Nx586$^{\text{â}}$ Processor Recognition Application Note

PRELIMINARY INFORMATION

NexGen, Incorporated.
1623 Buckeye Drive
Milpitas, CA 95035

ORDER # 754006-02

Contents

List of Figures

List Of Tables

Introduction

The NexGen Nx586® processor offers a powerful and affordable alternative to Intel's Pentium™ processor. The Nx586 processor is a 5th generation processor with full x86 binary compatibility. In order to properly identify NexGen processors and their features, NexGen is providing software that performs these functions. As the number of alternatives in the x86 market increases it is important for software to be able to identify the features and performance level associated with a given processor.

This application note explains the method for identifying a NexGen processor and its features. It provides a software routine necessary to perform this function. It also explains how this routine and the method it describes can be used by software developers in BIOS code, software applications, and utilities to properly identify current and future NexGen processors.

Processor Recognition

To best leverage existing processor recognition routines and minimize the effort for software developers, the NexGen processor recognition code is designed as an extension of the processor recognition code published by Intel in AP-485, *Intel Processor Identification With the CPUID Instruction Application Note.*

Figure 1 provides a flow chart of the process used to identify different processors. The areas in gray are those added to recognize NexGen processors. Note that there are two code additions to recognize NexGen processors.

The first is for older Nx586 processors. Since all older Nx586s use the same register and flag implementation as the 80386, Intel's recognition code will identify the Nx586 as an 80386. To identify the additional performance and features available, NexGen has developed code to distinguish an Nx586 processor from an 80386. This is done by using the fact that the 80386 and Nx586

