***Advance Information***

AMD

# AMD PROCESSOR RECOGNITION

# *Code Sample*

Publication # **21035** Rev: **A** Amendment/**0**
Issue Date: **July 1996**

# AMD Processor Recognition Code Sample

This document contains a code sample that uses the CPUID instruction to identify the processor and its features. The code was compiled with the Borland C++ compiler v4.5.

## DEFINES Header File (defines.h file)

```
// defines.h : HEADER FILES
#ifndef _H_DEFINES
#define _H_DEFINES
class cpuid {

public:
   int chkcpubit(void);
   int chkcpuid(void);
   void std_vendor_id_str (void);
   void std_cpu_signature(void);
   void ext_vendor_id_str (void);
   void ext_cpu_signature(void);
   void ext_cpu_name_str(void);
   void ext_cpu_cache_info(void);
};
#endif
```

## CHKCPUBIT Module (ckcpubit.cpp file)

```
#include "DEFINES.H"

//chkcpubit checks the processor ID bit (bit 21) in the EFLAGS register.
//The program aborts if the processor does not implement the CPUID instruction.

int cpuid::chkcpubit(void)
{
  asm {
            .486
            pushfd                         //Save eflags
            pop     eax
            test    eax,0x00200000         //Check ID bit (bit 21)
            jz      set_21                 //Bit 21 is not set, so jump to set_21
            and     eax,0xffdfffff         //Clear bit 21
            push    eax                    //Save new value in register
            popfd                          //Store new value in flags
            pushfd
            pop     eax
            test    eax,0x00200000         //Check ID bit
            jz      cpu_id_ok              //If bit 21 is clear, jump to cpu_id_ok
            jmp     err                    //If bit 21 is set, CPUID inst is not
           }                               //supported
           set_21:
           asm {
            or      eax,0x00200000         //Set bit 21
            push    eax                    //Store new value
            popfd                          //Store new value in EFLAGS
            pushfd
            pop     eax
            test    eax,0x00200000         //If bit 21 is on
            jnz     cpu_id_ok              //then jump to cpu_id_ok
           }
           err:
           asm {
            mov     eax,0xffffffff         //CPUID instruction is not supported
            jmp     exit                   //so exit
           }
           cpu_id_ok:                      //Support CPUID instruction
           asm  mov eax,0                  //Return 0
           exit:
           if(_EAX == 0xffffffff){
                 return (-1);
           }
           if (_EAX == 0x0) {
                 return (0);
           }
```

## CHKCPUID Module (chkcpuid.cpp file)

```
#include "DEFINES.H"
#include <iomanip.h>
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>

//chkcpuid identifies the processor name string.

int cpuid::chkcpuid()
{
  char idstr[13];                                //Vendor string variable

  asm {
      mov    eax,0x0                             //EAX = 0
      db     0x0F,0xA2                           //CPUID opcode
  }

//Store the 12 character ASCII string
  idstr[0] = _BL;
  idstr[1] = _BH;
  asm {
            ror ebx,0x10
  }
  idstr[2] = _BL;
  idstr[3] = _BH;
  idstr[4] = _DL;
  idstr[5] = _DH;
  asm {
           ror edx,0x10
  }
  idstr[6] = _DL;
  idstr[7] = _DH;
  idstr[8] = _CL;
  idstr[9] = _CH;
  asm {
            ror  ecx,0x10
  }
  idstr[10] = _CL;
  idstr[11] = _CH;
  idstr[12] = '\0';
  if ( strcmp(idstr, "AuthenticAMD") != 0 )
       return (0);
  else
       return (1);
}
```

## STD_VENDOR_ID_STR Module (cpuidstr.cpp file)

```
#include "DEFINES.H"
#include <iomanip.h>
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>

//std_vendor_id_str finds the largest function value recognized by
//AMD processors. It also identifies AMD as the vendor of the CPU
//by returning "AuthenticAMD" in idstr. If another vendor's identification
//is returned, the program aborts.

void cpuid::std_vendor_id_str()
{
  char idstr[13];                   //Vendor string variable
  int largest_func = 0;             //Largest function variable
  unsigned long  reg_eax,           //Register variable
                 reg_ebx,
                 reg_ecx,
                 reg_edx;
  asm {
      mov    eax,0x0                //EAX = 0
      db     0x0F,0xA2              //CPUID opcode
  }
  reg_eax = _EAX;                   //Store the vendor indentification string
  reg_ebx = _EBX;
  reg_edx = _EDX;
  reg_ecx = _ECX;
  largest_func = _EAX;              //The largest function value
  idstr[0] = _BL;                   //Get the 12-character ASCII string
  idstr[1] = _BH;                   //that identifies AMD as the vendor
  asm {                             //of the CPU.
      ror ebx,0x10
  }
  idstr[2] = _BL;
  idstr[3] = _BH;
  idstr[4] = _DL;
  idstr[5] = _DH;
  asm {
      ror edx,0x10                  //Get the leftmost bit
  }
  idstr[6] = _DL;
  idstr[7] = _DH;
  idstr[8] = _CL;
  idstr[9] = _CH;
  asm {
      ror  ecx,0x10
  }
  idstr[10] = _CL;
```

```
   idstr[11] = _CH;
   idstr[12] = '\0';
   cout.setf(ios::uppercase);
   cout << "\nFuction 0 (EAX = 0)" << endl;
   cout << "==================";
   cout << "\n\n";
   cout << "EAX == " <<setw(8)<< setfill('0') << hex << reg_eax;
   cout << "   EBX == " << setw(8) << hex << reg_ebx;
   cout << "   EDX == " << setw(8) << hex << reg_edx;
   cout << "   ECX == " << setw(8) << hex << reg_ecx;
   cout << "\n\n";
   cout << "     Largest Function Input Value : " << largest_func;
   cout << "\n\n";
   cout << "     Vendor Identification String : " << idstr;
   cout.unsetf(ios::uppercase);
   if ( strcmp(idstr, "AuthenticAMD") != 0 )     //If this not AMD then abort
       {
        cout <<"\n\n\n\n";
        cout <<"     This is not an AMD processor." << "\n\n";
        exit(1);
       }
       cout<< "\n\n\n                Press any key for more." << "\n\n";
       getch();
}
```

