheck_RP(Word32 Rs, Word64 Rtt)	600

brev

Rd=brev(Rs)	Word32 Q6_R_brev_R(Word32 Rs)	452

Rdd=brev(Rss)	Word64 Q6_P_brev_P(Word64 Rss)	452

C

cl0

Rd=cl0(Rs)	Word32 Q6_R_cl0_R(Word32 Rs)	440
Rd=cl0(Rss)	Word32 Q6_R_cl0_P(Word64 Rss)	440

cl1

Rd=cl1(Rs)	Word32 Q6_R_cl1_R(Word32 Rs)	440
Rd=cl1(Rss)	Word32 Q6_R_cl1_P(Word64 Rss)	440

clb

Rd=add(clb(Rs),#s6)	Word32 Q6_R_add_clb_RI(Word32 Rs, Word32 Is6)	440
Rd=add(clb(Rss),#s6)	Word32 Q6_R_add_clb_PI(Word64 Rss, Word32 Is6)	440
Rd=clb(Rs)	Word32 Q6_R_clb_R(Word32 Rs)	440
Rd=clb(Rss)	Word32 Q6_R_clb_P(Word64 Rss)	440

clrbit

Rd=clrbit(Rs,#u5)	Word32 Q6_R_clrbit_RI(Word32 Rs, Word32 Iu5)	453
Rd=clrbit(Rs,Rt)	Word32 Q6_R_clrbit_RR(Word32 Rs, Word32 Rt)	453

cmp.eq

Pd=!cmp.eq(Rs,#s10)	Byte Q6_p_not_cmp_eq_RI(Word32 Rs, Word32 Is10)	221
Pd=!cmp.eq(Rs,Rt)	Byte Q6_p_not_cmp_eq_RR(Word32 Rs, Word32 Rt)	221
Pd=cmp.eq(Rs,#s10)	Byte Q6_p_cmp_eq_RI(Word32 Rs, Word32 Is10)	221
Pd=cmp.eq(Rs,Rt)	Byte Q6_p_cmp_eq_RR(Word32 Rs, Word32 Rt)	221
Pd=cmp.eq(Rss,Rtt)	Byte Q6_p_cmp_eq_PP(Word64 Rss, Word64 Rtt)	605
Rd=!cmp.eq(Rs,#s8)	Word32 Q6_R_not_cmp_eq_RI(Word32 Rs, Word32 Is8)	223
Rd=!cmp.eq(Rs,Rt)	Word32 Q6_R_not_cmp_eq_RR(Word32 Rs, Word32 Rt)	223
Rd=cmp.eq(Rs,#s8)	Word32 Q6_R_cmp_eq_RI(Word32 Rs, Word32 Is8)	223
Rd=cmp.eq(Rs,Rt)	Word32 Q6_R_cmp_eq_RR(Word32 Rs, Word32 Rt)	223

cmp.ge

Pd=cmp.ge(Rs,#s8)	Byte Q6_p_cmp_ge_RI(Word32 Rs, Word32 Is8)	221

cmp.geu

Pd=cmp.geu(Rs,#u8)	Byte Q6_p_cmp_geu_RI(Word32 Rs, Word32 Iu8)	221

cmp.gt

Pd=!cmp.gt(Rs,#s10)	Byte Q6_p_not_cmp_gt_RI(Word32 Rs, Word32 Is10)	221
Pd=!cmp.gt(Rs,Rt)	Byte Q6_p_not_cmp_gt_RR(Word32 Rs, Word32 Rt)	221
Pd=cmp.gt(Rs,#s10)	Byte Q6_p_cmp_gt_RI(Word32 Rs, Word32 Is10)	221
Pd=cmp.gt(Rs,Rt)	Byte Q6_p_cmp_gt_RR(Word32 Rs, Word32 Rt)	222
Pd=cmp.gt(Rss,Rtt)	Byte Q6_p_cmp_gt_PP(Word64 Rss, Word64 Rtt)	605

cmp.gtu

Pd=!cmp.gtu(Rs,#u9)	Byte Q6_p_not_cmp_gtu_RI(Word32 Rs, Word32 Iu9)	221
Pd=!cmp.gtu(Rs,Rt)	Byte Q6_p_not_cmp_gtu_RR(Word32 Rs, Word32 Rt)	221
Pd=cmp.gtu(Rs,#u9)	Byte Q6_p_cmp_gtu_RI(Word32 Rs, Word32 Iu9)	222
Pd=cmp.gtu(Rs,Rt)	Byte Q6_p_cmp_gtu_RR(Word32 Rs, Word32 Rt)	222
Pd=cmp.gtu(Rss,Rtt)	Byte Q6_p_cmp_gtu_PP(Word64 Rss, Word64 Rtt)	605

cmp.lt

Pd=cmp.lt(Rs,Rt)	Byte Q6_p_cmp_lt_RR(Word32 Rs, Word32 Rt)	222

cmp.ltu

Pd=cmp.ltu(Rs,Rt)	Byte Q6_p_cmp_ltu_RR(Word32 Rs, Word32 Rt)	222

cmpb.eq

Pd=cmpb.eq(Rs,#u8)	Byte Q6_p_cmpb_eq_RI(Word32 Rs, Word32 Iu8)	601
Pd=cmpb.eq(Rs,Rt)	Byte Q6_p_cmpb_eq_RR(Word32 Rs, Word32 Rt)	601

cmpb.gt

Pd=cmpb.gt(Rs,#s8)	Byte Q6_p_cmpb_gt_RI(Word32 Rs, Word32 Is8)	601
Pd=cmpb.gt(Rs,Rt)	Byte Q6_p_cmpb_gt_RR(Word32 Rs, Word32 Rt)	601

cmpb.gtu

Pd=cmpb.gtu(Rs,#u7)	Byte Q6_p_cmpb_gtu_RI(Word32 Rs, Word32 Iu7)	601
Pd=cmpb.gtu(Rs,Rt)	Byte Q6_p_cmpb_gtu_RR(Word32 Rs, Word32 Rt)	601

cmph.eq

Pd=cmph.eq(Rs,#s8)	Byte Q6_p_cmph_eq_RI(Word32 Rs, Word32 Is8)	603
Pd=cmph.eq(Rs,Rt)	Byte Q6_p_cmph_eq_RR(Word32 Rs, Word32 Rt)	603

cmph.gt

Pd=cmph.gt(Rs,#s8)	Byte Q6_p_cmph_gt_RI(Word32 Rs, Word32 Is8)	603
Pd=cmph.gt(Rs,Rt)	Byte Q6_p_cmph_gt_RR(Word32 Rs, Word32 Rt)	603

cmph.gtu

Pd=cmph.gtu(Rs,#u7)	Byte Q6_p_cmph_gtu_RI(Word32 Rs, Word32 Iu7)	603
Pd=cmph.gtu(Rs,Rt)	Byte Q6_p_cmph_gtu_RR(Word32 Rs, Word32 Rt)	603

cmpy

Rd=cmpy(Rs,Rt):<<1:rnd:sat	Word32 Q6_R_cmpy_RR_s1_rnd_sat(Word32 Rs, Word32 Rt)	472
Rd=cmpy(Rs,Rt):rnd:sat	Word32 Q6_R_cmpy_RR_rnd_sat(Word32 Rs, Word32 Rt)	472
Rd=cmpy(Rs,Rt*):<<1:rnd:sat	Word32 Q6_R_cmpy_RR_conj_s1_rnd_sat(Word32 Rs, Word32 Rt)	472
Rd=cmpy(Rs,Rt*):rnd:sat	Word32 Q6_R_cmpy_RR_conj_rnd_sat(Word32 Rs, Word32 Rt)	472
Rdd=cmpy(Rs,Rt):<<1:sat	Word64 Q6_P_cmpy_RR_s1_sat(Word32 Rs, Word32 Rt)	467
Rdd=cmpy(Rs,Rt):sat	Word64 Q6_P_cmpy_RR_sat(Word32 Rs, Word32 Rt)	467
Rdd=cmpy(Rs,Rt*):<<1:sat	Word64 Q6_P_cmpy_RR_conj_s1_sat(Word32 Rs, Word32 Rt)	467
Rdd=cmpy(Rs,Rt*):sat	Word64 Q6_P_cmpy_RR_conj_sat(Word32 Rs, Word32 Rt)	467
Rxx+=cmpy(Rs,Rt):<<1:sat	Word64 Q6_P_cmpyacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)	467
Rxx+=cmpy(Rs,Rt):sat	Word64 Q6_P_cmpyacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)	467
Rxx+=cmpy(Rs,Rt*):<<1:sat	Word64 Q6_P_cmpyacc_RR_conj_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)	467
Rxx+=cmpy(Rs,Rt*):sat	Word64 Q6_P_cmpyacc_RR_conj_sat(Word64 Rxx, Word32 Rs, Word32 Rt)	468
Rxx-=cmpy(Rs,Rt):<<1:sat	Word64 Q6_P_cmpynac_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)	468
Rxx-=cmpy(Rs,Rt):sat	Word64 Q6_P_cmpynac_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)	468
Rxx-=cmpy(Rs,Rt*):<<1:sat	Word64 Q6_P_cmpynac_RR_conj_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)	468
Rxx-=cmpy(Rs,Rt*):sat	Word64 Q6_P_cmpynac_RR_conj_sat(Word64 Rxx, Word32 Rs, Word32 Rt)	468

cmpyi

Rdd=cmpyi(Rs,Rt)	Word64 Q6_P_cmpyi_RR(Word32 Rs, Word32 Rt)	469
Rxx+=cmpyi(Rs,Rt)	Word64 Q6_P_cmpyiacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)	469

cmpyiwh

Rd=cmpyiwh(Rss,Rt):<<1:rnd:sat	Word32 Q6_R_cmpyiwh_PR_s1_rnd_sat(Word64 Rss, Word32 Rt)	474

Rd=cmpyiwh(Rss,Rt*):<<1:rnd:sat	Word32 Q6_R_cmpyiwh_PR_conj_s1_rnd_sat(Word64 Rss, Word32 Rt)	474

cmpyr

Rdd=cmpyr(Rs,Rt)	Word64 Q6_P_cmpyr_RR(Word32 Rs, Word32 Rt)	469
Rxx+=cmpyr(Rs,Rt)	Word64 Q6_P_cmpyracc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)	469

cmpyrwh

Rd=cmpyrwh(Rss,Rt):<<1:rnd:sat	Word32 Q6_R_cmpyrwh_PR_s1_rnd_sat(Word64 Rss, Word32 Rt)	474
Rd=cmpyrwh(Rss,Rt*):<<1:rnd:sat	Word32 Q6_R_cmpyrwh_PR_conj_s1_rnd_sat(Word64 Rss, Word32 Rt)	474

combine

Rd=combine(Rt.H,Rs.H)	Word32 Q6_R_combine_RhRh(Word32 Rt, Word32 Rs)	201
Rd=combine(Rt.H,Rs.L)	Word32 Q6_R_combine_RhRl(Word32 Rt, Word32 Rs)	201
Rd=combine(Rt.L,Rs.H)	Word32 Q6_R_combine_RlRh(Word32 Rt, Word32 Rs)	201
Rd=combine(Rt.L,Rs.L)	Word32 Q6_R_combine_RlRl(Word32 Rt, Word32 Rs)	201
Rdd=combine(#s8,#S8)	Word64 Q6_P_combine_II(Word32 Is8, Word32 IS8)	201
Rdd=combine(#s8,Rs)	Word64 Q6_P_combine_IR(Word32 Is8, Word32 Rs)	201
Rdd=combine(Rs,#s8)	Word64 Q6_P_combine_RI(Word32 Rs, Word32 Is8)	201
Rdd=combine(Rs,Rt)	Word64 Q6_P_combine_RR(Word32 Rs, Word32 Rt)	201

convert_d2df

Rdd=convert_d2df(Rss)	Word64 Q6_P_convert_d2df_P(Word64 Rss)	500

convert_d2sf

Rd=convert_d2sf(Rss)	Word32 Q6_R_convert_d2sf_P(Word64 Rss)	500

convert_df2d

Rdd=convert_df2d(Rss)	Word64 Q6_P_convert_df2d_P(Word64 Rss)	503
Rdd=convert_df2d(Rss):chop	Word64 Q6_P_convert_df2d_P_chop(Word64 Rss)	503

convert_df2sf

Rd=convert_df2sf(Rss)	Word32 Q6_R_convert_df2sf_P(Word64 Rss)	499

convert_df2ud

Rdd=convert_df2ud(Rss)	Word64 Q6_P_convert_df2ud_P(Word64 Rss)	503
Rdd=convert_df2ud(Rss):chop	Word64 Q6_P_convert_df2ud_P_chop(Word64 Rss)	503