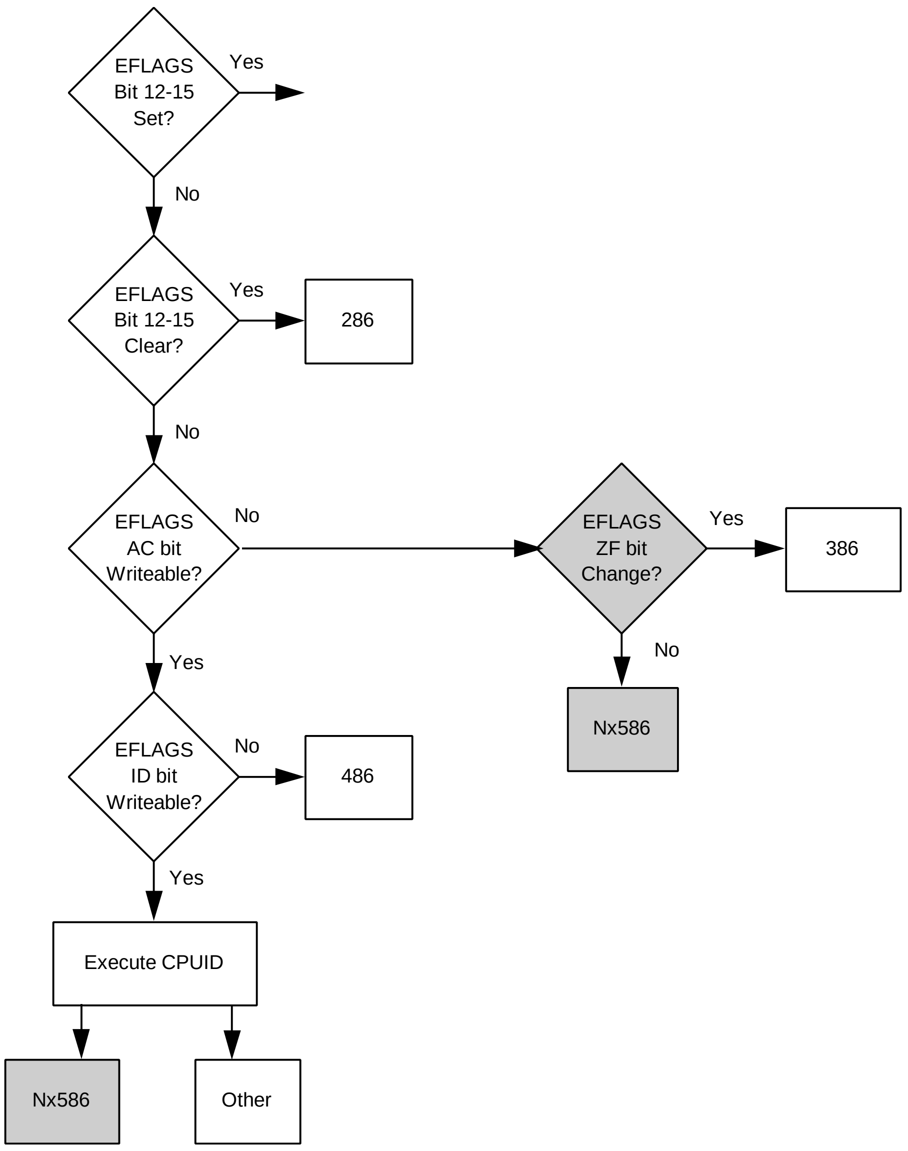

Figure 1 Flow Chart Of Processor Recognition Routine

processor affect the ZF flag (bit 6 of EFLAGs) differently as a result of a DIV instruction and specific operands. The Nx586 does not change the value of the ZF flag during the DIV while the 80386 changes the ZF flag according to the result of the execution.

The second piece of code uses the CPUID instruction to determine the type of processor and its features. NexGen supports the CPUID in newer versions of the Nx586 processor. To determine if the CPUID instruction is supported, software must test the ID bit (bit 21) in EFLAGS to determine if its value can be changed. The code example in this application note includes this test. Once the software determines that the CPUID instruction is available, it can execute this instruction to determine the processor's vendor, family, type, features, and other useful information. The section entitled "CPUID Instruction" describes the functionality of the CPUID instruction.

Recognition Program

The following code examples (Figure 2, Figure 3, and Figure 4) enable software to identify NexGen processors and the features that they support. These routines can be integrated with the recognition routines for other x86 processors to provide a complete solution for processor recognition.

An electronic copy of the code can be obtained from NexGen. The code is available as a self-extracting zip file, CPUID52.EXE. Table 1 provides the paths available for obtaining the code from NexGen.

		file area	filename
BBS	(408)955-1839	Techdesk	CPUID52.EXE
World Wide Web	http://www.nexgen.com	Support Desk	CPUID52.EXE
FTP	ftp.nexgen.com	Techdesk	CPUID52.EXE

Table 1 NexGen Online Resources

Figure 2 contains the file "cpuid.asm". This file contains two routines, "_get_nxcpu_type" and "_get_nxfpu", that have been written in assembly language. The "_get_nxcpu_type" routine implements the code necessary to identify NexGen processors. It first checks for the AC bit (bit 18) in EFLAGS. If the AC bit is not writeable, it then tests the ZF FLAG result from the DIV instruction. If the ZF FLAG is unchanged, it is an Nx586 processor.

If during the initial test, the AC bit is found to be writeable, the code immediately tests for the ID bit. If it is writeable, the code executes the CPUID instruction to identify the vendor, family, model, and features associated with the processor.

The “_get_nxfpu” routine implements the code necessary to determine if a floating point processor is present when the CPUID instruction is not supported. This routine tests for the presence of the floating point processor by testing the floating point status word.

Figure 3 contains the file “cpuclk.asm”. This file contains a single assembly language routine, “_Nx586_clock_rate” that determines the operating frequency for the Nx586 processor. This routine calculates the CPU clock rate by determining the time elapsed to execute a known number of CPU clock cycles. The frequency calculated to the nearest 1/10th MHz and is returned in the AX register as 10 times the number of MHz.