## STD_CPU_SIGNATURE Module (cpuname.cpp file)

```
#include "DEFINES.H"
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <conio.h>

//std_cpu_signature identifies the specific CPU by providing information
//regarding the type, instruction family, model, stepping revision, and the
//feature flags. The feature flags indicate the presence of specific features.

void cpuid :: std_cpu_signature (void)
{
  int signature = 0;                     //CPU signature variable
  int stepping_id = 0;                   //CPU stepping id variable
  int model = 0;                         //CPU model variable
  int inst_family = 0;                   //CPU instruction family variable
  unsigned int reg_ax = 0 ;              //AX register
  unsigned long reg_eax,reg_edx, test_reg;//EAX, EDX, and test register variables
  unsigned long print_eax,print_ebx,print_ecx,print_edx;   //Display variables
  int maxbit = 18;                       //Control loop variable
  int bits ;

  asm {
      mov EAX,1                          //EAX = 1 or function 1
      db 0x0F, 0xA2                      //CPUID opcode
  }
  //Display the value of the registers
  print_eax = _EAX;
  print_ebx = _EBX;
  print_ecx = _ECX;
  print_edx = _EDX;

  reg_edx = _EDX;                        //Store the standard feature flags
  reg_ax = _AX;
  asm mov BX, reg_ax
  asm  and BL,0x0F                       //Mask the rightmost 4 bits
  stepping_id = _BL;                     //to get the CPU stepping id

  asm  mov BX, reg_ax
  asm  and BL,0xF0                       //Mask the leftmost 4 bits
  asm  ror BL,4                          //to get the CPU model
  model = _BL;

  asm mov BX, reg_ax                     //Get the CPU instruction family
  asm and BH, 0x0F
  inst_family = _BH;

  asm and EAX,0xFFFFF000                 //Get the bits[31:12]
  asm ror EAX,12
  reg_eax = _EAX;

  asm mov BX, reg_ax                      //Get the CPU signature
  asm and BX,0x0FF0
  signature = _BX;

  clrscr();
  cout.setf(ios::uppercase);
  cout << "Function 1 (EAX = 1)" << endl;
  cout << "====================";
  cout << "\n\n";
  cout << "EAX == "<<setw(8)<<hex<<print_eax;
  cout << "  EBX == "<<setw(8)<<hex<<print_ebx;
  cout << "  ECX == "<<setw(8)<<hex<<print_ecx;
  cout << "  EDX == "<<setw(8)<<hex<<print_edx;
  cout.unsetf(ios::uppercase);
  cout << "\n\n";
  cout << "     EAX[3:0]   == " << setw(1) << hex << stepping_id << endl;
  cout << "     EAX[7:4]   == " << setw(1) << hex << model << endl;
  cout << "     EAX[11:8]  == " << setw(1) << hex << inst_family << endl;
  cout << "     EAX[31:12] == " << setw(5) << hex << reg_eax << endl;
  cout << "\n\n";
  cout << "     Processor Signature : ";
  if (signature == 0x0500)
       cout << "AMD-K5 (Model 0) " << endl;
  else if (signature == 0x0510)
       cout << "AMD-K5 (Model 1) " << endl;
             else if (signature == 0x0560)
                  cout << " AMD-K6 " << endl;
                     else if (signature == 0x0400)
                        cout << "Am486 and Am5x86 " << endl;
  cout << "\n\n             Press any key for more."<<"\n\n";
  getch();

  clrscr();
  cout << "\n";
  cout << "     Feature Flags : " << "\n\n";
  cout << "          EDX == ";
  cout.setf(ios::uppercase);
  cout << setw(8) << hex << reg_edx << "\n\n";
  cout.unsetf(ios::uppercase);

//Get the standard feature flags
  for ( bits = 0; bits < maxbit; bits++){
       switch (bits) {
            case 0 : test_reg = reg_edx;
                   if((test_reg & 0x00000001)== 0x00000001){//Test bit 0
                          cout.width(13);
                          cout.setf(ios::left);
                          cout << "EDX[0]  = 1b ";
                          cout.unsetf(ios::left);
```

```
                    cout << " (bit 0==1 indicates FPU present)" << endl;
               }
               else {
                    cout.width(13);
                    cout.setf(ios::left);
                    cout << "EDX[0]  = 0b ";
                    cout.unsetf(ios::left);
                    cout << " (bit 0==1 indicates FPU present)" << endl;
               }
               test_reg = reg_edx;
               break;
      case 1 : if ((test_reg & 0x00000002 )==0x00000002){  //Test bit 1
                    cout.width(13);
                    cout.setf(ios::left);
                    cout << "EDX[1]  = 1b ";
                    cout.unsetf(ios::left);
                    cout << " (bit 1==1 indicates Virtual Mode Extensions)"
                         << endl;
               }
               else {
                    cout.width(13);
                    cout.setf(ios::left);
                    cout << "EDX[1]  = 0b ";
                    cout.unsetf(ios::left);
                    cout << " (bit 1==1 indicates Virtual Mode Extensions)"
                         << endl;
               }
               test_reg = reg_edx;
               break;
      case 2 : if ((test_reg & 0x00000004 )==0x00000004){   //Test bit 2
                    cout.width(13);
                    cout.setf(ios::left);
                    cout << "EDX[2]  = 1b ";
                    cout.unsetf(ios::left);
                    cout << " (bit 2==1 indicates Debugging Extensions)"
                     << endl;
               }
               else {
                    cout.width(13);
                    cout.setf(ios::left);
                    cout << "EDX[2]  = 0b ";
                    cout.unsetf(ios::left);
                    cout << " (bit 2==1 indicates Debugging Extensions )"
                     << endl;
               }
               test_reg = reg_edx;
               break;
      case 3 : if ((test_reg & 0x00000008 )==0x00000008){   //Test bit 3
                    cout.width(13);
                    cout.setf(ios::left);
                    cout << "EDX[3]  = 1b ";
```

```
                        cout.unsetf(ios::left);
                        cout << " (bit 3==1 indicates Page Size Extensions)"
                         << endl;
                   }
                   else {
                        cout.width(13);
                        cout.setf(ios::left);
                        cout << "EDX[3]  = 0b ";
                        cout.unsetf(ios::left);
                        cout << " (bit 3==1 indicates Page Size Extensions)"
                         << endl;
                   }
                   test_reg = reg_edx;
                   break;
          case 4 : if ((test_reg & 0x00000010 )==0x00000010){    //Test bit 4
                        cout.width(13);
                        cout.setf(ios::left);
                        cout << "EDX[4]  = 1b ";
                        cout.unsetf(ios::left);
                        cout << " (bit 4==1 indicates Time Stamp Counter)"
                         << endl;
                   }
                   else {
                        cout.width(13);
                        cout.setf(ios::left);
                        cout << "EDX[4]  = 0b ";
                        cout.unsetf(ios::left);
                        cout << " (bit 4==1 indicates Time Stamp Counter )"
                         << endl;
                      }
                      test_reg = reg_edx;
                      break;
          case 5 : if ((test_reg & 0x00000020 )==0x00000020){   //Test bit 5
                        cout.width(13);
                        cout.setf(ios::left);
                        cout << "EDX[5]  = 1b ";
                        cout.unsetf(ios::left);
                        cout << " (bit 5==1 indicates K86 Model-Specific “;
                        cout <<“Registers)" << endl;
                   }
                   else {
                        cout.width(13);
                        cout.setf(ios::left);
                        cout << "EDX[5]  = 0b ";
                        cout.unsetf(ios::left);
                        cout << " (bit 5==1 indicates K86 Model-Specific “;
                        cout <<“Registers)" << endl;
                   }
                   test_reg = reg_edx;