convert_df2uw

Rd=convert_df2uw(Rss)	Word32 Q6_R_convert_df2uw_P(Word64 Rss)	502
Rd=convert_df2uw(Rss):chop	Word32 Q6_R_convert_df2uw_P_chop(Word64 Rss)	502

convert_df2w

Rd=convert_df2w(Rss)	Word32 Q6_R_convert_df2w_P(Word64 Rss)	502
Rd=convert_df2w(Rss):chop	Word32 Q6_R_convert_df2w_P_chop(Word64 Rss)	502

convert_sf2d

Rdd=convert_sf2d(Rs)	Word64 Q6_P_convert_sf2d_R(Word32 Rs)	503
Rdd=convert_sf2d(Rs):chop	Word64 Q6_P_convert_sf2d_R_chop(Word32 Rs)	503

convert_sf2df

Rdd=convert_sf2df(Rs)	Word64 Q6_P_convert_sf2df_R(Word32 Rs)	499

convert_sf2ud

Rdd=convert_sf2ud(Rs)	Word64 Q6_P_convert_sf2ud_R(Word32 Rs)	503

Rdd=convert_sf2ud(Rs):chop	Word64 Q6_P_convert_sf2ud_R_chop(Word32 Rs)	503

convert_sf2uw

Rd=convert_sf2uw(Rs)	Word32 Q6_R_convert_sf2uw_R(Word32 Rs)	502
Rd=convert_sf2uw(Rs):chop	Word32 Q6_R_convert_sf2uw_R_chop(Word32 Rs)	502

convert_sf2w

Rd=convert_sf2w(Rs)	Word32 Q6_R_convert_sf2w_R(Word32 Rs)	503
Rd=convert_sf2w(Rs):chop	Word32 Q6_R_convert_sf2w_R_chop(Word32 Rs)	503

convert_ud2df

Rdd=convert_ud2df(Rss)	Word64 Q6_P_convert_ud2df_P(Word64 Rss)	500

convert_ud2sf

Rd=convert_ud2sf(Rss)	Word32 Q6_R_convert_ud2sf_P(Word64 Rss)	500

convert_uw2df

Rdd=convert_uw2df(Rs)	Word64 Q6_P_convert_uw2df_R(Word32 Rs)	500

convert_uw2sf

Rd=convert_uw2sf(Rs)	Word32 Q6_R_convert_uw2sf_R(Word32 Rs)	500

convert_w2df

Rdd=convert_w2df(Rs)	Word64 Q6_P_convert_w2df_R(Word32 Rs)	500

convert_w2sf

Rd=convert_w2sf(Rs)	Word32 Q6_R_convert_w2sf_R(Word32 Rs)	500

cround

Rd=cround(Rs,#u5)	Word32 Q6_R_cround_RI(Word32 Rs, Word32 Iu5)	386
Rd=cround(Rs,Rt)	Word32 Q6_R_cround_RR(Word32 Rs, Word32 Rt)	386

ct0

Rd=ct0(Rs)	Word32 Q6_R_ct0_R(Word32 Rs)	442
Rd=ct0(Rss)	Word32 Q6_R_ct0_P(Word64 Rss)	442

ct1

Rd=ct1(Rs)	Word32 Q6_R_ct1_R(Word32 Rs)	442
Rd=ct1(Rss)	Word32 Q6_R_ct1_P(Word64 Rss)	442

D

dcfetch

dcfetch(Rs)	void Q6_dcfetch_A(Address a)	353

deinterleave

Rdd=deinterleave(Rss)	Word64 Q6_P_deinterleave_P(Word64 Rss)	449

dfclass

Pd=dfclass(Rss,#u5)	Byte Q6_p_dfclass_PI(Word64 Rss, Word32 Iu5)	496

dfcmp.eq

Pd=dfcmp.eq(Rss,Rtt)	Byte Q6_p_dfcmp_eq_PP(Word64 Rss, Word64 Rtt)	497

dfcmp.ge

Pd=dfcmp.ge(Rss,Rtt)	Byte Q6_p_dfcmp_ge_PP(Word64 Rss, Word64 Rtt)	497

dfcmp.gt

Pd=dfcmp.gt(Rss,Rtt)	Byte Q6_p_dfcmp_gt_PP(Word64 Rss, Word64 Rtt)	497

dfcmp.uo

Pd=dfcmp.uo(Rss,Rtt)	Byte Q6_p_dfcmp_uo_PP(Word64 Rss, Word64 Rtt)	497

dfmake

Rdd=dfmake(#u10):neg	Word64 Q6_P_dfmake_I_neg(Word32 Iu10)	510
Rdd=dfmake(#u10):pos	Word64 Q6_P_dfmake_I_pos(Word32 Iu10)	510

E

extract

Rd=extract(Rs,#u5,#U5)	Word32 Q6_R_extract_RII(Word32 Rs, Word32 Iu5, Word32 IU5)	445
Rd=extract(Rs,Rtt)	Word32 Q6_R_extract_RP(Word32 Rs, Word64 Rtt)	445
Rdd=extract(Rss,#u6,#U6)	Word64 Q6_P_extract_PII(Word64 Rss, Word32 Iu6, Word32 IU6)	445
Rdd=extract(Rss,Rtt)	Word64 Q6_P_extract_PP(Word64 Rss, Word64 Rtt)	445

extractu

Rd=extractu(Rs,#u5,#U5)	Word32 Q6_R_extractu_RII(Word32 Rs, Word32 Iu5, Word32 IU5)	445
Rd=extractu(Rs,Rtt)	Word32 Q6_R_extractu_RP(Word32 Rs, Word64 Rtt)	445
Rdd=extractu(Rss,#u6,#U6)	Word64 Q6_P_extractu_PII(Word64 Rss, Word32 Iu6, Word32 IU6)	445
Rdd=extractu(Rss,Rtt)	Word64 Q6_P_extractu_PP(Word64 Rss, Word64 Rtt)	445

F

fastcorner9

Pd=!fastcorner9(Ps,Pt)	Byte Q6_p_not_fastcorner9_pp(Byte Ps, Byte Pt)	226
Pd=fastcorner9(Ps,Pt)	Byte Q6_p_fastcorner9_pp(Byte Ps, Byte Pt)	226

I

insert

Rx=insert(Rs,#u5,#U5)	Word32 Q6_R_insert_RII(Word32 Rx, Word32 Rs, Word32 Iu5, Word32 IU5)	448
Rx=insert(Rs,Rtt)	Word32 Q6_R_insert_RP(Word32 Rx, Word32 Rs, Word64 Rtt)	448
Rxx=insert(Rss,#u6,#U6)	Word64 Q6_P_insert_PII(Word64 Rxx, Word64 Rss, Word32 Iu6, Word32 IU6)	448
Rxx=insert(Rss,Rtt)	Word64 Q6_P_insert_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)	448

interleave

Rdd=interleave(Rss)	Word64 Q6_P_interleave_P(Word64 Rss)	449

L

lfs

Rdd=lfs(Rss,Rtt)	Word64 Q6_P_lfs_PP(Word64 Rss, Word64 Rtt)	450

lsl

Rd=lsl(#s6,Rt)	Word32 Q6_R_lsl_IR(Word32 Is6, Word32 Rt)	635
Rd=lsl(Rs,Rt)	Word32 Q6_R_lsl_RR(Word32 Rs, Word32 Rt)	635
Rdd=lsl(Rss,Rt)	Word64 Q6_P_lsl_PR(Word64 Rss, Word32 Rt)	635
Rx&=lsl(Rs,Rt)	Word32 Q6_R_lsland_RR(Word32 Rx, Word32 Rs, Word32 Rt)	642
Rx+=lsl(Rs,Rt)	Word32 Q6_R_lslacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)	638

Rx-=lsl(Rs,Rt)	Word32 Q6_R_lslnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)	638
Rx|=lsl(Rs,Rt)	Word32 Q6_R_lslor_RR(Word32 Rx, Word32 Rs, Word32 Rt)	642
Rxx^=lsl(Rss,Rt)	Word64 Q6_P_lslxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	642
Rxx&=lsl(Rss,Rt)	Word64 Q6_P_lsland_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	642
Rxx+=lsl(Rss,Rt)	Word64 Q6_P_lslacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	638
Rxx-=lsl(Rss,Rt)	Word64 Q6_P_lslnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	639
Rxx|=lsl(Rss,Rt)	Word64 Q6_P_lslor_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	642

lsr

Rd=lsr(Rs,#u5)	Word32 Q6_R_lsr_RI(Word32 Rs, Word32 Iu5)	621
Rd=lsr(Rs,Rt)	Word32 Q6_R_lsr_RR(Word32 Rs, Word32 Rt)	635
Rdd=lsr(Rss,#u6)	Word64 Q6_P_lsr_PI(Word64 Rss, Word32 Iu6)	622
Rdd=lsr(Rss,Rt)	Word64 Q6_P_lsr_PR(Word64 Rss, Word32 Rt)	635
Rx^=lsr(Rs,#u5)	Word32 Q6_R_lsrxacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	628
Rx&=lsr(Rs,#u5)	Word32 Q6_R_lsrand_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	628
Rx&=lsr(Rs,Rt)	Word32 Q6_R_lsrand_RR(Word32 Rx, Word32 Rs, Word32 Rt)	642
Rx+=lsr(Rs,#u5)	Word32 Q6_R_lsracc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	624
Rx+=lsr(Rs,Rt)	Word32 Q6_R_lsracc_RR(Word32 Rx, Word32 Rs, Word32 Rt)	638
Rx=add(#u8,lsr(Rx,#U5))	Word32 Q6_R_add_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)	624
Rx=and(#u8,lsr(Rx,#U5))	Word32 Q6_R_and_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)	628
Rx-=lsr(Rs,#u5)	Word32 Q6_R_lsrnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	624
Rx-=lsr(Rs,Rt)	Word32 Q6_R_lsrnac_RR(Word32 Rx, Word32 Rs, Word32 Rt)	638
Rx=or(#u8,lsr(Rx,#U5))	Word32 Q6_R_or_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)	628
Rx=sub(#u8,lsr(Rx,#U5))	Word32 Q6_R_sub_lsr_IRI(Word32 Iu8, Word32 Rx, Word32 IU5)	624
Rx|=lsr(Rs,#u5)	Word32 Q6_R_lsror_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	628
Rx|=lsr(Rs,Rt)	Word32 Q6_R_lsror_RR(Word32 Rx, Word32 Rs, Word32 Rt)	642
Rxx^=lsr(Rss,#u6)	Word64 Q6_P_lsrxacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	629
Rxx^=lsr(Rss,Rt)	Word64 Q6_P_lsrxacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	642
Rxx&=lsr(Rss,#u6)	Word64 Q6_P_lsrand_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	628
Rxx&=lsr(Rss,Rt)	Word64 Q6_P_lsrand_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	642
Rxx+=lsr(Rss,#u6)	Word64 Q6_P_lsracc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	624
Rxx+=lsr(Rss,Rt)	Word64 Q6_P_lsracc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	639
Rxx-=lsr(Rss,#u6)	Word64 Q6_P_lsrnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	624
Rxx-=lsr(Rss,Rt)	Word64 Q6_P_lsrnac_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	639
Rxx|=lsr(Rss,#u6)	Word64 Q6_P_lsror_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	629
Rxx|=lsr(Rss,Rt)	Word64 Q6_P_lsror_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	642