Figure 4 contains the file “nxcpu.c”. This file contains the C language program that calls the “_get_nxcpu_type” and “_get_nxfpu” routines to identify the type of processor and determine the presence of the numeric processor. It then prints the results to the screen. In addition, the program calls “_Nx586_clock_rate” to determine the processor’s operating frequency and display this information on the screen. Finally, if the CPUID instruction is available, this routine displays the vendor identification string, the processor signature (family, model, and stepping), and the feature flags.

Figure 2 CPUID.ASM

```
       page ,132
;****************************************************************************
;
;   NexGen, Inc.
;   1623 Buckeye Drive
;   Milpitas, CA 95035
;   Phone: (408)435-0202
;
;****************************************************************************

;****************************************************************************
; File: cpuid.asm
;      Revision: 1.0
;
; This sample file contains two routines: "_get_nxcpu_type" and "_get_nxfpu".
; "_get_nxcpu_type" identifies NexGen's processor and saves CPU information
; in the data segment, and "_get_nxfpu" tests if FPU is present.
;
;****************************************************************************

DOSSEG
.model  small
.386
CPU_ID  macro
        db      0fh, 0a2h
        endm

NONE            equ     0
PRESENT         equ     1
```

```
Nx586          equ    5
UNKNOWN        equ    0

.data
       public _nxcpu
       public _cputype
       public _cpuid_flag
       public _vendor_id
       public _cpu_signature
       public _features_ecx
       public _features_edx
       public _features_ebx
       public _nxfpu

_nxcpu         db     NONE                 ;default to none
_cputype       db     UNKNOWN              ;default to unknown
_cpuid_flag    db     NONE                 ;default to no CPUID
_vendor_id     db     "************"
_cpu_signature dd     0
_features_ecx  dd     0
_features_edx  dd     0
_features_ebx  dd     0
_nxfpu         db     NONE                 ;default to none
fp_status      dw     0
NexGen_id      db     "NexGenDriven"


.code
;==========================================================================
; _get_nxcpu_type
;      This routine identifies NexGen's processor type in following steps:
;
;      if (no AC flag) {      //current Nx586 does not support AC flag
;             set ZF=1;
;             execute DIV to result a none zero value;
;             if (ZF=0) {    //ZF is changed
;                    not a NexGen processor;
;                    exit;
;             } else {       //Nx586 does not change ZF on DIV instruction
;                    if (ID bit not writeable) {
;                           CPU is Nx586 with no CPUID support
;                    } else {              //Nx586 with CPUID support
;                           execute CPUID instruction;
;                           save CPU information;
;                    }
;             }
;      } else {
;             if (ID bit not writeable) {
;                    not a NexGen processor;
;             } else {       //NexGen future processors support CPUID
;                    execute CPUID instruction;
;                    save CPU information;
;             }
;      }
;
;==========================================================================
       public _get_nxcpu_type
_get_nxcpu_type       proc   near

       mov    byte ptr _nxcpu,PRESENT      ; default to present

; test AC bit on EFLAGS register
       mov    bx,sp          ; save the current stack pointer
       and    sp,not 3       ; align the stack to avoid AC fault
       pushfd                ;
       pop    eax            ; get the original EFLAGS
       mov    ecx,eax        ; save the original EFLAGS
       xor    eax,40000h     ; flip AC bit in EFLAGS
       push   eax            ; save for EFLAGS
       popfd                 ; copy it to EFLAGS
       pushfd                ;
       pop    eax            ; get the new EFLAGS value
       mov    sp,bx          ; restore stack pointer
       xor    eax,ecx        ; if the AC bit is unchanged
       je     test_zf        ;      goto second step
```

```
        jmp     nx_future_cpu

test_zf:
; test ZF on DIV instruction
        mov     ax,5555h        ; init AX with a non-zero value
        xor     dx,dx           ; set ZF=1
        mov     cx,2
        div     cx              ; Nx586 processor does not modify ZF on DIV
        jnz     not_nx_cpu      ; not a NexGen processor if ZF=0 (modified)

test_cpuid:
; test if CPUID instruction is available
; new Nx586 or future CPU supports CPUID instruction
        pushfd                  ; get EFLAGs
        pop     eax
        mov     ecx,eax         ; save it
        xor     eax,200000h     ; modify ID bit
        push    eax
        popfd                   ; save it in new EFLAGS
        pushfd                  ; get new EFLAGS
        pop     eax             ;
        xor     eax,ecx         ; is ID bit changed?
        jnz     cpuid_present   ; yes

        mov     byte ptr _cputype,Nx586       ; no, current Nx586
        jz      cpuid_exit      ; stop testing

nx_future_cpu:
; all NexGen's future processors feature a CPUID instruction
        mov     eax,ecx         ; get original EFLAGS
        xor     eax,200000h     ; modify ID bit
        push    eax
        popfd                   ; save it in new EFLAGS
        pushfd                  ; get new EFLAGS
        pop     eax             ;
        xor     eax,ecx         ; is ID bit changed?
        jz      not_nx_cpu      ; no, not a NexGen processor

cpuid_present:
; execute CPUID instruction to get vendor name, stepping and feature info
        xor     eax,eax
        CPU_ID
        mov     dword ptr _vendor_id,ebx
        mov     dword ptr _vendor_id[+4],edx
        mov     dword ptr _vendor_id[+8],ecx

        mov     bx,ds
        mov     es,bx
        mov     si,offset _vendor_id
        mov     di,offset NexGen_id
        mov     cx,12
        cld
        repe    cmpsb           ; compare vendor ID string
        jne     not_nx_cpu

        mov     byte ptr _cpuid_flag,PRESENT
        cmp     eax,1           ; check highest level
        jl      cpuid_exit

        mov     eax,1
        CPU_ID
        mov     _cpu_signature,eax
        mov     _features_ecx,ecx
        mov     _features_edx,edx
        mov     _features_ebx,ebx
        shr     eax,8
        and     al,0fh
        mov     _cputype,al
        jmp     cpuid_exit

not_nx_cpu:
        mov     byte ptr _nxcpu,NONE
cpuid_exit:
       ret
_get_nxcpu_type       endp

;==========================================================================
; _get_nxfpu
;      This procedure identifies NexGen's floating point processor by
; testing the floating point status word.
;
;==========================================================================
       public  _get_nxfpu
_get_nxfpu     proc    near

       mov     _nxfpu, PRESENT               ; default to present
       fninit                        ; reset fpu status word
       mov     fp_status,0aa55h
       fnstsw  fp_status
       mov     ax,fp_status
       cmp     al,0
       je      nxfpu_end
       mov     _nxfpu, NONE
nxfpu_end:

       ret
_get_nxfpu     endp


       end
```