                   break;
          case 6 : if ((test_reg & 0x00000040 )==0x00000000){   //Test bit 6
```

```
                    cout.width(12);
                    cout.setf(ios::left);
                    cout << "EDX[6]  = 0b";
                    cout.unsetf(ios::left);
                    cout << "  Reserved" << endl;
               }
               test_reg = reg_edx;
               break;
      case 7 : if ((test_reg & 0x00000080 )==0x00000080){  //Test bit 7
                    cout.width(13);
                    cout.setf(ios::left);
                    cout << "EDX[7]  = 1b ";
                    cout.unsetf(ios::left);
                    cout << " (bit 7==1 indicates Support of Machine";
                    cout << " Check Exception)" << endl;
               }
               else {
                    cout.width(13);
                    cout.setf(ios::left);
                    cout << "EDX[7]  = 0b ";
                    cout.unsetf(ios::left);
                    cout << " (bit 7==1 indicates Support of Machine";
                    cout << " Check Exception)" << endl;
               }
               test_reg = reg_edx;
               break;
      case 8 : if ((test_reg & 0x00000100 )==0x00000100){    //Test bit 8
                    cout.width(13);
                    cout.setf(ios::left);
                    cout << "EDX[8]  = 1b ";
                    cout.unsetf(ios::left);
                    cout << " (bit 8==1 indicates Support of CMPXCHG8B";
                    cout << " Extensions)" << endl;
               }
               else {
                    cout.width(13);
                    cout.setf(ios::left);
                    cout << "EDX[8]  = 0b ";
                    cout.unsetf(ios::left);
                    cout << " (bit 8==1 indicates Support of CMPXCHG8B";
                    cout << " Extensions)" << endl;
               }
               test_reg = reg_edx;
               break;
```

```
case 9 : if ((test_reg & 0x00000200 )==0x00000200){ //Test bit 9
               if (signature == 0x0500){
                 cout.width(13);
                 cout.setf(ios::left);
                 cout << "EDX[9]  = 1b ";
                 cout.unsetf(ios::left);
                 cout << " (bit 9==1 indicates Support of Global";
                 cout << " Paging Extension)"<< endl;
               }
               else {
                 cout.width(13);
                 cout.setf(ios::left);
                 cout << "EDX[9]  = 1b ";
                 cout.unsetf(ios::left);
                 cout << " (bit 9==1 indicates Support of APIC "
                  << endl;
               }
         }
         else {
              if (signature == 0x0500){
                cout.width(13);
                cout.setf(ios::left);
                cout << "EDX[9]  = 0b ";
                cout.unsetf(ios::left);
                cout << " (bit 9==1 indicates Support of Global";
                cout << " Paging Extensions)"<< endl;
               }
               else {
                 cout.width(13);
                 cout.setf(ios::left);
                 cout << "EDX[9]  = 0b ";
                 cout.unsetf(ios::left);
                 cout << " (bit 9==1 indicates Support of APIC "
                  << endl;
               }
         }
         test_reg = reg_edx;
         break;
case 10 : if ((test_reg & 0x00000C00 )==0x00000000){ //Test bits 10:11
              cout.width(12);
              cout.setf(ios::left);
              cout << "EDX[10:11] = ";
              cout.unsetf(ios::left);
              cout << " Reserved"<< endl;
         }
         test_reg = reg_edx;
         break;
case 11 : if ((test_reg & 0x00001000 )==0x00001000){ //Test bit 12
              cout.width(12);
              cout.setf(ios::left);
              cout << "EDX[12] = 1b";
```

```
                   cout.unsetf(ios::left);
                   cout << " (bit 12==1 indicates Memory Type Range “;
                   cout <<Registers)" << endl;
              }
              else {
                   cout.width(12);
                   cout.setf(ios::left);
                   cout << "EDX[12] = 0b ";
                   cout.unsetf(ios::left);
                   cout << " (bit 12==1 indicates Memory Type Range “;
                   cout << “Registers)" << endl;
              }
              test_reg = reg_edx;
              break;
     case 12 : if ((test_reg & 0x00002000 )==0x00002000){ //Test bit 13
                   cout.width(12);
                   cout.setf(ios::left);
                   cout << "EDX[13] = 1b ";
                   cout.unsetf(ios::left);
                   cout << " (bit 13==1 indicates Global Paging Extension)"
                           << endl;
              }
              else {
                     if (signature == 0x0500){
                       cout.width(12);
                       cout.setf(ios::left);
                       cout << "EDX[13]  = 0b";
                       cout.unsetf(ios::left);
                       cout << "  Reserved" << endl;
                     }
                     else {
                       cout.width(12);
                       cout.setf(ios::left);
                       cout << "EDX[13] = 0b ";
                       cout.unsetf(ios::left);
                       cout << " (bit 13==1 indicates Global Paging ”;
                       cout << “Extension)"  << endl;
                     }
              }
              test_reg = reg_edx;
              break;
     case 13 : if ((test_reg & 0x00004000 )==0x00000000){ //Test bit 14
                     cout.width(12);
                     cout.setf(ios::left);
                     cout << "EDX[14] = 0b ";
                     cout.unsetf(ios::left);
                     cout << " Reserved" << endl;
              }
              test_reg = reg_edx;
              break;
```

```
case 14 : if ((test_reg & 0x00008000 )==0x00008000){ //Test bit 15
               cout.width(12);
               cout.setf(ios::left);
               cout << "EDX[15] = 1b ";
               cout.unsetf(ios::left);
               cout << " (bit 15==1 indicates Conditional Move “;
               cout <<“Instruction)" << endl;
          }
          else {
               cout.width(12);
               cout.setf(ios::left);
               cout << "EDX[15] = 0b ";
               cout.unsetf(ios::left);
               cout << " (bit 15==1 indicates Conditional Move “;
               cout <<“Instruction)" << endl;
          }
          test_reg = reg_edx;
          break;
case 15 : if ((test_reg & 0x007F0000 )==0x00000000){ //Test bits 16:22
               cout.width(12);
               cout.setf(ios::left);
               cout << "EDX[16:22] = ";
               cout.unsetf(ios::left);
               cout << " Reserved" << endl;
          }
          test_reg = reg_edx;
          break;
case 16 : if ((test_reg & 0x00800000 )==0x00800000){ //Test bit 23
               cout.width(12);
               cout.setf(ios::left);
               cout << "EDX[23] = 1b ";
               cout.unsetf(ios::left);
               cout << " (bit 23==1 indicates Support of Multimedia"
                    << " Extensions)" << endl;
          }
          else {
               cout.width(12);
               cout.setf(ios::left);
               cout << "EDX[23] = 0b ";
               cout.unsetf(ios::left);
               cout << " (bit 23==1 indicates Support of Multimedia"
                    << " Extensions)" << endl;
          }
          test_reg = reg_edx;
          break;
```

```
            case 17: if ((test_reg & 0xFF000000) == 0x00000000){ //Test bits 24:31
                         cout.width(12);
                         cout.setf(ios::left);
                         cout << "EDX[24:31] = ";
                         cout.unsetf(ios::left);
                         cout << " Reserved" << endl;
                      }
        }
  }
  cout << "\n                   Press any key for more. " << endl;
  getch();
}
```