M

mask

Rdd=mask(Pt)	Word64 Q6_P_mask_p(Byte Pt)	607

max

Rd=max(Rs,Rt)	Word32 Q6_R_max_RR(Word32 Rs, Word32 Rt)	380
Rdd=max(Rss,Rtt)	Word64 Q6_P_max_PP(Word64 Rss, Word64 Rtt)	381

maxu

Rd=maxu(Rs,Rt)	UWord32 Q6_R_maxu_RR(Word32 Rs, Word32 Rt)	380
Rdd=maxu(Rss,Rtt)	UWord64 Q6_P_maxu_PP(Word64 Rss, Word64 Rtt)	381

min

Rd=min(Rt,Rs)	Word32 Q6_R_min_RR(Word32 Rt, Word32 Rs)	382
Rdd=min(Rtt,Rss)	Word64 Q6_P_min_PP(Word64 Rtt, Word64 Rss)	383

minu

Rd=minu(Rt,Rs)	UWord32 Q6_R_minu_RR(Word32 Rt, Word32 Rs)	382
Rdd=minu(Rtt,Rss)	UWord64 Q6_P_minu_PP(Word64 Rtt, Word64 Rss)	383

modwrap

Rd=modwrap(Rs,Rt)	Word32 Q6_R_modwrap_RR(Word32 Rs, Word32 Rt)	384

mpy

Rd=mpy(Rs,Rt.H):<<1:rnd:sat	Word32 Q6_R_mpy_RRh_s1_rnd_sat(Word32 Rs, Word32 Rt)	546
Rd=mpy(Rs,Rt.H):<<1:sat	Word32 Q6_R_mpy_RRh_s1_sat(Word32 Rs, Word32 Rt)	546
Rd=mpy(Rs,Rt.L):<<1:rnd:sat	Word32 Q6_R_mpy_RRl_s1_rnd_sat(Word32 Rs, Word32 Rt)	546
Rd=mpy(Rs,Rt.L):<<1:sat	Word32 Q6_R_mpy_RRl_s1_sat(Word32 Rs, Word32 Rt)	546
Rd=mpy(Rs,Rt)	Word32 Q6_R_mpy_RR(Word32 Rs, Word32 Rt)	546
Rd=mpy(Rs,Rt):<<1	Word32 Q6_R_mpy_RR_s1(Word32 Rs, Word32 Rt)	546
Rd=mpy(Rs,Rt):<<1:sat	Word32 Q6_R_mpy_RR_s1_sat(Word32 Rs, Word32 Rt)	546
Rd=mpy(Rs,Rt):rnd	Word32 Q6_R_mpy_RR_rnd(Word32 Rs, Word32 Rt)	546
Rd=mpy(Rs.H,Rt.H)	Word32 Q6_R_mpy_RhRh(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.H):<<1	Word32 Q6_R_mpy_RhRh_s1(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.H):<<1:rnd	Word32 Q6_R_mpy_RhRh_s1_rnd(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.H):<<1:rnd:sat	Word32 Q6_R_mpy_RhRh_s1_rnd_sat(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.H):<<1:sat	Word32 Q6_R_mpy_RhRh_s1_sat(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.H):rnd	Word32 Q6_R_mpy_RhRh_rnd(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.H):rnd:sat	Word32 Q6_R_mpy_RhRh_rnd_sat(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.H):sat	Word32 Q6_R_mpy_RhRh_sat(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.L)	Word32 Q6_R_mpy_RhRl(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.L):<<1	Word32 Q6_R_mpy_RhRl_s1(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.L):<<1:rnd	Word32 Q6_R_mpy_RhRl_s1_rnd(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.L):<<1:rnd:sat	Word32 Q6_R_mpy_RhRl_s1_rnd_sat(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.L):<<1:sat	Word32 Q6_R_mpy_RhRl_s1_sat(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.L):rnd	Word32 Q6_R_mpy_RhRl_rnd(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.L):rnd:sat	Word32 Q6_R_mpy_RhRl_rnd_sat(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.H,Rt.L):sat	Word32 Q6_R_mpy_RhRl_sat(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.L,Rt.H)	Word32 Q6_R_mpy_RlRh(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.L,Rt.H):<<1	Word32 Q6_R_mpy_RlRh_s1(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.L,Rt.H):<<1:rnd	Word32 Q6_R_mpy_RlRh_s1_rnd(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.L,Rt.H):<<1:rnd:sat	Word32 Q6_R_mpy_RlRh_s1_rnd_sat(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.L,Rt.H):<<1:sat	Word32 Q6_R_mpy_RlRh_s1_sat(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.L,Rt.H):rnd	Word32 Q6_R_mpy_RlRh_rnd(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.L,Rt.H):rnd:sat	Word32 Q6_R_mpy_RlRh_rnd_sat(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.L,Rt.H):sat	Word32 Q6_R_mpy_RlRh_sat(Word32 Rs, Word32 Rt)	530
Rd=mpy(Rs.L,Rt.L)	Word32 Q6_R_mpy_RlRl(Word32 Rs, Word32 Rt)	531
Rd=mpy(Rs.L,Rt.L):<<1	Word32 Q6_R_mpy_RlRl_s1(Word32 Rs, Word32 Rt)	531
Rd=mpy(Rs.L,Rt.L):<<1:rnd	Word32 Q6_R_mpy_RlRl_s1_rnd(Word32 Rs, Word32 Rt)	531
Rd=mpy(Rs.L,Rt.L):<<1:rnd:sat	Word32 Q6_R_mpy_RlRl_s1_rnd_sat(Word32 Rs, Word32 Rt)	531
Rd=mpy(Rs.L,Rt.L):<<1:sat	Word32 Q6_R_mpy_RlRl_s1_sat(Word32 Rs, Word32 Rt)	531
Rd=mpy(Rs.L,Rt.L):rnd	Word32 Q6_R_mpy_RlRl_rnd(Word32 Rs, Word32 Rt)	531
Rd=mpy(Rs.L,Rt.L):rnd:sat	Word32 Q6_R_mpy_RlRl_rnd_sat(Word32 Rs, Word32 Rt)	531
Rd=mpy(Rs.L,Rt.L):sat	Word32 Q6_R_mpy_RlRl_sat(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs,Rt)	Word64 Q6_P_mpy_RR(Word32 Rs, Word32 Rt)	548
Rdd=mpy(Rs.H,Rt.H)	Word64 Q6_P_mpy_RhRh(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs.H,Rt.H):<<1	Word64 Q6_P_mpy_RhRh_s1(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs.H,Rt.H):<<1:rnd	Word64 Q6_P_mpy_RhRh_s1_rnd(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs.H,Rt.H):rnd	Word64 Q6_P_mpy_RhRh_rnd(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs.H,Rt.L)	Word64 Q6_P_mpy_RhRl(Word32 Rs, Word32 Rt)	531

Rdd=mpy(Rs.H,Rt.L):<<1	Word64 Q6_P_mpy_RhRl_s1(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs.H,Rt.L):<<1:rnd	Word64 Q6_P_mpy_RhRl_s1_rnd(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs.H,Rt.L):rnd	Word64 Q6_P_mpy_RhRl_rnd(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs.L,Rt.H)	Word64 Q6_P_mpy_RlRh(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs.L,Rt.H):<<1	Word64 Q6_P_mpy_RlRh_s1(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs.L,Rt.H):<<1:rnd	Word64 Q6_P_mpy_RlRh_s1_rnd(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs.L,Rt.H):rnd	Word64 Q6_P_mpy_RlRh_rnd(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs.L,Rt.L)	Word64 Q6_P_mpy_RlRl(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs.L,Rt.L):<<1	Word64 Q6_P_mpy_RlRl_s1(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs.L,Rt.L):<<1:rnd	Word64 Q6_P_mpy_RlRl_s1_rnd(Word32 Rs, Word32 Rt)	531
Rdd=mpy(Rs.L,Rt.L):rnd	Word64 Q6_P_mpy_RlRl_rnd(Word32 Rs, Word32 Rt)	531
Rx+=mpy(Rs,Rt):<<1:sat	Word32 Q6_R_mpyacc_RR_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)	546
Rx+=mpy(Rs.H,Rt.H)	Word32 Q6_R_mpyacc_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)	531
Rx+=mpy(Rs.H,Rt.H):<<1	Word32 Q6_R_mpyacc_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)	531
Rx+=mpy(Rs.H,Rt.H):<<1:sat	Word32 Q6_R_mpyacc_RhRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)	531
Rx+=mpy(Rs.H,Rt.H):sat	Word32 Q6_R_mpyacc_RhRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)	531
Rx+=mpy(Rs.H,Rt.L)	Word32 Q6_R_mpyacc_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)	531
Rx+=mpy(Rs.H,Rt.L):<<1	Word32 Q6_R_mpyacc_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)	531
Rx+=mpy(Rs.H,Rt.L):<<1:sat	Word32 Q6_R_mpyacc_RhRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)	531
Rx+=mpy(Rs.H,Rt.L):sat	Word32 Q6_R_mpyacc_RhRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx+=mpy(Rs.L,Rt.H)	Word32 Q6_R_mpyacc_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx+=mpy(Rs.L,Rt.H):<<1	Word32 Q6_R_mpyacc_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx+=mpy(Rs.L,Rt.H):<<1:sat	Word32 Q6_R_mpyacc_RlRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx+=mpy(Rs.L,Rt.H):sat	Word32 Q6_R_mpyacc_RlRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx+=mpy(Rs.L,Rt.L)	Word32 Q6_R_mpyacc_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx+=mpy(Rs.L,Rt.L):<<1	Word32 Q6_R_mpyacc_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx+=mpy(Rs.L,Rt.L):<<1:sat	Word32 Q6_R_mpyacc_RlRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx+=mpy(Rs.L,Rt.L):sat	Word32 Q6_R_mpyacc_RlRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs,Rt):<<1:sat	Word32 Q6_R_mpynac_RR_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)	546
Rx-=mpy(Rs.H,Rt.H)	Word32 Q6_R_mpynac_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.H,Rt.H):<<1	Word32 Q6_R_mpynac_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.H,Rt.H):<<1:sat	Word32 Q6_R_mpynac_RhRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.H,Rt.H):sat	Word32 Q6_R_mpynac_RhRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.H,Rt.L)	Word32 Q6_R_mpynac_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.H,Rt.L):<<1	Word32 Q6_R_mpynac_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.H,Rt.L):<<1:sat	Word32 Q6_R_mpynac_RhRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.H,Rt.L):sat	Word32 Q6_R_mpynac_RhRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.L,Rt.H)	Word32 Q6_R_mpynac_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.L,Rt.H):<<1	Word32 Q6_R_mpynac_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.L,Rt.H):<<1:sat	Word32 Q6_R_mpynac_RlRh_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.L,Rt.H):sat	Word32 Q6_R_mpynac_RlRh_sat(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.L,Rt.L)	Word32 Q6_R_mpynac_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.L,Rt.L):<<1	Word32 Q6_R_mpynac_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.L,Rt.L):<<1:sat	Word32 Q6_R_mpynac_RlRl_s1_sat(Word32 Rx, Word32 Rs, Word32 Rt)	532
Rx-=mpy(Rs.L,Rt.L):sat	Word32 Q6_R_mpynac_RlRl_sat(Word32 Rx, Word32 Rs, Word32 Rt)	533
Rxx+=mpy(Rs,Rt)	Word64 Q6_P_mpyacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)	548
Rxx+=mpy(Rs.H,Rt.H)	Word64 Q6_P_mpyacc_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)	533
Rxx+=mpy(Rs.H,Rt.H):<<1	Word64 Q6_P_mpyacc_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	533
Rxx+=mpy(Rs.H,Rt.L)	Word64 Q6_P_mpyacc_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)	533
Rxx+=mpy(Rs.H,Rt.L):<<1	Word64 Q6_P_mpyacc_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	533
Rxx+=mpy(Rs.L,Rt.H)	Word64 Q6_P_mpyacc_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)	533
Rxx+=mpy(Rs.L,Rt.H):<<1	Word64 Q6_P_mpyacc_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	533
Rxx+=mpy(Rs.L,Rt.L)	Word64 Q6_P_mpyacc_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)	533
Rxx+=mpy(Rs.L,Rt.L):<<1	Word64 Q6_P_mpyacc_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	533