Figure 3 CPUCLK.ASM

```
       page ,132
;*************************************************************************
;
;   NexGen, Inc.
;   1623 Buckeye Drive
;   Milpitas, CA 95035
;   Phone: (408)435-0202
;
;*************************************************************************

;*************************************************************************
; File: cpuclk.asm
;      Revision: 1.0
;
; This file contains a "C" callable routine:
;
;      1) _Nx586_clock_rate returns CPU clock rate in MHz*10 unit.
;         (i.e. A value of 600 means 60.0 MHz)
;
; The routine returns the result in AX register. You need to declare
; the function prototypes in the C program as:
;
;      extern unsigned _Nx586_clock_rate (void);
;
; The _Nx586_clock_rate returns the clock rate in MHz*10 unit.;
;
; Notice: these routines are coded for SMALL memory model.
; You have to change the .MODEL directive if your C program is using
; a different memory model. For example, the following directive will
; make routines callable from a C program in the LARGE memory model.
;
;      .MODEL LARGE, C
;
; To assemble this file into an object file if you are using Microsoft
; Assembler (5.10 or later), type this command:
;
;      masm cpuclk;
;
```

```
; If you are using Borland Turbo Assembler, type this command:
;
;       tasm cpuclk;
;
;
; Revision History:
;       1.0 - initial release
;
;***************************************************************************

.MODEL  small
.386p
.DATA
clkcnt  dw      16 dup (0)

.CODE

;===========================================================================
; _Nx586_clock_rate
;       This routine calculates the CPU clock rate by reading the time
; elapsed on a known number of CPU clock cycles. The total number of
; clock cycles is obtained from the cycle number defference of two
; instruction loops; a long cycle loop (DIV EBX) and a short cycle loop
; (DIV BX). The time elapsed on executing these number of cycles is
; the time difference of the long and short loop.
;
;       This routine gets the time difference of the two cycle loops for
; five times, and calculates their average. Then, the routine computes
; the CPU clock rate in MHz*100 unit, rounds off the last digit, and
; return the clock rate in MHz*10 unit. The calling program (C program)
; has to convert it to MHz unit.
;
; Input: none
; Output: AX = clock rate in MHz*10 unit
;===========================================================================
        PUBLIC  _Nx586_clock_rate
_Nx586_clock_rate       PROC
        push    ds
        push    es

        mov     ax,@data
        mov     ds,ax
        mov     es,ax
        ASSUME  DS:@data

        mov     cx,1            ; execute sub-routine once to make sure
                                ;   cache hit
        xor     al,al           ; a dummy call to clock routine
        call    getclk

        mov     di,offset clkcnt
        cld
        mov     cx,5            ; run clock detection for 5 times
nextcount:
        push    cx
                                ; for each time:
        mov     cx,400          ;  perform 400 short delay loops (DIV BX)
        mov     al,1            ;  select short cycle loop
        call    getclk          ;  get timer tic
        mov     bx,ax           ;  save it
        mov     cx,400          ;  perform 400 long delay loops (DIV EBX)
        xor     al,al           ;  select long cycle loop
        call    getclk          ;  get timer tic
        sub     ax,bx           ;  calculate time difference of the two loops
        stosw                   ;  save it
        pop     cx
        loop    nextcount

        xor     dx,dx           ; init DX:AX
        xor     ax,ax
        mov     cx,5
        mov     si,offset clkcnt
totalcount:                     ; sum the total time difference in DX:AX
```

```
        add     ax,[si]
        adc     dx,+0
        inc     si
        inc     si
        loop    totalcount

        mov     bx,5            ; calculate average time difference of
        div     bx              ;  each loop

        mov     bx,ax           ; calculate clock rate in MHz*100 unit
        mov     ax,64000        ; freq=(total cycle/time elapsed)*100
        mov     cx,1193         ;     =(("DIV EBX" clock - "DIV BX" clock)*100
        xor     dx,dx           ;      *400 loops)/(time difference/1.193 MHz)
        mul     cx              ;      *100
        div     bx              ;     =((34-18)*100*400/(timer difference/