## EXT_VENDOR_ID_STR Module (extidstr.cpp file)

```
#include "DEFINES.H"
#include <iomanip.h>
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>

//ext_vendor_id_str finds the largest extended function value
//recognized by AMD processors.

void cpuid::ext_vendor_id_str()
{
  unsigned long reg_eax,                    //Register variables
                reg_ebx,
                reg_ecx,
                reg_edx,
                largest_func;               //Largest function variable
  asm {
      mov    eax,0x80000000                 //EAX = 8000_0000h
      db     0x0F,0xA2                      //CPUID opcode
  }
  largest_func = _EAX;                      //The largest function value
  reg_ebx = _EBX;
  reg_edx = _EDX;
  reg_ecx = _ECX;
  clrscr();
  cout.setf(ios::uppercase);
  cout << "\nFuction 8000_0000h (EAX = 80000000)" << endl;
  cout << "===================================";
  cout << "\n\n";
  cout << "      EAX == " <<setw(8)<< hex << largest_func<< endl;;
  cout << "      EBX == " << setw(8) << hex << reg_ebx << endl;
  cout << "      EDX == " << setw(8) << hex << reg_edx << endl;
  cout << "      ECX == " << setw(8) << hex << reg_ecx << endl;
  cout << "\n\n";
  cout.unsetf(ios::uppercase);
  cout << "     Largest Extended Function Input Value : " << largest_func;
  cout << "\n\n";
  cout << "     EBX, EDX, ECX : Reserved " << "\n\n";
  cout << "                    Press any key for more." << endl;
  cout <<"\n\n\n\n";
  getch();
}
```