Rxx-=mpy(Rs,Rt)	Word64 Q6_P_mpynac_RR(Word64 Rxx, Word32 Rs, Word32 Rt)	548
Rxx-=mpy(Rs.H,Rt.H)	Word64 Q6_P_mpynac_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)	533
Rxx-=mpy(Rs.H,Rt.H):<<1	Word64 Q6_P_mpynac_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	533
Rxx-=mpy(Rs.H,Rt.L)	Word64 Q6_P_mpynac_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)	533
Rxx-=mpy(Rs.H,Rt.L):<<1	Word64 Q6_P_mpynac_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	533
Rxx-=mpy(Rs.L,Rt.H)	Word64 Q6_P_mpynac_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)	533
Rxx-=mpy(Rs.L,Rt.H):<<1	Word64 Q6_P_mpynac_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	533
Rxx-=mpy(Rs.L,Rt.L)	Word64 Q6_P_mpynac_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)	533
Rxx-=mpy(Rs.L,Rt.L):<<1	Word64 Q6_P_mpynac_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	533

mpyi

Rd=mpyi(Rs,#m9)	Word32 Q6_R_mpyi_RI(Word32 Rs, Word32 Im9)	517
Rd=mpyi(Rs,Rt)	Word32 Q6_R_mpyi_RR(Word32 Rs, Word32 Rt)	517
Rx+=mpyi(Rs,#u8)	Word32 Q6_R_mpyiacc_RI(Word32 Rx, Word32 Rs, Word32 Iu8)	517
Rx+=mpyi(Rs,Rt)	Word32 Q6_R_mpyiacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)	517
Rx-=mpyi(Rs,#u8)	Word32 Q6_R_mpyinac_RI(Word32 Rx, Word32 Rs, Word32 Iu8)	517

mpysu

Rd=mpysu(Rs,Rt)	Word32 Q6_R_mpysu_RR(Word32 Rs, Word32 Rt)	546

mpyu

Rd=mpyu(Rs,Rt)	UWord32 Q6_R_mpyu_RR(Word32 Rs, Word32 Rt)	546
Rd=mpyu(Rs.H,Rt.H)	UWord32 Q6_R_mpyu_RhRh(Word32 Rs, Word32 Rt)	537
Rd=mpyu(Rs.H,Rt.H):<<1	UWord32 Q6_R_mpyu_RhRh_s1(Word32 Rs, Word32 Rt)	537
Rd=mpyu(Rs.H,Rt.L)	UWord32 Q6_R_mpyu_RhRl(Word32 Rs, Word32 Rt)	537
Rd=mpyu(Rs.H,Rt.L):<<1	UWord32 Q6_R_mpyu_RhRl_s1(Word32 Rs, Word32 Rt)	537
Rd=mpyu(Rs.L,Rt.H)	UWord32 Q6_R_mpyu_RlRh(Word32 Rs, Word32 Rt)	537
Rd=mpyu(Rs.L,Rt.H):<<1	UWord32 Q6_R_mpyu_RlRh_s1(Word32 Rs, Word32 Rt)	537
Rd=mpyu(Rs.L,Rt.L)	UWord32 Q6_R_mpyu_RlRl(Word32 Rs, Word32 Rt)	537
Rd=mpyu(Rs.L,Rt.L):<<1	UWord32 Q6_R_mpyu_RlRl_s1(Word32 Rs, Word32 Rt)	537
Rdd=mpyu(Rs,Rt)	UWord64 Q6_P_mpyu_RR(Word32 Rs, Word32 Rt)	548
Rdd=mpyu(Rs.H,Rt.H)	UWord64 Q6_P_mpyu_RhRh(Word32 Rs, Word32 Rt)	537
Rdd=mpyu(Rs.H,Rt.H):<<1	UWord64 Q6_P_mpyu_RhRh_s1(Word32 Rs, Word32 Rt)	537
Rdd=mpyu(Rs.H,Rt.L)	UWord64 Q6_P_mpyu_RhRl(Word32 Rs, Word32 Rt)	537
Rdd=mpyu(Rs.H,Rt.L):<<1	UWord64 Q6_P_mpyu_RhRl_s1(Word32 Rs, Word32 Rt)	537
Rdd=mpyu(Rs.L,Rt.H)	UWord64 Q6_P_mpyu_RlRh(Word32 Rs, Word32 Rt)	537
Rdd=mpyu(Rs.L,Rt.H):<<1	UWord64 Q6_P_mpyu_RlRh_s1(Word32 Rs, Word32 Rt)	537
Rdd=mpyu(Rs.L,Rt.L)	UWord64 Q6_P_mpyu_RlRl(Word32 Rs, Word32 Rt)	537
Rdd=mpyu(Rs.L,Rt.L):<<1	UWord64 Q6_P_mpyu_RlRl_s1(Word32 Rs, Word32 Rt)	537
Rx+=mpyu(Rs.H,Rt.H)	Word32 Q6_R_mpyuacc_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)	537
Rx+=mpyu(Rs.H,Rt.H):<<1	Word32 Q6_R_mpyuacc_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)	537
Rx+=mpyu(Rs.H,Rt.L)	Word32 Q6_R_mpyuacc_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)	537
Rx+=mpyu(Rs.H,Rt.L):<<1	Word32 Q6_R_mpyuacc_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)	537
Rx+=mpyu(Rs.L,Rt.H)	Word32 Q6_R_mpyuacc_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)	537
Rx+=mpyu(Rs.L,Rt.H):<<1	Word32 Q6_R_mpyuacc_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)	537
Rx+=mpyu(Rs.L,Rt.L)	Word32 Q6_R_mpyuacc_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)	537
Rx+=mpyu(Rs.L,Rt.L):<<1	Word32 Q6_R_mpyuacc_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)	537
Rx-=mpyu(Rs.H,Rt.H)	Word32 Q6_R_mpyunac_RhRh(Word32 Rx, Word32 Rs, Word32 Rt)	537
Rx-=mpyu(Rs.H,Rt.H):<<1	Word32 Q6_R_mpyunac_RhRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)	537
Rx-=mpyu(Rs.H,Rt.L)	Word32 Q6_R_mpyunac_RhRl(Word32 Rx, Word32 Rs, Word32 Rt)	537
Rx-=mpyu(Rs.H,Rt.L):<<1	Word32 Q6_R_mpyunac_RhRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)	537
Rx-=mpyu(Rs.L,Rt.H)	Word32 Q6_R_mpyunac_RlRh(Word32 Rx, Word32 Rs, Word32 Rt)	538
Rx-=mpyu(Rs.L,Rt.H):<<1	Word32 Q6_R_mpyunac_RlRh_s1(Word32 Rx, Word32 Rs, Word32 Rt)	538
Rx-=mpyu(Rs.L,Rt.L)	Word32 Q6_R_mpyunac_RlRl(Word32 Rx, Word32 Rs, Word32 Rt)	538

Rx-=mpyu(Rs.L,Rt.L):<<1	Word32 Q6_R_mpyunac_RlRl_s1(Word32 Rx, Word32 Rs, Word32 Rt)	538
Rxx+=mpyu(Rs,Rt)	Word64 Q6_P_mpyuacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)	548
Rxx+=mpyu(Rs.H,Rt.H)	Word64 Q6_P_mpyuacc_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx+=mpyu(Rs.H,Rt.H):<<1	Word64 Q6_P_mpyuacc_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx+=mpyu(Rs.H,Rt.L)	Word64 Q6_P_mpyuacc_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx+=mpyu(Rs.H,Rt.L):<<1	Word64 Q6_P_mpyuacc_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx+=mpyu(Rs.L,Rt.H)	Word64 Q6_P_mpyuacc_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx+=mpyu(Rs.L,Rt.H):<<1	Word64 Q6_P_mpyuacc_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx+=mpyu(Rs.L,Rt.L)	Word64 Q6_P_mpyuacc_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx+=mpyu(Rs.L,Rt.L):<<1	Word64 Q6_P_mpyuacc_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx-=mpyu(Rs,Rt)	Word64 Q6_P_mpyunac_RR(Word64 Rxx, Word32 Rs, Word32 Rt)	548
Rxx-=mpyu(Rs.H,Rt.H)	Word64 Q6_P_mpyunac_RhRh(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx-=mpyu(Rs.H,Rt.H):<<1	Word64 Q6_P_mpyunac_RhRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx-=mpyu(Rs.H,Rt.L)	Word64 Q6_P_mpyunac_RhRl(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx-=mpyu(Rs.H,Rt.L):<<1	Word64 Q6_P_mpyunac_RhRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx-=mpyu(Rs.L,Rt.H)	Word64 Q6_P_mpyunac_RlRh(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx-=mpyu(Rs.L,Rt.H):<<1	Word64 Q6_P_mpyunac_RlRh_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx-=mpyu(Rs.L,Rt.L)	Word64 Q6_P_mpyunac_RlRl(Word64 Rxx, Word32 Rs, Word32 Rt)	538
Rxx-=mpyu(Rs.L,Rt.L):<<1	Word64 Q6_P_mpyunac_RlRl_s1(Word64 Rxx, Word32 Rs, Word32 Rt)	538

mpyui

Rd=mpyui(Rs,Rt)	Word32 Q6_R_mpyui_RR(Word32 Rs, Word32 Rt)	517

mux

Rd=mux(Pu,#s8,#S8)	Word32 Q6_R_mux_pII(Byte Pu, Word32 Is8, Word32 IS8)	202
Rd=mux(Pu,#s8,Rs)	Word32 Q6_R_mux_pIR(Byte Pu, Word32 Is8, Word32 Rs)	202
Rd=mux(Pu,Rs,#s8)	Word32 Q6_R_mux_pRI(Byte Pu, Word32 Rs, Word32 Is8)	202
Rd=mux(Pu,Rs,Rt)	Word32 Q6_R_mux_pRR(Byte Pu, Word32 Rs, Word32 Rt)	202

N

neg

Rd=neg(Rs)	Word32 Q6_R_neg_R(Word32 Rs)	187
Rd=neg(Rs):sat	Word32 Q6_R_neg_R_sat(Word32 Rs)	385
Rdd=neg(Rss)	Word64 Q6_P_neg_P(Word64 Rss)	385

no mnemonic

Pd=Ps	Byte Q6_p_equals_p(Byte Ps)	233
Pd=Rs	Byte Q6_p_equals_R(Word32 Rs)	609
Rd=#s16	Word32 Q6_R_equals_I(Word32 Is16)	192
Rd=Ps	Word32 Q6_R_equals_p(Byte Ps)	609
Rd=Rs	Word32 Q6_R_equals_R(Word32 Rs)	194
Rdd=#s8	Word64 Q6_P_equals_I(Word32 Is8)	192
Rdd=Rss	Word64 Q6_P_equals_P(Word64 Rss)	194
Rx.H=#u16	Word32 Q6_Rh_equals_I(Word32 Rx, Word32 Iu16)	192
Rx.L=#u16	Word32 Q6_Rl_equals_I(Word32 Rx, Word32 Iu16)	192

normamt

Rd=normamt(Rs)	Word32 Q6_R_normamt_R(Word32 Rs)	440
Rd=normamt(Rss)	Word32 Q6_R_normamt_P(Word64 Rss)	440

not

Pd=not(Ps)	Byte Q6_p_not_p(Byte Ps)	233
Rd=not(Rs)	Word32 Q6_R_not_R(Word32 Rs)	185

Rdd=not(Rss)	Word64 Q6_P_not_P(Word64 Rss)	375

O

or

Pd=and(Ps,or(Pt,!Pu))	Byte Q6_p_and_or_ppnp(Byte Ps, Byte Pt, Byte Pu)	233
Pd=and(Ps,or(Pt,Pu))	Byte Q6_p_and_or_ppp(Byte Ps, Byte Pt, Byte Pu)	233
Pd=or(Ps,and(Pt,!Pu))	Byte Q6_p_or_and_ppnp(Byte Ps, Byte Pt, Byte Pu)	233
Pd=or(Ps,and(Pt,Pu))	Byte Q6_p_or_and_ppp(Byte Ps, Byte Pt, Byte Pu)	233
Pd=or(Ps,or(Pt,!Pu))	Byte Q6_p_or_or_ppnp(Byte Ps, Byte Pt, Byte Pu)	233
Pd=or(Ps,or(Pt,Pu))	Byte Q6_p_or_or_ppp(Byte Ps, Byte Pt, Byte Pu)	233
Pd=or(Pt,!Ps)	Byte Q6_p_or_pnp(Byte Pt, Byte Ps)	233
Pd=or(Pt,Ps)	Byte Q6_p_or_pp(Byte Pt, Byte Ps)	233
Rd=or(Rs,#s10)	Word32 Q6_R_or_RI(Word32 Rs, Word32 Is10)	185
Rd=or(Rs,Rt)	Word32 Q6_R_or_RR(Word32 Rs, Word32 Rt)	185
Rd=or(Rt,~Rs)	Word32 Q6_R_or_RnR(Word32 Rt, Word32 Rs)	185
Rdd=or(Rss,Rtt)	Word64 Q6_P_or_PP(Word64 Rss, Word64 Rtt)	375
Rdd=or(Rtt,~Rss)	Word64 Q6_P_or_PnP(Word64 Rtt, Word64 Rss)	375
Rx^=or(Rs,Rt)	Word32 Q6_R_orxacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)	378
Rx&=or(Rs,Rt)	Word32 Q6_R_orand_RR(Word32 Rx, Word32 Rs, Word32 Rt)	378
Rx=or(Ru,and(Rx,#s10))	Word32 Q6_R_or_and_RRI(Word32 Ru, Word32 Rx, Word32 Is10)	378
Rx|=or(Rs,#s10)	Word32 Q6_R_oror_RI(Word32 Rx, Word32 Rs, Word32 Is10)	379
Rx|=or(Rs,Rt)	Word32 Q6_R_oror_RR(Word32 Rx, Word32 Rs, Word32 Rt)	379