                                ;      1193000))*100
                                ;     =64000*1193/time difference
        xor     dx,dx
        add     ax,5            ; round off the last digit
        adc     dx,+0
        mov     bx,10           ; disgard the last digit
        div     bx              ; return clock rate in MHz*10 unit
        pop     es
        pop     ds
        ret
_Nx586_clock_rate      ENDP

;=======================================================================
; Getclk
;
;       Get timer tics after executing one of two clock loops. The long
; clock loop performs 100 "DIV EBX" instructions, and the short clock
; loop executes 100 "DIV BX" instructions. A loop count is passed through
; CX register to extend the total delay time.
;
;       The time tics returned from long and short clock loops can be used
; for clock rate calculation. The time difference of two clock loops is
; exactly same as the total cycle difference of the two loops. The total
; cycle difference is:
;
;               (34-18)*100*loopcount
;
; The Nx586 processor uses 34 cycles to execute "DIV EBX" instruction, and
; uses 18 cycles to execute "DIV BX" instruction.
;
; Input: CX = loop count
;        AL = 0 to select long clock loop
;           = 1 to select short clock loop
;
; Output: AX = timer tic from timer 2
;=======================================================================

getclk  PROC    NEAR

        push    si
        push    di
        push    bx
        push    dx
        mov     bl,al           ; save cycle loop selection
        xor     dx,dx
        in      al,61h          ; get current value from port 61h
        and     al,0FCh         ; disable Gate2 and Speaker Data
        out     61h,al
        mov     al,0B4h         ; program timer 2 with mode 2 (rate generator)
        out     43h,al

        mov     al,0ffh         ; init timer 2 starting count
        out     42h,al          ; write LSB
        out     42h,al          ; then MSB

        in      al,61h          ; read port 61h again
        mov     di,ax           ; save this value for later use (to disarm
                                ;   timer 2
        or      al,1            ; enable timer 2 by enabling Gate2
```

```
       cmp     bl,0            ; check which loop to perform
       je      long_loop
       jmp     short_loop
long_loop:
       cli                     ; disable interrupt
       out     61h,al          ; arm timer 2
       mov     ebx,2           ; set divisor to a simple value
       xor     edx,edx         ; init EDX:EAX
       xor     eax,eax         ; NOTE: the time spent on instructions after
                               ;   arming the timer 2 and before the DIV loop
                               ;   will be eliminated by calculating the
                               ;   time difference of two cycle loops. (see
                               ;   short_loop below)
next_long:
       div     ebx             ; perform "DIV EBX" for 100 times
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
       div     ebx
```

```
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx
        div     ebx

        dec     cx
        jnz     next_long
        jmp     end_loop

short_loop:                     ; select short cycle loop
        cli                     ; disable interrupts
        out     61h,al          ; arm timer 2
        mov     ebx,2           ; init divisor to a simple value
        xor     edx,edx         ; init EDX:EAX
        xor     eax,eax         ; these three instructions are exactly same
                                ;   the ones used in the long cycle loop
next_short:
        div     bx              ; perform "DIV BX" for 100 times
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx
        div     bx

        dec     cx
        jnz     next_short
        jmp     end_loop        ; same JMP instruction as the one used in the
        nop                     ;    long loop
        nop
end_loop:
        mov     ax,di           ; retrieve saved value for port 61h
        out     61h,al          ; disarm timer 2
        sti                     ; enable interrupt
        in      al,42h          ; read LSB from timer 2
        xchg    ah,al
        in      al,42h          ; read MSB
        xchg    ah,al
        neg     ax              ; total count elasped

        pop     dx
        pop     bx
        pop     di
        pop     si

        ret
getclk  ENDP

        END
```