## EXT_CPU_SIGNATURE Module (extname.cpp file)

```
#include "DEFINES.H"
#include <iostream.h>
#include <iomanip.h>
#include <stdlib.h>
#include <conio.h>

//ext_cpu_signature identifies the specific CPU by providing information
//regarding the type, instruction family, model, stepping revision, and
//feature flags. The feature flags indicate the presence of specific
//features.

void cpuid :: ext_cpu_signature (void)
{
  int signature = 0;                            //Signature variable
  int stepping_id = 0;                          //Steeping id variable
  int model = 0;                                //Model variable
  int inst_family = 0;                          //Instruction family variable
  unsigned int reg_ax = 0 ;                     //AX register
  unsigned long reg_eax,reg_edx,test_reg,       //Register variables
            print_eax,print_ebx,print_ecx,print_edx;//Display variables
  int maxbit = 18;                              //Control loop
  int bits ;                                    //Case statement
  asm {
      mov EAX,0x80000001                        //EAX = 8000_0001h
      db 0x0F, 0xA2                             //CPUID opcode
  }
  print_eax = _EAX;
  print_ebx = _EBX;
  print_ecx = _ECX;
  print_edx = _EDX;
  reg_edx = _EDX;                       //Store the exteneded feature flags
  reg_ax = _AX;
  asm mov BX, reg_ax
  asm  and BL,0x0F                      //Mask the rightmost 4 bits
  stepping_id = _BL;                    //to get the CPU stepping id
  asm  mov BX, reg_ax
  asm  and BL,0xF0                      //Mask the leftmost 4 bits
  model = _BL;                          //to get the CPU model id
  asm mov BX, reg_ax
  asm and BH, 0x0F
  inst_family = _BH;
  asm and EAX,0xFFFFF000                //Get bits 31-12
  asm ror EAX,12
  reg_eax = _EAX;
  asm mov BX, reg_ax                    //Get the CPU signature
  asm and BX,0x0FF0
  signature = _BX;
  clrscr();
  cout.setf(ios::uppercase);
```

```
cout << "Function 8000_0001h (EAX = 80000001)" << endl;
cout << "=====================================";
cout << "\n";
cout << "EAX == "<< setw(8) << hex << print_eax<<"   EBX == " << setw(8)
     << hex << print_ebx << "   ECX == "<< setw (8) << hex << print_ecx
     << "   EDX == " << setw(8) << hex << print_edx << "\n\n";
cout << "     EAX[3:0]   == " << setw(1) << hex << stepping_id << endl;
cout << "     EAX[7:4]   == " << setw(1) << hex << model << endl;
cout << "     EAX[11:8]  == " << setw(1) << hex << inst_family << endl;
cout << "     EAX[31:12] == " << setw(5) << hex << reg_eax << endl;
cout.unsetf(ios::uppercase);

cout << "\n";
cout << "     AMD Processor Signature : ";

if (signature == 0x0660)
     cout << " AMD-K6 " << endl;
else {
  if(signature == 0x0510)
     cout << " AMD-K5 (Model 1) " << endl;
  else cout << " Undefined " << endl;
}

cout << "\n";
cout << "     Feature Flags : " << "\n\n";
cout << "          EDX == ";
cout.setf(ios::uppercase);
cout << setw(8) << hex << reg_edx << "\n\n";
cout.unsetf(ios::uppercase);
cout << "                    Press any key for more. " << endl;
cout << "\n\n";
getch();
clrscr();
//Get the feature flags
for ( bits = 0; bits <= maxbit; bits++){
     switch (bits) {
          case 0 : test_reg = reg_edx;
                   if((test_reg & 0x00000001)== 0x00000001){//Test bit 0
                      cout.width(13);
                      cout.setf(ios::left);
                      cout << "EDX[0]  = 1b ";
                      cout.unsetf(ios::left);
                      cout << " (bit 0==1 indicates FPU present)" << endl;
                   }
                   else {
                      cout.width(13);
                      cout.setf(ios::left);
                      cout << "EDX[0]  = 0b ";
                      cout.unsetf(ios::left);
                      cout << " (bit 0==1 indicates FPU present)" << endl;
                   }
```

```
            test_reg = reg_edx;
            break;
case 1 : if ((test_reg & 0x00000002 )==0x00000002){ //Test bit 1
            cout.width(13);
            cout.setf(ios::left);
            cout << "EDX[1]  = 1b ";
            cout.unsetf(ios::left);
            cout << " (bit 1==1 indicates Virtual Mode Extensions)"
                   << endl;
         }
         else {
            cout.width(13);
            cout.setf(ios::left);
            cout << "EDX[1]  = 0b ";
            cout.unsetf(ios::left);
            cout << " (bit 1==1 indicates Virtual Mode Extensions)"
                     << endl;
         }
         test_reg = reg_edx;
         break;
case 2 : if ((test_reg & 0x00000004 )==0x00000004){   //Test bit 2
            cout.width(13);
            cout.setf(ios::left);
            cout << "EDX[2]  = 1b ";
            cout.unsetf(ios::left);
            cout << " (bit 2==1 indicates Debugging Extensions)"
                     << endl;
         }
         else {
            cout.width(13);
            cout.setf(ios::left);
            cout << "EDX[2]  = 0b ";
            cout.unsetf(ios::left);
            cout << " (bit 2==1 indicates Debugging Extensions )"
                     << endl;
         }
         test_reg = reg_edx;
         break;
case 3 : if ((test_reg & 0x00000008 )==0x00000008){   //Test bit 3
            cout.width(13);
            cout.setf(ios::left);
            cout << "EDX[3]  = 1b ";
            cout.unsetf(ios::left);
            cout << " (bit 3==1 indicates Page Size Extensions)"
                     << endl;
         }
         else {
            cout.width(13);
            cout.setf(ios::left);
            cout << "EDX[3]  = 0b ";
            cout.unsetf(ios::left);
```

```
                                cout << " (bit 3==1 indicates Page Size Extensions)"
                                         << endl;
                             }
                             test_reg = reg_edx;
                             break;
                    case 4 : if ((test_reg & 0x00000010 )==0x00000010){    //Test bit 4
                                cout.width(13);
                                cout.setf(ios::left);
                                cout << "EDX[4]  = 1b ";
                                cout.unsetf(ios::left);
                                cout << " (bit 4==1 indicates Time Stamp Counter)"
                                            << endl;
                             }
                             else {
                                cout.width(13);
                                cout.setf(ios::left);
                                cout << "EDX[4]  = 0b ";
                                cout.unsetf(ios::left);
                                cout << " (bit 4==1 indicates Time Stamp Counter )"
                                            << endl;
                             }
                             test_reg = reg_edx;
                             break;
                    case 5 : if ((test_reg & 0x00000020 )==0x00000020){   //Test bit 5
                                cout.width(13);
                                cout.setf(ios::left);
                                cout << "EDX[5]  = 1b ";
                                cout.unsetf(ios::left);
                                cout << " (bit 5==1 indicates K86 Model-Specific"
                                   << “ Registers)" << endl;
                             }
                             else {
                                cout.width(13);
                                cout.setf(ios::left);
                                cout << "EDX[5]  = 0b ";
                                cout.unsetf(ios::left);
                                cout << " (bit 5==1 indicates K86 Model-Specific"
                                   << " Registers)" << endl;
                             }
                             test_reg = reg_edx;
                             break;
                    case 6 : if((test_reg & 0x00000040) == 0x00000000){   //Test bit 6
                                cout.width(13);
                                cout.setf(ios::left);
                                cout << "EDX[6]  = 0b ";
                                cout.unsetf(ios::left);
                                cout << " Reserved" << endl;
                             }
                             test_reg = reg_edx;
                             break;
                    case 7 : if ((test_reg & 0x00000080 )==0x00000080){  //Test bit 7
```

```
                  cout.width(13);
                  cout.setf(ios::left);
                  cout << "EDX[7]  = 1b ";
                  cout.unsetf(ios::left);
                  cout << " (bit 7==1 indicates Support of Machine";
                  cout << " Check Exception)" << endl;
               }
               else {
                  cout.width(13);
                  cout.setf(ios::left);
                  cout << "EDX[7]  = 0b ";
                  cout.unsetf(ios::left);
                  cout << " (bit 7==1 indicates Support of Machine";
                  cout << " Check Exception)" << endl;
               }
               test_reg = reg_edx;
               break;
      case 8 : if ((test_reg & 0x00000100 )==0x00000100){//Test bit 8
                  cout.width(13);
                  cout.setf(ios::left);
                  cout << "EDX[8]  = 1b ";
                  cout.unsetf(ios::left);
                  cout << " (bit 8==1 indicates Support of CMPXCHG8B "
                        << "instruction)" << endl;
               }
               else {
                  cout.width(13);
                  cout.setf(ios::left);
                  cout << "EDX[8]  = 0b ";
                  cout.unsetf(ios::left);
                  cout << " (bit 8==1 indicates Support of CMPXCHG8B";
                  cout << " instruction)" << endl;
               }
               test_reg = reg_edx;
               break;
      case 9 : if ((test_reg & 0x00000200 )==0x00000000){//Test bit 9
                  cout.width(13);
                  cout.setf(ios::left);
                  cout << "EDX[9]  = 0b ";
                  cout.unsetf(ios::left);
                  cout << " Reserved" << endl;
               }
               test_reg = reg_edx;
               break;
      case 10 :if ((test_reg & 0x00000400 )==0x00000400){//Test bit 10
                  cout.width(12);
                  cout.setf(ios::left);
                  cout << "EDX[10] = 1b ";
                  cout.unsetf(ios::left);
                  cout << " (bit 10==1 indicates Support of SYSCALL";
                  cout << " and SYSRET Extension)"<< endl;
```

```
            }
            else {
               cout.width(12);
               cout.setf(ios::left);
               cout << "EDX[10] = 0b ";
               cout.unsetf(ios::left);
               cout << " (bit 10==1 indicates Support of SYSCALL";
               cout << " and SYSRET Extensions)" << endl;
            }
            test_reg = reg_edx;
            break;
   case 11 :if ((test_reg & 0x00001800 )==0x00000000){  //Test bit 11
               cout.width(12);
               cout.setf(ios::left);
               cout << "EDX[11:12] = ";
               cout.unsetf(ios::left);
               cout << " Reserved" << endl;
            }
            test_reg = reg_edx;
            break;
   case 12 :if ((test_reg & 0x00002000 )==0x00002000){  //Test bit 12
               cout.width(12);
               cout.setf(ios::left);
               cout << "EDX[13] = 1b ";
               cout.unsetf(ios::left);
               cout << " (bit 13==1 indicates Support of Global Paging "
                      << "Extensions)" << endl;
            }
            else {
               cout.width(12);
               cout.setf(ios::left);
               cout << "EDX[13] = 0b ";
               cout.unsetf(ios::left);
               cout << " (bit 13==1 indicates Support of Global Paging "
                      << "Extensions) "<< endl;
            }
            test_reg = reg_edx;
            break;
   case 13 :if ((test_reg & 0x00004000 )==0x00000000){//Test bit 13
               cout.width(12);
               cout.setf(ios::left);
               cout << "EDX[14] = ";
               cout.unsetf(ios::left);
               cout << "  Reserved" << endl;
            }
            test_reg = reg_edx;
            break;
   case 14 :if ((test_reg & 0x00008000 )==0x00008000){//Test bit 14
               cout.width(12);
               cout.setf(ios::left);
               cout << "EDX[15] = 1b ";
```

```
                  cout.unsetf(ios::left);
                  cout << " (bit 15==1 indicates Support of Integer "
                         << "Conditional Move" << endl;
                  cout << "              Instructions)" << endl;
               }
               else {
                  cout.width(12);
                  cout.setf(ios::left);
                  cout << "EDX[15] = 0b ";
                  cout.unsetf(ios::left);
                  cout << " (bit 15==1 indicates Support of Integer"
                         << " Conditional Move" << endl;
                  cout << "              Instructions)" << endl;
               }
               test_reg = reg_edx;
               break;
      case 15 :if ((test_reg & 0x00010000) == 0x00010000){//Test bit 15
                  cout.width(12);
                  cout.setf(ios::left);
                  cout << "EDX[16] = 1b ";
                  cout.unsetf(ios::left);
                  cout << " (bit 16==1 indicates Support of Floating-Point"
                         << " Conditional Move" << endl;
                  cout << "              Instructions) " << endl;
               }
               else {
                  cout.width(12);
                  cout.setf(ios::left);
                  cout << "EDX[16] = 0b ";
                  cout.unsetf(ios::left);
                  cout << " (bit 16==1 indicates Support of Floating-Point"
                         << " Conditional Move" << endl;
                  cout << "              Instructions) " << endl;
               }
               test_reg = reg_edx;
               break;
      case 16 :if ((test_reg & 0x007E0000) == 0x00000000) {//Test bit 16
                  cout.width(12);
                  cout.setf(ios::left);
                  cout << "EDX[17:22] = ";
                  cout.unsetf(ios::left);
                  cout << " Reserved" << endl;
               }
               test_reg = reg_edx;
               break;
      case 17 :if ((test_reg & 0x00800000) == 0x00800000){//Test bit 17
                  cout.width(12);
                  cout.setf(ios::left);
                  cout << "EDX[23] = 1b ";
                  cout.unsetf(ios::left);
                  cout << " (bit 23==1 indicates Support of Multimedia"
                                << " Extensions) " << endl;
                    }
                    else {
                       cout.width(12);
                       cout.setf(ios::left);
                       cout << "EDX[23] = 0b ";
                       cout.unsetf(ios::left);
                       cout << " (bit 16==1 indicates Support of Multimedia"
                              << " Extensions) " << endl;
                    }
                    test_reg = reg_edx;
                    break;
           case 18 :if ((test_reg & 0xFF000000) == 0x00000000) {//Test bit 18
                       cout.width(12);
                       cout.setf(ios::left);
                       cout << "EDX[24:31] = ";
                       cout.unsetf(ios::left);
                       cout << " Reserved" << endl;
                    }
       }
  }
  cout << "                    Press any key for more. " << endl;
  getch();
}
```