P

packhl

Rdd=packhl(Rs,Rt)	Word64 Q6_P_packhl_RR(Word32 Rs, Word32 Rt)	206

parity

Rd=parity(Rs,Rt)	Word32 Q6_R_parity_RR(Word32 Rs, Word32 Rt)	451
Rd=parity(Rss,Rtt)	Word32 Q6_R_parity_PP(Word64 Rss, Word64 Rtt)	451

pmpyw

Rdd=pmpyw(Rs,Rt)	Word64 Q6_P_pmpyw_RR(Word32 Rs, Word32 Rt)	542
Rxx^=pmpyw(Rs,Rt)	Word64 Q6_P_pmpywxacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)	542

popcount

Rd=popcount(Rss)	Word32 Q6_R_popcount_P(Word64 Rss)	441

R

rol

Rd=rol(Rs,#u5)	Word32 Q6_R_rol_RI(Word32 Rs, Word32 Iu5)	622
Rdd=rol(Rss,#u6)	Word64 Q6_P_rol_PI(Word64 Rss, Word32 Iu6)	622
Rx^=rol(Rs,#u5)	Word32 Q6_R_rolxacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	628
Rx&=rol(Rs,#u5)	Word32 Q6_R_roland_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	628
Rx+=rol(Rs,#u5)	Word32 Q6_R_rolacc_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	624
Rx-=rol(Rs,#u5)	Word32 Q6_R_rolnac_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	624
Rx|=rol(Rs,#u5)	Word32 Q6_R_rolor_RI(Word32 Rx, Word32 Rs, Word32 Iu5)	628
Rxx^=rol(Rss,#u6)	Word64 Q6_P_rolxacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	629
Rxx&=rol(Rss,#u6)	Word64 Q6_P_roland_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	629
Rxx+=rol(Rss,#u6)	Word64 Q6_P_rolacc_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	624
Rxx-=rol(Rss,#u6)	Word64 Q6_P_rolnac_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	624
Rxx|=rol(Rss,#u6)	Word64 Q6_P_rolor_PI(Word64 Rxx, Word64 Rss, Word32 Iu6)	629

round

Rd=round(Rs,#u5)	Word32 Q6_R_round_RI(Word32 Rs, Word32 Iu5)	386
Rd=round(Rs,#u5):sat	Word32 Q6_R_round_RI_sat(Word32 Rs, Word32 Iu5)	386
Rd=round(Rs,Rt)	Word32 Q6_R_round_RR(Word32 Rs, Word32 Rt)	386
Rd=round(Rs,Rt):sat	Word32 Q6_R_round_RR_sat(Word32 Rs, Word32 Rt)	386
Rd=round(Rss):sat	Word32 Q6_R_round_P_sat(Word64 Rss)	386

S

sat

Rd=sat(Rss)	Word32 Q6_R_sat_P(Word64 Rss)	576

satb

Rd=satb(Rs)	Word32 Q6_R_satb_R(Word32 Rs)	576

sath

Rd=sath(Rs)	Word32 Q6_R_sath_R(Word32 Rs)	576

satub

Rd=satub(Rs)	Word32 Q6_R_satub_R(Word32 Rs)	576

satuh

Rd=satuh(Rs)	Word32 Q6_R_satuh_R(Word32 Rs)	576

setbit

Rd=setbit(Rs,#u5)	Word32 Q6_R_setbit_RI(Word32 Rs, Word32 Iu5)	453
Rd=setbit(Rs,Rt)	Word32 Q6_R_setbit_RR(Word32 Rs, Word32 Rt)	453

sfadd

Rd=sfadd(Rs,Rt)	Word32 Q6_R_sfadd_RR(Word32 Rs, Word32 Rt)	495

sfclass

Pd=sfclass(Rs,#u5)	Byte Q6_p_sfclass_RI(Word32 Rs, Word32 Iu5)	496

sfcmp.eq

Pd=sfcmp.eq(Rs,Rt)	Byte Q6_p_sfcmp_eq_RR(Word32 Rs, Word32 Rt)	497

sfcmp.ge

Pd=sfcmp.ge(Rs,Rt)	Byte Q6_p_sfcmp_ge_RR(Word32 Rs, Word32 Rt)	497

sfcmp.gt

Pd=sfcmp.gt(Rs,Rt)	Byte Q6_p_sfcmp_gt_RR(Word32 Rs, Word32 Rt)	497

sfcmp.uo

Pd=sfcmp.uo(Rs,Rt)	Byte Q6_p_sfcmp_uo_RR(Word32 Rs, Word32 Rt)	497

sffixupd

Rd=sffixupd(Rs,Rt)	Word32 Q6_R_sffixupd_RR(Word32 Rs, Word32 Rt)	504

sffixupn

Rd=sffixupn(Rs,Rt)	Word32 Q6_R_sffixupn_RR(Word32 Rs, Word32 Rt)	504

sffixupr

Rd=sffixupr(Rs)	Word32 Q6_R_sffixupr_R(Word32 Rs)	504

sfmake

Rd=sfmake(#u10):neg	Word32 Q6_R_sfmake_I_neg(Word32 Iu10)	510
Rd=sfmake(#u10):pos	Word32 Q6_R_sfmake_I_pos(Word32 Iu10)	510

sfmax

Rd=sfmax(Rs,Rt)	Word32 Q6_R_sfmax_RR(Word32 Rs, Word32 Rt)	511

sfmin

Rd=sfmin(Rs,Rt)	Word32 Q6_R_sfmin_RR(Word32 Rs, Word32 Rt)	512

sfmpy

Rd=sfmpy(Rs,Rt)	Word32 Q6_R_sfmpy_RR(Word32 Rs, Word32 Rt)	513
Rx+=sfmpy(Rs,Rt,Pu):scale	Word32 Q6_R_sfmpyacc_RRp_scale(Word32 Rx, Word32 Rs, Word32 Rt, Byte Pu)	506
Rx+=sfmpy(Rs,Rt)	Word32 Q6_R_sfmpyacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)	505
Rx+=sfmpy(Rs,Rt):lib	Word32 Q6_R_sfmpyacc_RR_lib(Word32 Rx, Word32 Rs, Word32 Rt)	508
Rx-=sfmpy(Rs,Rt)	Word32 Q6_R_sfmpynac_RR(Word32 Rx, Word32 Rs, Word32 Rt)	505
Rx-=sfmpy(Rs,Rt):lib	Word32 Q6_R_sfmpynac_RR_lib(Word32 Rx, Word32 Rs, Word32 Rt)	508

sfsub

Rd=sfsub(Rs,Rt)	Word32 Q6_R_sfsub_RR(Word32 Rs, Word32 Rt)	515

shuffeb

Rdd=shuffeb(Rss,Rtt)	Word64 Q6_P_shuffeb_PP(Word64 Rss, Word64 Rtt)	589

shuffeh

Rdd=shuffeh(Rss,Rtt)	Word64 Q6_P_shuffeh_PP(Word64 Rss, Word64 Rtt)	589

shuffob

Rdd=shuffob(Rtt,Rss)	Word64 Q6_P_shuffob_PP(Word64 Rtt, Word64 Rss)	589

shuffoh

Rdd=shuffoh(Rtt,Rss)	Word64 Q6_P_shuffoh_PP(Word64 Rtt, Word64 Rss)	589

sub

Rd=add(Rs,sub(#s6,Ru))	Word32 Q6_R_add_sub_RIR(Word32 Rs, Word32 Is6, Word32 Ru)	367
Rd=sub(#s10,Rs)	Word32 Q6_R_sub_IR(Word32 Is10, Word32 Rs)	189
Rd=sub(Rt,Rs)	Word32 Q6_R_sub_RR(Word32 Rt, Word32 Rs)	189
Rd=sub(Rt,Rs):sat	Word32 Q6_R_sub_RR_sat(Word32 Rt, Word32 Rs)	189
Rd=sub(Rt.H,Rs.H):<<16	Word32 Q6_R_sub_RhRh_s16(Word32 Rt, Word32 Rs)	391
Rd=sub(Rt.H,Rs.H):sat:<<16	Word32 Q6_R_sub_RhRh_sat_s16(Word32 Rt, Word32 Rs)	391
Rd=sub(Rt.H,Rs.L):<<16	Word32 Q6_R_sub_RhRl_s16(Word32 Rt, Word32 Rs)	391
Rd=sub(Rt.H,Rs.L):sat:<<16	Word32 Q6_R_sub_RhRl_sat_s16(Word32 Rt, Word32 Rs)	391
Rd=sub(Rt.L,Rs.H)	Word32 Q6_R_sub_RlRh(Word32 Rt, Word32 Rs)	391
Rd=sub(Rt.L,Rs.H):<<16	Word32 Q6_R_sub_RlRh_s16(Word32 Rt, Word32 Rs)	391
Rd=sub(Rt.L,Rs.H):sat	Word32 Q6_R_sub_RlRh_sat(Word32 Rt, Word32 Rs)	391
Rd=sub(Rt.L,Rs.H):sat:<<16	Word32 Q6_R_sub_RlRh_sat_s16(Word32 Rt, Word32 Rs)	391
Rd=sub(Rt.L,Rs.L)	Word32 Q6_R_sub_RlRl(Word32 Rt, Word32 Rs)	391
Rd=sub(Rt.L,Rs.L):<<16	Word32 Q6_R_sub_RlRl_s16(Word32 Rt, Word32 Rs)	391
Rd=sub(Rt.L,Rs.L):sat	Word32 Q6_R_sub_RlRl_sat(Word32 Rt, Word32 Rs)	391
Rd=sub(Rt.L,Rs.L):sat:<<16	Word32 Q6_R_sub_RlRl_sat_s16(Word32 Rt, Word32 Rs)	391
Rdd=sub(Rtt,Rss)	Word64 Q6_P_sub_PP(Word64 Rtt, Word64 Rss)	388
Rx+=sub(Rt,Rs)	Word32 Q6_R_subacc_RR(Word32 Rx, Word32 Rt, Word32 Rs)	389

swiz

Rd=swiz(Rs)	Word32 Q6_R_swiz_R(Word32 Rs)	578

sxtb

Rd=sxtb(Rs)	Word32 Q6_R_sxtb_R(Word32 Rs)	191

sxth

Rd=sxth(Rs)	Word32 Q6_R_sxth_R(Word32 Rs)	191

sxtw

Rdd=sxtw(Rs)	Word64 Q6_P_sxtw_R(Word32 Rs)	393

T

tableidxb

Rx=tableidxb(Rs,#u4,#U5)	Word32 Q6_R_tableidxb_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5)	459

tableidxd

Rx=tableidxd(Rs,#u4,#U5)	Word32 Q6_R_tableidxd_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5)	459

tableidxh

Rx=tableidxh(Rs,#u4,#U5)	Word32 Q6_R_tableidxh_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5)	459

tableidxw

Rx=tableidxw(Rs,#u4,#U5)	Word32 Q6_R_tableidxw_RII(Word32 Rx, Word32 Rs, Word32 Iu4, Word32 IU5)	459

tlbmatch

Pd=tlbmatch(Rss,Rt)	Byte Q6_p_tlbmatch_PR(Word64 Rss, Word32 Rt)	608

togglebit

Rd=togglebit(Rs,#u5)	Word32 Q6_R_togglebit_RI(Word32 Rs, Word32 Iu5)	453
Rd=togglebit(Rs,Rt)	Word32 Q6_R_togglebit_RR(Word32 Rs, Word32 Rt)	453

tstbit

Pd=!tstbit(Rs,#u5)	Byte Q6_p_not_tstbit_RI(Word32 Rs, Word32 Iu5)	610
Pd=!tstbit(Rs,Rt)	Byte Q6_p_not_tstbit_RR(Word32 Rs, Word32 Rt)	610
Pd=tstbit(Rs,#u5)	Byte Q6_p_tstbit_RI(Word32 Rs, Word32 Iu5)	610
Pd=tstbit(Rs,Rt)	Byte Q6_p_tstbit_RR(Word32 Rs, Word32 Rt)	610