Figure 4 NXCPU.C

```
//***************************************************************************
//
//   NexGen, Inc.
//   1623 Buckeye Drive
//   Milpitas, CA 95035
//   Phone: (408)435-0202
//
//***************************************************************************

//***************************************************************************
// File: nxcpu.c
//   Revision: 1.0
//
// This sample C program identifies Nx586 processor and prints its information
// according to the data saved by the external procedures "get_nxcpu_type",
// "get_nxfpu", and "cpuclk". If the CPUID instruction is available, the
// vendor ID, family ID, stepping number and features supported will be
// displayed.
//
// The first two external functions can be found in CPUID.ASM. The
// CPUID.ASM is assembled in SMALL model, and should be linked with this
// program.
//
// The third routine, "Nx586_clock_rate", is found in CPUCLK.ASM. It should
// also be assembled in SMALL model, and linked with this program.
//
// Revision History:
//     1.0 - initial release
//
//***************************************************************************


#include <stdio.h>

extern  char nxcpu;
extern  char cputype;
extern  char cpuid_flag;
extern  unsigned long cpu_signature;
extern  unsigned long features_ecx;
extern  unsigned long features_edx;
extern  unsigned long features_ebx;
extern  char nxfpu;
extern  void get_nxcpu_type(void);
extern  void get_nxfpu(void);
extern  int Nx586_clock_rate(void);

void main (void)
{
        get_nxcpu_type();
        get_nxfpu();
        print_cpu_info();
}

print_cpu_info()
{
        if (!nxcpu) {
                printf ("This system does not have a NexGen processor.\n");
                exit(-1);
        }
        printf ("This system has an ");
        switch (cputype) {
                case 5:
                   printf ("Nx586[R] processor ");
                   if (nxfpu)
                       printf ("and a floating point processor");
                   printf ("\n");
                           printf    ("\nProcessor    running    at    %d    MHz\n",
(int)Nx586_clock_rate()/10);
                   printf ("\n");
                   if (cpuid_flag)
                       print_id_info();             //print more CPU information
                   break;
                default:
                       //reserved for future expansion
                   break;
        }
}

print_id_info()
{
        printf ("Vendor ID: NexGenDriven\n");
        printf ("Processor Family: %x\n",(char)((cpu_signature>>8) & 0xff));
        printf ("Stepping: %x\n",(char)(cpu_signature & 0xf));
        printf ("Feature Flags: %x\n",(char)(features_edx & 1));
}
```