## EXT_CPU_NAME_STR Module (extstr.cpp file)

```
#include "defines.h"
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>

//ext_cpu_name_str displays the processor name string (up to 48 characters).
//The processor name string is the name of the AMD processor.

void cpuid :: ext_cpu_name_str(void)
{
 unsigned long reg_eax, reg_ebx, reg_ecx, reg_edx;    //Register variables
 char idstr[48];                                     //Processor name string variable
 int func;                                           //Case statement variable
 int maxfunc = 2;                                    //Control loop variable
 clrscr();
 cout << "Function 8000_0002, 8000_0003, 8000_0004 (EAX = 80000002/3/4)" << endl;
 cout << "==============================================================";
 cout << "\n\n";
 for (func = 0; func <= maxfunc; func++){
      switch (func){
            case 0 :  cout << "          Input: EAX = 80000002 " << endl;
                        cout << "              EAX = ";
                        cout.setf(ios::uppercase);
                        asm mov eax, 0x80000002 //EAX=80000002
                        asm db 0x0F, 0xA2       //CPUID opcode
                        reg_eax = _EAX;         //Store the processor name string
                        reg_ebx = _EBX;
                        reg_edx = _EDX;
                        reg_ecx = _ECX;
                        cout << setw(8) << hex << reg_eax << endl;
                        cout << "              EBX = " << setw(8) << hex
                             << reg_ebx << endl;
                        cout << "              ECX = "
                              << setw(8) << hex << reg_ecx << endl;
                        cout << "              EDX = " << setw(8) << hex
                             << reg_edx << endl;
                         _EAX = reg_eax;
                         _EBX = reg_ebx;
                         _ECX = reg_ecx;
                         _EDX = reg_edx;
                     //Get the first 12 characters of the processor name string
                        idstr[0] = _AL;
                        idstr[1] = _AH;
                        asm  ror eax,0x10
                        idstr[2] = _AL;
                        idstr[3] = _AH;
                        idstr[4] = _BL;
                        idstr[5] = _BH;
                        asm ror ebx,0x10
```

```
                    idstr[6] = _BL;
                    idstr[7] = _BH;
                    idstr[8] = _CL;
                    idstr[9] = _CH;
                    asm ror  ecx,0x10
                    idstr[10] = _CL;
                    idstr[11] = _CH;
                    idstr[12] = _DL;
                    idstr[13] = _DH;
                    asm ror edx, 0x10;
                    idstr[14] = _DL;
                    idstr[15] = _DH;
                    //idstr[16] = ’\0’;
                    break;
        case 1 :  cout << "        Input: EAX = 80000003 " << endl;
                    cout << "              EAX = ";
                    asm mov eax, 0x80000003    //EAX = 8000_0003
                    asm db 0x0F, 0xA2          //CPUID opcode
                    reg_eax = _EAX;
                    reg_ebx = _EBX;
                    reg_edx = _EDX;
                    reg_ecx = _ECX;
                    cout << setw(8) << hex << reg_eax << endl
                         << "              EBX = " << setw(8) << hex
                         << reg_ebx << endl << "              ECX = "
                         << setw(8) << hex << reg_ecx << endl
                         << "              EDX = " << setw(8) << hex
                         << reg_edx << endl;
                     _EAX = reg_eax;
                     _EBX = reg_ebx;
                     _ECX = reg_ecx;
                     _EDX = reg_edx;
                 //Get the second 12 characters of the processor name string
                    idstr[16] = _AL;
                    idstr[17] = _AH;
                    asm  ror eax,0x10
                    idstr[18] = _AL;
                    idstr[19] = _AH;
                    idstr[20] = _BL;
                    idstr[21] = _BH;
                    asm ror ebx,0x10
                    idstr[22] = _BL;
                    idstr[23] = _BH;
                    idstr[24] = _CL;
                    idstr[25] = _CH;
                    asm ror  ecx,0x10
                    idstr[26] = _CL;
                    idstr[27] = _CH;
                    idstr[28] = _DL;
                    idstr[29] = _DH;
                    asm ror edx, 0x10;
```

```
                        idstr[30] = _DL;
                        idstr[31] = _DH;
                        break;
            case 2 :  cout << "          Input: EAX = 80000004 " << endl;
                        cout << "              EAX = ";
                        asm mov eax, 0x80000004      //EAX = 8000_00004
                        asm db 0x0F, 0xA2            //CPUID opcode
                        reg_eax = _EAX;
                        reg_ebx = _EBX;
                        reg_edx = _EDX;
                        reg_ecx = _ECX;
                        cout << setw(8) << hex << reg_eax << endl
                             << "              EBX = " << setw(8) << hex
                             << reg_ebx << endl << "              ECX = "
                             << setw(8) << hex << reg_ecx << endl
                             << "              EDX = " << setw(8) << hex
                             << reg_edx << endl;
                         cout.unsetf(ios::uppercase);
                         _EAX = reg_eax;
                         _EBX = reg_ebx;
                         _ECX = reg_ecx;
                         _EDX = reg_edx;
                     //Get the rest of the processor name string
                        idstr[32] = _AL;
                        idstr[33] = _AH;
                        asm  ror eax,0x10
                        idstr[34] = _AL;
                        idstr[35] = _AH;
                        idstr[36] = _BL;
                        idstr[37] = _BH;
                        asm ror ebx,0x10
                        idstr[38] = _BL;
                        idstr[39] = _BH;
                        idstr[40] = _CL;
                        idstr[41] = _CH;
                        asm ror  ecx,0x10
                        idstr[42] = _CL;
                        idstr[43] = _CH;
                        idstr[44] = _DL;
                        idstr[45] = _DH;
                        asm ror edx, 0x10;
                        idstr[46] = _DL;
                        idstr[47] = _DH;
                        idstr[48] = '\0';
                        break;
      }
 }
 cout << "\n   Processor Name String : " << idstr;
 cout << "\n\n          Press any key for more."<<"\n\n";
 getch();
}
```