V

vabsdiffb

Rdd=vabsdiffb(Rtt,Rss)	Word64 Q6_P_vabsdiffb_PP(Word64 Rtt, Word64 Rss)	396

vabsdiffh

Rdd=vabsdiffh(Rtt,Rss)	Word64 Q6_P_vabsdiffh_PP(Word64 Rtt, Word64 Rss)	397

vabsdiffub

Rdd=vabsdiffub(Rtt,Rss)	Word64 Q6_P_vabsdiffub_PP(Word64 Rtt, Word64 Rss)	396

vabsdiffw

Rdd=vabsdiffw(Rtt,Rss)	Word64 Q6_P_vabsdiffw_PP(Word64 Rtt, Word64 Rss)	398

vabsh

Rdd=vabsh(Rss)	Word64 Q6_P_vabsh_P(Word64 Rss)	394

Rdd=vabsh(Rss):sat	Word64 Q6_P_vabsh_P_sat(Word64 Rss)	394

vabsw

Rdd=vabsw(Rss)	Word64 Q6_P_vabsw_P(Word64 Rss)	395
Rdd=vabsw(Rss):sat	Word64 Q6_P_vabsw_P_sat(Word64 Rss)	395

vaddb

Rdd=vaddb(Rss,Rtt)	Word64 Q6_P_vaddb_PP(Word64 Rss, Word64 Rtt)	409

vaddh

Rd=vaddh(Rs,Rt)	Word32 Q6_R_vaddh_RR(Word32 Rs, Word32 Rt)	195
Rd=vaddh(Rs,Rt):sat	Word32 Q6_R_vaddh_RR_sat(Word32 Rs, Word32 Rt)	195
Rdd=vaddh(Rss,Rtt)	Word64 Q6_P_vaddh_PP(Word64 Rss, Word64 Rtt)	402
Rdd=vaddh(Rss,Rtt):sat	Word64 Q6_P_vaddh_PP_sat(Word64 Rss, Word64 Rtt)	402

vaddhub

Rd=vaddhub(Rss,Rtt):sat	Word32 Q6_R_vaddhub_PP_sat(Word64 Rss, Word64 Rtt)	404

vaddub

Rdd=vaddub(Rss,Rtt)	Word64 Q6_P_vaddub_PP(Word64 Rss, Word64 Rtt)	409
Rdd=vaddub(Rss,Rtt):sat	Word64 Q6_P_vaddub_PP_sat(Word64 Rss, Word64 Rtt)	409

vadduh

Rd=vadduh(Rs,Rt):sat	Word32 Q6_R_vadduh_RR_sat(Word32 Rs, Word32 Rt)	195
Rdd=vadduh(Rss,Rtt):sat	Word64 Q6_P_vadduh_PP_sat(Word64 Rss, Word64 Rtt)	402

vaddw

Rdd=vaddw(Rss,Rtt)	Word64 Q6_P_vaddw_PP(Word64 Rss, Word64 Rtt)	410
Rdd=vaddw(Rss,Rtt):sat	Word64 Q6_P_vaddw_PP_sat(Word64 Rss, Word64 Rtt)	410

valignb

Rdd=valignb(Rtt,Rss,#u3)	Word64 Q6_P_valignb_PPI(Word64 Rtt, Word64 Rss, Word32 Iu3)	579
Rdd=valignb(Rtt,Rss,Pu)	Word64 Q6_P_valignb_PPp(Word64 Rtt, Word64 Rss, Byte Pu)	579

vaslh

Rdd=vaslh(Rss,#u4)	Word64 Q6_P_vaslh_PI(Word64 Rss, Word32 Iu4)	646
Rdd=vaslh(Rss,Rt)	Word64 Q6_P_vaslh_PR(Word64 Rss, Word32 Rt)	653

vaslw

Rdd=vaslw(Rss,#u5)	Word64 Q6_P_vaslw_PI(Word64 Rss, Word32 Iu5)	654
Rdd=vaslw(Rss,Rt)	Word64 Q6_P_vaslw_PR(Word64 Rss, Word32 Rt)	657

vasrh

Rdd=vasrh(Rss,#u4)	Word64 Q6_P_vasrh_PI(Word64 Rss, Word32 Iu4)	646
Rdd=vasrh(Rss,#u4):rnd	Word64 Q6_P_vasrh_PI_rnd(Word64 Rss, Word32 Iu4)	648
Rdd=vasrh(Rss,Rt)	Word64 Q6_P_vasrh_PR(Word64 Rss, Word32 Rt)	653

vasrhub

Rd=vasrhub(Rss,#u4):rnd:sat	Word32 Q6_R_vasrhub_PI_rnd_sat(Word64 Rss, Word32 Iu4)	651
Rd=vasrhub(Rss,#u4):sat	Word32 Q6_R_vasrhub_PI_sat(Word64 Rss, Word32 Iu4)	651

vasrw

Rd=vasrw(Rss,#u5)	Word32 Q6_R_vasrw_PI(Word64 Rss, Word32 Iu5)	658
Rd=vasrw(Rss,Rt)	Word32 Q6_R_vasrw_PR(Word64 Rss, Word32 Rt)	658

Rdd=vasrw(Rss,#u5)	Word64 Q6_P_vasrw_PI(Word64 Rss, Word32 Iu5)	654
Rdd=vasrw(Rss,Rt)	Word64 Q6_P_vasrw_PR(Word64 Rss, Word32 Rt)	657

vavgh

Rd=vavgh(Rs,Rt)	Word32 Q6_R_vavgh_RR(Word32 Rs, Word32 Rt)	196
Rd=vavgh(Rs,Rt):rnd	Word32 Q6_R_vavgh_RR_rnd(Word32 Rs, Word32 Rt)	196
Rdd=vavgh(Rss,Rtt)	Word64 Q6_P_vavgh_PP(Word64 Rss, Word64 Rtt)	412
Rdd=vavgh(Rss,Rtt):crnd	Word64 Q6_P_vavgh_PP_crnd(Word64 Rss, Word64 Rtt)	412
Rdd=vavgh(Rss,Rtt):rnd	Word64 Q6_P_vavgh_PP_rnd(Word64 Rss, Word64 Rtt)	412

vavgub

Rdd=vavgub(Rss,Rtt)	Word64 Q6_P_vavgub_PP(Word64 Rss, Word64 Rtt)	413
Rdd=vavgub(Rss,Rtt):rnd	Word64 Q6_P_vavgub_PP_rnd(Word64 Rss, Word64 Rtt)	413

vavguh

Rdd=vavguh(Rss,Rtt)	Word64 Q6_P_vavguh_PP(Word64 Rss, Word64 Rtt)	412
Rdd=vavguh(Rss,Rtt):rnd	Word64 Q6_P_vavguh_PP_rnd(Word64 Rss, Word64 Rtt)	412

vavguw

Rdd=vavguw(Rss,Rtt)	Word64 Q6_P_vavguw_PP(Word64 Rss, Word64 Rtt)	415
Rdd=vavguw(Rss,Rtt):rnd	Word64 Q6_P_vavguw_PP_rnd(Word64 Rss, Word64 Rtt)	415

vavgw

Rdd=vavgw(Rss,Rtt)	Word64 Q6_P_vavgw_PP(Word64 Rss, Word64 Rtt)	415
Rdd=vavgw(Rss,Rtt):crnd	Word64 Q6_P_vavgw_PP_crnd(Word64 Rss, Word64 Rtt)	415
Rdd=vavgw(Rss,Rtt):rnd	Word64 Q6_P_vavgw_PP_rnd(Word64 Rss, Word64 Rtt)	415

vcmpb.eq

Pd=any8(vcmpb.eq(Rss,Rtt))	Byte Q6_p_any8_vcmpb_eq_PP(Word64 Rss, Word64 Rtt)	613
Pd=vcmpb.eq(Rss,#u8)	Byte Q6_p_vcmpb_eq_PI(Word64 Rss, Word32 Iu8)	615
Pd=vcmpb.eq(Rss,Rtt)	Byte Q6_p_vcmpb_eq_PP(Word64 Rss, Word64 Rtt)	615

vcmpb.gt

Pd=vcmpb.gt(Rss,#s8)	Byte Q6_p_vcmpb_gt_PI(Word64 Rss, Word32 Is8)	615
Pd=vcmpb.gt(Rss,Rtt)	Byte Q6_p_vcmpb_gt_PP(Word64 Rss, Word64 Rtt)	615

vcmpb.gtu

Pd=vcmpb.gtu(Rss,#u7)	Byte Q6_p_vcmpb_gtu_PI(Word64 Rss, Word32 Iu7)	615
Pd=vcmpb.gtu(Rss,Rtt)	Byte Q6_p_vcmpb_gtu_PP(Word64 Rss, Word64 Rtt)	615

vcmph.eq

Pd=vcmph.eq(Rss,#s8)	Byte Q6_p_vcmph_eq_PI(Word64 Rss, Word32 Is8)	612
Pd=vcmph.eq(Rss,Rtt)	Byte Q6_p_vcmph_eq_PP(Word64 Rss, Word64 Rtt)	612

vcmph.gt

Pd=vcmph.gt(Rss,#s8)	Byte Q6_p_vcmph_gt_PI(Word64 Rss, Word32 Is8)	612
Pd=vcmph.gt(Rss,Rtt)	Byte Q6_p_vcmph_gt_PP(Word64 Rss, Word64 Rtt)	612

vcmph.gtu

Pd=vcmph.gtu(Rss,#u7)	Byte Q6_p_vcmph_gtu_PI(Word64 Rss, Word32 Iu7)	612
Pd=vcmph.gtu(Rss,Rtt)	Byte Q6_p_vcmph_gtu_PP(Word64 Rss, Word64 Rtt)	612

vcmpw.eq

Pd=vcmpw.eq(Rss,#s8)	Byte Q6_p_vcmpw_eq_PI(Word64 Rss, Word32 Is8)	616

Pd=vcmpw.eq(Rss,Rtt)	Byte Q6_p_vcmpw_eq_PP(Word64 Rss, Word64 Rtt)	616
vcmpw.gt		
Pd=vcmpw.gt(Rss,#s8)	Byte Q6_p_vcmpw_gt_PI(Word64 Rss, Word32 Is8)	616
Pd=vcmpw.gt(Rss,Rtt)	Byte Q6_p_vcmpw_gt_PP(Word64 Rss, Word64 Rtt)	616
vcmpw.gtu		
Pd=vcmpw.gtu(Rss,#u7)	Byte Q6_p_vcmpw_gtu_PI(Word64 Rss, Word32 Iu7)	616
Pd=vcmpw.gtu(Rss,Rtt)	Byte Q6_p_vcmpw_gtu_PP(Word64 Rss, Word64 Rtt)	616
vcmpyi		
Rdd=vcmpyi(Rss,Rtt):<<1:sat	Word64 Q6_P_vcmpyi_PP_s1_sat(Word64 Rss, Word64 Rtt)	477
Rdd=vcmpyi(Rss,Rtt):sat	Word64 Q6_P_vcmpyi_PP_sat(Word64 Rss, Word64 Rtt)	477
Rxx+=vcmpyi(Rss,Rtt):sat	Word64 Q6_P_vcmpyiacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	477
vcmpyr		
Rdd=vcmpyr(Rss,Rtt):<<1:sat	Word64 Q6_P_vcmpyr_PP_s1_sat(Word64 Rss, Word64 Rtt)	477
Rdd=vcmpyr(Rss,Rtt):sat	Word64 Q6_P_vcmpyr_PP_sat(Word64 Rss, Word64 Rtt)	477
Rxx+=vcmpyr(Rss,Rtt):sat	Word64 Q6_P_vcmpyracc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	477
vcnegh		
Rdd=vcnegh(Rss,Rt)	Word64 Q6_P_vcnegh_PR(Word64 Rss, Word32 Rt)	416
vconj		
Rdd=vconj(Rss):sat	Word64 Q6_P_vconj_P_sat(Word64 Rss)	478
vcrotate		
Rdd=vcrotate(Rss,Rt)	Word64 Q6_P_vcrotate_PR(Word64 Rss, Word32 Rt)	480
vdmpy		
Rd=vdmpy(Rss,Rtt):<<1:rnd:sat	Word32 Q6_R_vdmpy_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)	554
Rd=vdmpy(Rss,Rtt):rnd:sat	Word32 Q6_R_vdmpy_PP_rnd_sat(Word64 Rss, Word64 Rtt)	554
Rdd=vdmpy(Rss,Rtt):<<1:sat	Word64 Q6_P_vdmpy_PP_s1_sat(Word64 Rss, Word64 Rtt)	552
Rdd=vdmpy(Rss,Rtt):sat	Word64 Q6_P_vdmpy_PP_sat(Word64 Rss, Word64 Rtt)	552
Rxx+=vdmpy(Rss,Rtt):<<1:sat	Word64 Q6_P_vdmpyacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	552
Rxx+=vdmpy(Rss,Rtt):sat	Word64 Q6_P_vdmpyacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	552
vdmpybsu		
Rdd=vdmpybsu(Rss,Rtt):sat	Word64 Q6_P_vdmpybsu_PP_sat(Word64 Rss, Word64 Rtt)	558
Rxx+=vdmpybsu(Rss,Rtt):sat	Word64 Q6_P_vdmpybsuacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	558
vitpack		
Rd=vitpack(Ps,Pt)	Word32 Q6_R_vitpack_pp(Byte Ps, Byte Pt)	618
vlslh		
Rdd=vlslh(Rss,Rt)	Word64 Q6_P_vlslh_PR(Word64 Rss, Word32 Rt)	653
vlslw		
Rdd=vlslw(Rss,Rt)	Word64 Q6_P_vlslw_PR(Word64 Rss, Word32 Rt)	657
vlsrh		
Rdd=vlsrh(Rss,#u4)	Word64 Q6_P_vlsrh_PI(Word64 Rss, Word32 Iu4)	646
Rdd=vlsrh(Rss,Rt)	Word64 Q6_P_vlsrh_PR(Word64 Rss, Word32 Rt)	653