CPUID Instruction

The CPUID instruction is an application level instruction that software can execute to identify the processor and its feature set. It can be executed from any privilege level. Software can use this information to tune functionality for the specific processor and its features.

Not all processors implement the CPUID instruction. Before executing the instruction, software should first test to see if the instruction exists. Existence of the CPUID instruction is indicated by the ID bit (21) in the EFLAGS register. If this bit is writeable, the CPUID instruction exists.

Opcode: 0F A2

Input: EAX

Output: EAX, EBX, ECX, EDX

Function:

EAX = 0:

EAX = Highest input value recognized by CPUID instruction

EBX, EDX, ECX = Vendor identification string

EAX = 1:

EAX = Processor signature

EBX = Reserved

ECX = Reserved

EDX = Feature flags

EAX > 1:

EAX = Undefined

EBX = Undefined

ECX = Undefined

EDX = Undefined

Highest Input Value:

The highest input value recognized by the CPUID instruction in the Nx586 or Nx686 is 1. If an input value greater than 1 is used the values returned in EAX, EBX, ECX, and EDX are undefined. Future processors may implement higher values and the results returned by these values will be defined at that time.

Vendor Identification String:

The vendor identification string identifies NexGen as the vendor for the CPU. It does so by returning "NexGenDriven" in the EBX, EDX, and ECX registers.

EBX = 4778654Eh (GxeN)

EDX = 72446E65h (rDne)

ECX = 6E657669h (nevi)

Processor Signature:

The processor signature identifies the specific CPU by providing information regarding its type, family, model, and stepping revision. The information is formatted as follows:

EAX[0:3] = Stepping Revision

EAX[4:7] = CPU Model

EAX[8:11] = CPU Family

EAX[12:31] = Reserved

Feature Flags:

The feature flags indicate the existence or presence of specific features. In most cases a "1" indicates the feature is present. The following is an explanation of the feature flags currently defined. Reserved bits will be used in the future for new features as they are added.

EDX[0] = Floating Point Unit (1 indicates floating point unit is present, 0 indicates no floating point unit)

EDX[1:31] = Reserved

Note: All registers and bits marked "Reserved" should return 0.

Table 2 NexGen Processor Signatures

CPU Family	CPU Model	Stepping Revision[1]	Description
0101	0000	xxxx	Nx586 Processor
0110	0000	xxxx	Nx686 Processor

Notes:

1. Contact NexGen for specific stepping revision information.