## EXT_CPU_CACHE_INFO Module (extcache.cpp file)

```
#include "defines.h"
#include <iostream.h>
#include <iomanip.h>
#include <conio.h>

//ext_cpu_cache_info provides information about the instruction TLB,
//data TLB, L1 instruction cache, and L1 data cache.

void cpuid::ext_cpu_cache_info(void)
{
  unsigned long reg_eax, reg_ebx, reg_ecx, reg_edx, test_reg; //Register variables
  unsigned long  bits7_0, bits15_8, bits23_16, bits31_24;     //TLB and data cache
                                                       //variables and L1
                                                       //instruction cache info
  clrscr();
  cout << "Function 8000_0005 (EAX = 80000005)" << endl;
  cout << "==================================== " << "\n\n";
  cout << " Processor Cache Information : " << "\n\n";
  asm mov eax,0x80000005                        //EAX = 8000_0005h
  asm db 0x0F, 0xA2                             //CPUID opcode
  reg_eax = _EAX;                               //Store the EAX register
  reg_ebx = _EBX;                               //Store data and instruction TLB
  reg_edx = _EDX;                               //Store the L1 data cache
  reg_ecx = _ECX;                               //Store the L1 instruction cache
  cout.setf(ios::uppercase);
  cout << " EAX == " << setw(8) << hex << reg_eax << "  EBX == "
           << setw(8) << hex << reg_ebx << "  ECX == " << setw(8)
           << hex << reg_ecx << "  EDX == " << setw(8) << hex
           << reg_edx << "\n\n";

  test_reg = reg_ebx;                           //Data and instruction TLB
  bits7_0 = (test_reg & 0x000000ff);            //Instruction TLB entries
  test_reg = reg_ebx;
  bits15_8 = (test_reg & 0x0000ff00);           //Associativity of inst TLB
  bits15_8 >>= 8;
  test_reg = reg_ebx;
  bits23_16 = (test_reg & 0x00ff0000);          //Data TLB entries
  bits23_16 >>= 16;
  test_reg = reg_ebx;
  bits31_24 = (test_reg & 0xff000000);          //Associativity of data TLB
  bits31_24 >>= 24;
  cout<<"\n\n";
  cout<<"      ------------------------------------------------------------- "
          <<endl;
  cout<<"       |        |         Data TLB        |  Instruction TLB       |"
          << endl;
  cout<<"      ------------------------------------------------------------- "
          << endl;
  cout<<"       |        |Associativity| #Entries  |Associativity| #Entries |"
```

```
          << endl;
cout<<"       ------------------------------------------------------------ "
          << endl;
cout<<"       |        |Bits 31-24   |Bits 23-16 | Bits 15-8   | Bits 7-0 |"
          << endl;
cout<<"       ------------------------------------------------------------"
          << endl;
cout<<"       |  EBX   |     " << setw(2) << hex << bits31_24
          <<"      |    " << setw(2)<< hex << bits23_16 <<"     |      "
          << setw(2) << dec
          << bits15_8 << "     |     " << setw(2) << hex << bits7_0 << "   |"
                << endl;
cout<<"       ------------------------------------------------------------"
          << endl;
cout<<"       Note: " << endl;
cout<<"            Full associativity is indicated by a value of 0FFh."
          <<"\n\n";
cout<< "                        Press any key for more." <<"\n\n\n";
getch();
test_reg = reg_ecx;
bits7_0 = (test_reg & 0x000000ff);            //Line size of L1 data cache
test_reg = reg_ecx;
bits15_8 = (test_reg & 0x0000ff00);           //Lines per tag of L1 data cache
bits15_8 >>= 8;
test_reg = reg_ecx;
bits23_16 = (test_reg & 0x00ff0000);          //Associativity
bits23_16 >>= 16;
test_reg = reg_ecx;
bits31_24 = (test_reg & 0xff000000);//Size
bits31_24 >>= 24;
clrscr();
cout<< "\n\n\n";
cout<<"       ------------------------------------------------------------ "
          <<endl;
cout<<"       |        |                L1    Data   Cache                |"
          << endl;
cout<<"       ------------------------------------------------------------ "
          << endl;
cout<<"       |        |    Size     |Associa -  | Lines  per  |Line Size |"
          << endl;
cout<<"       |        |  (Kbytes)   |tivity     |      Tag    |  (bytes) |"
          << endl;
cout<<"       ------------------------------------------------------------ "
          << endl;
cout<<"       |        |Bits 31-24   |Bits 23-16 | Bits 15-8   | Bits 7-0 |"
          << endl;
cout<<"       ------------------------------------------------------------"
          << endl;
cout<<"       |  ECX   |     " << setw(2) << hex << bits31_24
          <<"      |    " << setw(2)<< hex << bits23_16 <<"     |      "
          << setw(2) << dec
          << bits15_8 << "     |     " << setw(2) << hex << bits7_0 << "   |"
                << endl;
  cout<<"       ------------------------------------------------------------"
          << endl;
  cout<<"       Note: " << endl;
  cout<<"            Full associativity is indicated by a value of OFFh."
          <<"\n\n";
  cout<< "                        Press any key for more." << endl;
  getch();
  test_reg = reg_edx;
  bits7_0 = (test_reg & 0x000000ff);      //Line size of L1 instruction cache
  test_reg = reg_edx;
  bits15_8 = (test_reg & 0x0000ff00);     //Lines per tag of L1 instruction cache
  bits15_8 >>= 8;
  test_reg = reg_edx;
  bits23_16 = (test_reg & 0x00ff0000);    //Associativity
  bits23_16 >>= 16;
  test_reg = reg_edx;
  bits31_24 = (test_reg & 0xff000000);    //Size
  bits31_24 >>= 24;
  clrscr();
  cout<< "\n\n";
  cout<<"       ------------------------------------------------------------ "
          <<endl;
  cout<<"       |        |            L1 Instruction Cache                 |"
          << endl;
  cout<<"       ------------------------------------------------------------ "
          << endl;
  cout<<"       |        |    Size     |Associa -  | Lines  per  |Line Size |"
          << endl;
  cout<<"       |        |  (Kbytes)   |tivity     |      Tag    |  (bytes) |"
          << endl;
  cout<<"       ------------------------------------------------------------ "
          << endl;
  cout<<"       |        |Bits 31-24   |Bits 23-16 | Bits 15-8   | Bits 7-0 |"
          << endl;
  cout<<"       ------------------------------------------------------------"
          << endl;
  cout<<"       |  EDX   |     " << setw(2) << hex << bits31_24
          <<"      |    " << setw(2)<< hex << bits23_16 <<"     |      "
          << setw(2) << dec
          << bits15_8 << "     |     " << setw(2) << hex << bits7_0 << "   |"
                << endl;
  cout<<"       ------------------------------------------------------------"
          << endl;
  cout<<"       Note: " << endl;
  cout<<"            Full associativity is indicated by a value of OFFh."
          <<endl;
  cout<<"\n\n";
  cout.unsetf(ios::uppercase);
}
```