vlsrw

Rdd=vlsrw(Rss,#u5)	Word64 Q6_P_vlsrw_PI(Word64 Rss, Word32 Iu5)	654
Rdd=vlsrw(Rss,Rt)	Word64 Q6_P_vlsrw_PR(Word64 Rss, Word32 Rt)	657

vmaxb

Rdd=vmaxb(Rtt,Rss)	Word64 Q6_P_vmaxb_PP(Word64 Rtt, Word64 Rss)	418

vmaxh

Rdd=vmaxh(Rtt,Rss)	Word64 Q6_P_vmaxh_PP(Word64 Rtt, Word64 Rss)	419

vmaxub

Rdd=vmaxub(Rtt,Rss)	Word64 Q6_P_vmaxub_PP(Word64 Rtt, Word64 Rss)	418

vmaxuh

Rdd=vmaxuh(Rtt,Rss)	Word64 Q6_P_vmaxuh_PP(Word64 Rtt, Word64 Rss)	419

vmaxuw

Rdd=vmaxuw(Rtt,Rss)	Word64 Q6_P_vmaxuw_PP(Word64 Rtt, Word64 Rss)	424

vmaxw

Rdd=vmaxw(Rtt,Rss)	Word64 Q6_P_vmaxw_PP(Word64 Rtt, Word64 Rss)	424

vminb

Rdd=vminb(Rtt,Rss)	Word64 Q6_P_vminb_PP(Word64 Rtt, Word64 Rss)	425

vminh

Rdd=vminh(Rtt,Rss)	Word64 Q6_P_vminh_PP(Word64 Rtt, Word64 Rss)	427

vminub

Rdd=vminub(Rtt,Rss)	Word64 Q6_P_vminub_PP(Word64 Rtt, Word64 Rss)	425

vminuh

Rdd=vminuh(Rtt,Rss)	Word64 Q6_P_vminuh_PP(Word64 Rtt, Word64 Rss)	427

vminuw

Rdd=vminuw(Rtt,Rss)	Word64 Q6_P_vminuw_PP(Word64 Rtt, Word64 Rss)	432

vminw

Rdd=vminw(Rtt,Rss)	Word64 Q6_P_vminw_PP(Word64 Rtt, Word64 Rss)	432

vmpybsu

Rdd=vmpybsu(Rs,Rt)	Word64 Q6_P_vmpybsu_RR(Word32 Rs, Word32 Rt)	570
Rxx+=vmpybsu(Rs,Rt)	Word64 Q6_P_vmpybsuacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)	570

vmpybu

Rdd=vmpybu(Rs,Rt)	Word64 Q6_P_vmpybu_RR(Word32 Rs, Word32 Rt)	570
Rxx+=vmpybu(Rs,Rt)	Word64 Q6_P_vmpybuacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)	570

vmpyeh

Rdd=vmpyeh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpyeh_PP_s1_sat(Word64 Rss, Word64 Rtt)	560
Rdd=vmpyeh(Rss,Rtt):sat	Word64 Q6_P_vmpyeh_PP_sat(Word64 Rss, Word64 Rtt)	560
Rxx+=vmpyeh(Rss,Rtt)	Word64 Q6_P_vmpyehacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)	560
Rxx+=vmpyeh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpyehacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	560
Rxx+=vmpyeh(Rss,Rtt):sat	Word64 Q6_P_vmpyehacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	560

vmpyh

Rd=vmpyh(Rs,Rt):<<1:rnd:sat	Word32 Q6_R_vmpyh_RR_s1_rnd_sat(Word32 Rs, Word32 Rt)	564
Rd=vmpyh(Rs,Rt):rnd:sat	Word32 Q6_R_vmpyh_RR_rnd_sat(Word32 Rs, Word32 Rt)	564
Rdd=vmpyh(Rs,Rt):<<1:sat	Word64 Q6_P_vmpyh_RR_s1_sat(Word32 Rs, Word32 Rt)	562
Rdd=vmpyh(Rs,Rt):sat	Word64 Q6_P_vmpyh_RR_sat(Word32 Rs, Word32 Rt)	562
Rxx+=vmpyh(Rs,Rt)	Word64 Q6_P_vmpyhacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)	562
Rxx+=vmpyh(Rs,Rt):<<1:sat	Word64 Q6_P_vmpyhacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)	562
Rxx+=vmpyh(Rs,Rt):sat	Word64 Q6_P_vmpyhacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)	562

vmpyhsu

Rdd=vmpyhsu(Rs,Rt):<<1:sat	Word64 Q6_P_vmpyhsu_RR_s1_sat(Word32 Rs, Word32 Rt)	565
Rdd=vmpyhsu(Rs,Rt):sat	Word64 Q6_P_vmpyhsu_RR_sat(Word32 Rs, Word32 Rt)	565
Rxx+=vmpyhsu(Rs,Rt):<<1:sat	Word64 Q6_P_vmpyhsuacc_RR_s1_sat(Word64 Rxx, Word32 Rs, Word32 Rt)	565
Rxx+=vmpyhsu(Rs,Rt):sat	Word64 Q6_P_vmpyhsuacc_RR_sat(Word64 Rxx, Word32 Rs, Word32 Rt)	565

vmpyweh

Rdd=vmpyweh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpyweh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)	522
Rdd=vmpyweh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpyweh_PP_s1_sat(Word64 Rss, Word64 Rtt)	522
Rdd=vmpyweh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpyweh_PP_rnd_sat(Word64 Rss, Word64 Rtt)	522
Rdd=vmpyweh(Rss,Rtt):sat	Word64 Q6_P_vmpyweh_PP_sat(Word64 Rss, Word64 Rtt)	522
Rxx+=vmpyweh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpywehacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	522
Rxx+=vmpyweh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpywehacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	522
Rxx+=vmpyweh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpywehacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	522
Rxx+=vmpyweh(Rss,Rtt):sat	Word64 Q6_P_vmpywehacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	522

vmpyweuh

Rdd=vmpyweuh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpyweuh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)	527
Rdd=vmpyweuh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpyweuh_PP_s1_sat(Word64 Rss, Word64 Rtt)	527
Rdd=vmpyweuh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpyweuh_PP_rnd_sat(Word64 Rss, Word64 Rtt)	527
Rdd=vmpyweuh(Rss,Rtt):sat	Word64 Q6_P_vmpyweuh_PP_sat(Word64 Rss, Word64 Rtt)	527
Rxx+=vmpyweuh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpyweuhacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	527
Rxx+=vmpyweuh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpyweuhacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	527
Rxx+=vmpyweuh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpyweuhacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	527
Rxx+=vmpyweuh(Rss,Rtt):sat	Word64 Q6_P_vmpyweuhacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	527

vmpywoh

Rdd=vmpywoh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpywoh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)	522
Rdd=vmpywoh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpywoh_PP_s1_sat(Word64 Rss, Word64 Rtt)	522
Rdd=vmpywoh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpywoh_PP_rnd_sat(Word64 Rss, Word64 Rtt)	522
Rdd=vmpywoh(Rss,Rtt):sat	Word64 Q6_P_vmpywoh_PP_sat(Word64 Rss, Word64 Rtt)	522
Rxx+=vmpywoh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpywohacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	522
Rxx+=vmpywoh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpywohacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	522
Rxx+=vmpywoh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpywohacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	522
Rxx+=vmpywoh(Rss,Rtt):sat	Word64 Q6_P_vmpywohacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	522

vmpywouh

Rdd=vmpywouh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpywouh_PP_s1_rnd_sat(Word64 Rss, Word64 Rtt)	527
Rdd=vmpywouh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpywouh_PP_s1_sat(Word64 Rss, Word64 Rtt)	527
Rdd=vmpywouh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpywouh_PP_rnd_sat(Word64 Rss, Word64 Rtt)	527
Rdd=vmpywouh(Rss,Rtt):sat	Word64 Q6_P_vmpywouh_PP_sat(Word64 Rss, Word64 Rtt)	527
Rxx+=vmpywouh(Rss,Rtt):<<1:rnd:sat	Word64 Q6_P_vmpywouhacc_PP_s1_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	527
Rxx+=vmpywouh(Rss,Rtt):<<1:sat	Word64 Q6_P_vmpywouhacc_PP_s1_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	527
Rxx+=vmpywouh(Rss,Rtt):rnd:sat	Word64 Q6_P_vmpywouhacc_PP_rnd_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	527
Rxx+=vmpywouh(Rss,Rtt):sat	Word64 Q6_P_vmpywouhacc_PP_sat(Word64 Rxx, Word64 Rss, Word64 Rtt)	527

vmux

Rdd=vmux(Pu,Rss,Rtt)	Word64 Q6_P_vmux_pPP(Byte Pu, Word64 Rss, Word64 Rtt)	619

vnavgh

Rd=vnavgh(Rt,Rs)	Word32 Q6_R_vnavgh_RR(Word32 Rt, Word32 Rs)	196
Rdd=vnavgh(Rtt,Rss)	Word64 Q6_P_vnavgh_PP(Word64 Rtt, Word64 Rss)	412
Rdd=vnavgh(Rtt,Rss):crnd:sat	Word64 Q6_P_vnavgh_PP_crnd_sat(Word64 Rtt, Word64 Rss)	412
Rdd=vnavgh(Rtt,Rss):rnd:sat	Word64 Q6_P_vnavgh_PP_rnd_sat(Word64 Rtt, Word64 Rss)	412

vnavgw

Rdd=vnavgw(Rtt,Rss)	Word64 Q6_P_vnavgw_PP(Word64 Rtt, Word64 Rss)	415
Rdd=vnavgw(Rtt,Rss):crnd:sat	Word64 Q6_P_vnavgw_PP_crnd_sat(Word64 Rtt, Word64 Rss)	415
Rdd=vnavgw(Rtt,Rss):rnd:sat	Word64 Q6_P_vnavgw_PP_rnd_sat(Word64 Rtt, Word64 Rss)	415

vpmpyh

Rdd=vpmpyh(Rs,Rt)	Word64 Q6_P_vpmpyh_RR(Word32 Rs, Word32 Rt)	573
Rxx^=vpmpyh(Rs,Rt)	Word64 Q6_P_vpmpyhxacc_RR(Word64 Rxx, Word32 Rs, Word32 Rt)	573

vraddh

Rd=vraddh(Rss,Rtt)	Word32 Q6_R_vraddh_PP(Word64 Rss, Word64 Rtt)	407

vraddub

Rdd=vraddub(Rss,Rtt)	Word64 Q6_P_vraddub_PP(Word64 Rss, Word64 Rtt)	405
Rxx+=vraddub(Rss,Rtt)	Word64 Q6_P_vraddubacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)	405

vradduh

Rd=vradduh(Rss,Rtt)	Word32 Q6_R_vradduh_PP(Word64 Rss, Word64 Rtt)	407

vrcmpyi

Rdd=vrcmpyi(Rss,Rtt)	Word64 Q6_P_vrcmpyi_PP(Word64 Rss, Word64 Rtt)	482
Rdd=vrcmpyi(Rss,Rtt*)	Word64 Q6_P_vrcmpyi_PP_conj(Word64 Rss, Word64 Rtt)	482
Rxx+=vrcmpyi(Rss,Rtt)	Word64 Q6_P_vrcmpyiacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)	482
Rxx+=vrcmpyi(Rss,Rtt*)	Word64 Q6_P_vrcmpyiacc_PP_conj(Word64 Rxx, Word64 Rss, Word64 Rtt)	482

vrcmpyr

Rdd=vrcmpyr(Rss,Rtt)	Word64 Q6_P_vrcmpyr_PP(Word64 Rss, Word64 Rtt)	482
Rdd=vrcmpyr(Rss,Rtt*)	Word64 Q6_P_vrcmpyr_PP_conj(Word64 Rss, Word64 Rtt)	482
Rxx+=vrcmpyr(Rss,Rtt)	Word64 Q6_P_vrcmpyracc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)	482
Rxx+=vrcmpyr(Rss,Rtt*)	Word64 Q6_P_vrcmpyracc_PP_conj(Word64 Rxx, Word64 Rss, Word64 Rtt)	482

vrcmpys

Rd=vrcmpys(Rss,Rt):<<1:rnd:sat	Word32 Q6_R_vrcmpys_PR_s1_rnd_sat(Word64 Rss, Word32 Rt)	490
Rdd=vrcmpys(Rss,Rt):<<1:sat	Word64 Q6_P_vrcmpys_PR_s1_sat(Word64 Rss, Word32 Rt)	486
Rxx+=vrcmpys(Rss,Rt):<<1:sat	Word64 Q6_P_vrcmpysacc_PR_s1_sat(Word64 Rxx, Word64 Rss, Word32 Rt)	486

vrcnegh

Rxx+=vrcnegh(Rss,Rt)	Word64 Q6_P_vrcneghacc_PR(Word64 Rxx, Word64 Rss, Word32 Rt)	416

vrcrotate

Rdd=vrcrotate(Rss,Rt,#u2)	Word64 Q6_P_vrcrotate_PRI(Word64 Rss, Word32 Rt, Word32 Iu2)	493
Rxx+=vrcrotate(Rss,Rt,#u2)	Word64 Q6_P_vrcrotateacc_PRI(Word64 Rxx, Word64 Rss, Word32 Rt, Word32 Iu2)	493

vrmaxh

Rxx=vrmaxh(Rss,Ru)	Word64 Q6_P_vrmaxh_PR(Word64 Rxx, Word64 Rss, Word32 Ru)	420

vrmaxuh

Rxx=vrmaxuh(Rss,Ru)	Word64 Q6_P_vrmaxuh_PR(Word64 Rxx, Word64 Rss, Word32 Ru)	420

vrmaxuw

Rxx=vrmaxuw(Rss,Ru)	Word64 Q6_P_vrmaxuw_PR(Word64 Rxx, Word64 Rss, Word32 Ru)	422

vrmaxw

Rxx=vrmaxw(Rss,Ru)	Word64 Q6_P_vrmaxw_PR(Word64 Rxx, Word64 Rss, Word32 Ru)	422

vrminh

Rxx=vrminh(Rss,Ru)	Word64 Q6_P_vrminh_PR(Word64 Rxx, Word64 Rss, Word32 Ru)	428

vrminuh

Rxx=vrminuh(Rss,Ru)	Word64 Q6_P_vrminuh_PR(Word64 Rxx, Word64 Rss, Word32 Ru)	428

vrminuw

Rxx=vrminuw(Rss,Ru)	Word64 Q6_P_vrminuw_PR(Word64 Rxx, Word64 Rss, Word32 Ru)	430

vrminw

Rxx=vrminw(Rss,Ru)	Word64 Q6_P_vrminw_PR(Word64 Rxx, Word64 Rss, Word32 Ru)	430

vrmpybsu

Rdd=vrmpybsu(Rss,Rtt)	Word64 Q6_P_vrmpybsu_PP(Word64 Rss, Word64 Rtt)	556
Rxx+=vrmpybsu(Rss,Rtt)	Word64 Q6_P_vrmpybsuacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)	556

vrmpybu

Rdd=vrmpybu(Rss,Rtt)	Word64 Q6_P_vrmpybu_PP(Word64 Rss, Word64 Rtt)	556
Rxx+=vrmpybu(Rss,Rtt)	Word64 Q6_P_vrmpybuacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)	556

vrmpyh

Rdd=vrmpyh(Rss,Rtt)	Word64 Q6_P_vrmpyh_PP(Word64 Rss, Word64 Rtt)	567
Rxx+=vrmpyh(Rss,Rtt)	Word64 Q6_P_vrmpyhacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)	567

vrmpyweh

Rdd=vrmpyweh(Rss,Rtt)	Word64 Q6_P_vrmpyweh_PP(Word64 Rss, Word64 Rtt)	544
Rdd=vrmpyweh(Rss,Rtt):<<1	Word64 Q6_P_vrmpyweh_PP_s1(Word64 Rss, Word64 Rtt)	544
Rxx+=vrmpyweh(Rss,Rtt)	Word64 Q6_P_vrmpywehacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)	544
Rxx+=vrmpyweh(Rss,Rtt):<<1	Word64 Q6_P_vrmpywehacc_PP_s1(Word64 Rxx, Word64 Rss, Word64 Rtt)	544

vrmpywoh

Rdd=vrmpywoh(Rss,Rtt)	Word64 Q6_P_vrmpywoh_PP(Word64 Rss, Word64 Rtt)	544
Rdd=vrmpywoh(Rss,Rtt):<<1	Word64 Q6_P_vrmpywoh_PP_s1(Word64 Rss, Word64 Rtt)	544
Rxx+=vrmpywoh(Rss,Rtt)	Word64 Q6_P_vrmpywohacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)	544
Rxx+=vrmpywoh(Rss,Rtt):<<1	Word64 Q6_P_vrmpywohacc_PP_s1(Word64 Rxx, Word64 Rss, Word64 Rtt)	544

vrndwh

Rd=vrndwh(Rss)	Word32 Q6_R_vrndwh_P(Word64 Rss)	581
Rd=vrndwh(Rss):sat	Word32 Q6_R_vrndwh_P_sat(Word64 Rss)	581

vrsadub

Rdd=vrsadub(Rss,Rtt)	Word64 Q6_P_vrsadub_PP(Word64 Rss, Word64 Rtt)	434
Rxx+=vrsadub(Rss,Rtt)	Word64 Q6_P_vrsadubacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)	434

vsathb

Rd=vsathb(Rs)	Word32 Q6_R_vsathb_R(Word32 Rs)	584
Rd=vsathb(Rss)	Word32 Q6_R_vsathb_P(Word64 Rss)	584
Rdd=vsathb(Rss)	Word64 Q6_P_vsathb_P(Word64 Rss)	587

vsathub

Rd=vsathub(Rs)	Word32 Q6_R_vsathub_R(Word32 Rs)	584
Rd=vsathub(Rss)	Word32 Q6_R_vsathub_P(Word64 Rss)	584
Rdd=vsathub(Rss)	Word64 Q6_P_vsathub_P(Word64 Rss)	587

vsatwh

Rd=vsatwh(Rss)	Word32 Q6_R_vsatwh_P(Word64 Rss)	584
Rdd=vsatwh(Rss)	Word64 Q6_P_vsatwh_P(Word64 Rss)	587

vsatwuh

Rd=vsatwuh(Rss)	Word32 Q6_R_vsatwuh_P(Word64 Rss)	584
Rdd=vsatwuh(Rss)	Word64 Q6_P_vsatwuh_P(Word64 Rss)	587

vsplatb

Rd=vsplatb(Rs)	Word32 Q6_R_vsplatb_R(Word32 Rs)	590
Rdd=vsplatb(Rs)	Word64 Q6_P_vsplatb_R(Word32 Rs)	590

vsplath

Rdd=vsplath(Rs)	Word64 Q6_P_vsplath_R(Word32 Rs)	591

vspliceb

Rdd=vspliceb(Rss,Rtt,#u3)	Word64 Q6_P_vspliceb_PPI(Word64 Rss, Word64 Rtt, Word32 Iu3)	592
Rdd=vspliceb(Rss,Rtt,Pu)	Word64 Q6_P_vspliceb_PPp(Word64 Rss, Word64 Rtt, Byte Pu)	592

vsubb

Rdd=vsubb(Rss,Rtt)	Word64 Q6_P_vsubb_PP(Word64 Rss, Word64 Rtt)	437

vsubh

Rd=vsubh(Rt,Rs)	Word32 Q6_R_vsubh_RR(Word32 Rt, Word32 Rs)	197
Rd=vsubh(Rt,Rs):sat	Word32 Q6_R_vsubh_RR_sat(Word32 Rt, Word32 Rs)	197
Rdd=vsubh(Rtt,Rss)	Word64 Q6_P_vsubh_PP(Word64 Rtt, Word64 Rss)	435
Rdd=vsubh(Rtt,Rss):sat	Word64 Q6_P_vsubh_PP_sat(Word64 Rtt, Word64 Rss)	435

vsubub

Rdd=vsubub(Rtt,Rss)	Word64 Q6_P_vsubub_PP(Word64 Rtt, Word64 Rss)	437
Rdd=vsubub(Rtt,Rss):sat	Word64 Q6_P_vsubub_PP_sat(Word64 Rtt, Word64 Rss)	437

vsubuh

Rd=vsubuh(Rt,Rs):sat	Word32 Q6_R_vsubuh_RR_sat(Word32 Rt, Word32 Rs)	197
Rdd=vsubuh(Rtt,Rss):sat	Word64 Q6_P_vsubuh_PP_sat(Word64 Rtt, Word64 Rss)	435

vsubw

Rdd=vsubw(Rtt,Rss)	Word64 Q6_P_vsubw_PP(Word64 Rtt, Word64 Rss)	438
Rdd=vsubw(Rtt,Rss):sat	Word64 Q6_P_vsubw_PP_sat(Word64 Rtt, Word64 Rss)	438

vsxtbh

Rdd=vsxtbh(Rs)	Word64 Q6_P_vsxtbh_R(Word32 Rs)	593

vsxthw

Rdd=vsxthw(Rs)	Word64 Q6_P_vsxthw_R(Word32 Rs)	593

vtrunehb

Rd=vtrunehb(Rss)	Word32 Q6_R_vtrunehb_P(Word64 Rss)	596
Rdd=vtrunehb(Rss,Rtt)	Word64 Q6_P_vtrunehb_PP(Word64 Rss, Word64 Rtt)	596

vtrunewh

Rdd=vtrunewh(Rss,Rtt)	Word64 Q6_P_vtrunewh_PP(Word64 Rss, Word64 Rtt)	596

vtrunohb

Rd=vtrunohb(Rss)	Word32 Q6_R_vtrunohb_P(Word64 Rss)	596
Rdd=vtrunohb(Rss,Rtt)	Word64 Q6_P_vtrunohb_PP(Word64 Rss, Word64 Rtt)	596

vtrunowh

Rdd=vtrunowh(Rss,Rtt)	Word64 Q6_P_vtrunowh_PP(Word64 Rss, Word64 Rtt)	596

vxaddsubh

Rdd=vxaddsubh(Rss,Rtt):rnd:>>1:sat	Word64 Q6_P_vxaddsubh_PP_rnd_rs1_sat(Word64 Rss, Word64 Rtt)	461
Rdd=vxaddsubh(Rss,Rtt):sat	Word64 Q6_P_vxaddsubh_PP_sat(Word64 Rss, Word64 Rtt)	461

vxaddsubw

Rdd=vxaddsubw(Rss,Rtt):sat	Word64 Q6_P_vxaddsubw_PP_sat(Word64 Rss, Word64 Rtt)	463

vxsubaddh

Rdd=vxsubaddh(Rss,Rtt):rnd:>>1:sat	Word64 Q6_P_vxsubaddh_PP_rnd_rs1_sat(Word64 Rss, Word64 Rtt)	461
Rdd=vxsubaddh(Rss,Rtt):sat	Word64 Q6_P_vxsubaddh_PP_sat(Word64 Rss, Word64 Rtt)	461

vxsubaddw

Rdd=vxsubaddw(Rss,Rtt):sat	Word64 Q6_P_vxsubaddw_PP_sat(Word64 Rss, Word64 Rtt)	463

vzxtbh

Rdd=vzxtbh(Rs)	Word64 Q6_P_vzxtbh_R(Word32 Rs)	598

vzxthw

Rdd=vzxthw(Rs)	Word64 Q6_P_vzxthw_R(Word32 Rs)	598

X

xor

Pd=xor(Ps,Pt)	Byte Q6_p_xor_pp(Byte Ps, Byte Pt)	233
Rd=xor(Rs,Rt)	Word32 Q6_R_xor_RR(Word32 Rs, Word32 Rt)	185
Rdd=xor(Rss,Rtt)	Word64 Q6_P_xor_PP(Word64 Rss, Word64 Rtt)	375
Rx^=xor(Rs,Rt)	Word32 Q6_R_xorxacc_RR(Word32 Rx, Word32 Rs, Word32 Rt)	378
Rx&=xor(Rs,Rt)	Word32 Q6_R_xorand_RR(Word32 Rx, Word32 Rs, Word32 Rt)	378
Rx|=xor(Rs,Rt)	Word32 Q6_R_xoror_RR(Word32 Rx, Word32 Rs, Word32 Rt)	379
Rxx^=xor(Rss,Rtt)	Word64 Q6_P_xorxacc_PP(Word64 Rxx, Word64 Rss, Word64 Rtt)	377

Z

zxtb

Rd=zxtb(Rs)	Word32 Q6_R_zxtb_R(Word32 Rs)	199

zxth

Rd=zxth(Rs)	Word32 Q6_R_zxth_R(Word32 Rs)	199