## CPUID Module (cpuid.cpp file)

```
#pragma inline
#include <fstream.h>
#include <iomanip.h>
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <conio.h>
#include <dos.h>
#include "DEFINES.H"

cpuid k86;                                  //Object of cpuid class
int main(void)
{
 int maxnum;                                //Case statement variable
 int result = 0;                            //Return value variable
 int func = 3;                              //Control loop variable

 result = k86.chkcpubit();                  //Check ID bit in EFLAGS
 if(result == -1) {
      clrscr();
      cout << "\n\n";
      cout << " CPUID instruction is not supported by this processor.";
      cout << "\n\n";
      exit(1);
 }
 else {
      result = k86.chkcpuid();              //Check vendor id string
      if(result == 1) {
        clrscr();
        cout << "\n\n";
        cout << "AMD-K86 CPU supporting CPUID is in place.";
        cout << "\n\n";
      }
      else {
        clrscr();
        cout << "\n\n";
        cout << "CPU supporting CPUID is in place.";
        cout << "\n\n";
        }
 }
//These are the standard functions
 k86.std_vendor_id_str();
 k86.std_cpu_signature();

//These are the extended functions
 for (maxnum=0; maxnum<=func; maxnum++)  {
       switch (maxnum) {
              case 0 : k86.ext_vendor_id_str(); //Vendor Identification String
                          break;
```

```
                case 1 : k86.ext_cpu_signature(); //Processor Signature
                            break;
                case 2 : k86.ext_cpu_name_str();  //Processor Name String
                            break;
                case 3 : k86.ext_cpu_cache_info();//Processor Cache Information
                            break;
        }
 }
        return 0;
}
```