# Chapter 10 -- Pentium Architecture

It is time to stop programming in SASM, and get to the real stuff:
Pentium assembly language.

Notes ahead of time:

-- The assembly language details presented are not specific to just the Pentium architecture. They are applicable to the 486 and to the Pentium II and the Pentium Pro. Many details also work for the earlier Intel architectures.

-- I'm only presenting a subset of the Pentium instruction set. The real instruction set is REALLY large, and contains many instructions that no one really uses anymore. We don't cover those "obsolete" instructions. We also don't cover the instructions that only the operating system would use.

## About the Pentium Architecture

-- It is not a load/store architecture.

-- The instruction set is huge! We go over only a fraction of the instruction set. The text only presents a fraction.

-- There are lots of restrictions on how instructions/operands are put together, but there is also an amazing amount of flexibility.

### Registers

The Intel architectures as a set just do not have enough registers to satisfy most assembly language programmers. Still, the processors have been around for a LONG time, and they have a sufficient number of registers to do whatever is necessary.

For our (mostly) general purpose use, we get

| 32-bit | 16-bit | 8-bit (high part of 16) | 8-bit (low part of 16) |
|---|---|---|---|
| EAX | AX | AH | AL |
| EBX | BX | BH | BL |
| ECX | CX | CH | CL |
| EDX | DX | DH | DL |

and

| | |
|---|---|
| EBP | BP |
| ESI | SI |

```
 EDI         DI
 ESP         SP
```

There are a few more, but we won't use or discuss them. They are only used for memory accessability in the segmented memory model.

Using the registers:
As an operand, just use the name (upper case and lower case both work interchangeably).

EBP is a frame pointer (see Chapter 11).
ESP is a stack pointer (see Chapter 11).

Oddities:
This is the only architecture that I know of where the programmer can designate part of a register as an operand. On ALL other machines, the whole register is designated and used.

ONE MORE REGISTER:
Many bits used for controlling the action of the processor and setting state are in the register called EFLAGS. This register contains the condition codes:

```
  OF  Overflow flag
  SF  Sign flag
  ZF  Zero flag
  PF  Parity flag
  CF  Carry flag
```

The settings of these flags are checked in conditional control instructions. Many instructions set one or more of the flags. (Note that we only utilized SF and ZF in SASM.)

There are many other bits in the EFLAGS register: TO BE DISCUSSED LATER.

The use of the EFLAGS register is implied (rather than explicit) in instructions.

## Accessing Memory

There are 2 memory models supported in the Pentium architecture. (Actually it is the 486 and more recent models that support 2 models.)

In both models, memory is accessed using an address. It is the way that addresses are formed (within the processor) that differs in the 2 models.

### FLAT MEMORY MODEL

-- The memory model that we use. AND, the memory model that every

other manufactures' processors also use.

--

SEGMENTED MEMORY MODEL

-- Different parts of a program are assumed to be in their own, set-aside portions of memory. These portions are called segments.

-- An address is formed from 2 pieces: a segment location and an offset within a segment.

Note that each of these pieces can be shorter (contain fewer bits) than a whole address. This is much of the reason that Intel chose this form of memory model for its earliest single-chip processors.

-- There are segments for:

code
data
stack
other

-- Which segment something is in can be implied by the memory access involved. An instruction fetch will always be looking in the code segment. A push instruction (we'll talk about this with chapter 11) always accesses the stack segment. Etc.

Addressing Modes
---

Some would say that the Intel architectures only support 1 addressing mode. It looks (something like) this:

effective address = base reg + (index reg x scaling factor) + displacement

where
base reg is EAX, EBX, ECX, EDX or ESP or EBP
index reg is EDI or ESI
scaling factor is 1, 2, 4, or 8

The syntax of using this (very general) addressing mode will vary from system to system. It depends on the preprocessor and the syntax accepted by the assembler.

For our implementation, an operand within an instruction that uses this addressing mode could look like
[EAX][EDI*2 + 80]

The effective address calculated with be the contents of register EDI multiplied times 2 added to the constant 80, added to the contents of register EAX.

There are extremely few times where a high-level language

compiler can utilize such a complex addressing mode. It is
much more likely that simplified versions of this mode
will be used.

SOME ADDRESSING MODES

-- register mode --
The operand is in a register. The effective address is the
register (wierd).

Example instruction:

```
mov  eax, ecx
```

Both operands use register mode. The contents of register ecx
is copied to register eax.

-- immediate mode --
The operand is in the instruction. The effective address is within
the instruction.

Example instruction:

```
mov  eax, 26
```

The second operand uses immediate mode. Within the instruction
is the operand. It is copied to register eax.

-- register direct mode --
The effective address is in a register.

Example instruction:

```
mov  eax, [esp]
```

The second operand uses register direct mode. The contents of
register esp is the effective address. The contents of memory
at the effective address are copied into register eax.

-- direct mode --
The effective address is in the instruction.

Example instruction:

```
mov  eax, var_name
```

The second operand uses direct mode. The instruction contains
the effective address. The contents of memory
at the effective address are copied into register eax.

-- base displacement mode --
The effective address is the sum of a constant and the contents
of a register.

Example instruction:

```
      mov  eax, [esp + 4]

      The second operand uses base displacement mode.  The instruction
      contains a constant.  That constant is added to the contents
      of register esp to form an effective address.  The contents
      of memory at the effective address are copied into register eax.

  --  base-indexed mode --  (Intel's name)
    The effective address is the sum of the contents of two registers.

    Example instruction:

      mov  eax, [esp][esi]

      The contents of registers esp and esi are added to form an
      effective address.  The contents of memory at the effective
      address are copied into register eax.

      Note that there are restrictions on the combinations of registers
      that can be used in this addressing mode.

  --  PC relative mode --
    The effective address is the sum of the contents of the PC and
    a constant contained within the instruction.

    Example instruction:

      jmp  a_label

      The contents of the program counter is added to an offset that
      is within the machine code for the instruction.  The resulting
      sum is placed back into the program counter.  Note that from the
      assembly language it is not clear that a PC relative addressing
      mode is used.  It is the assembler that generates the offset
      to place in the instruction.



 Instruction Set
 ---------------

Generalities:
 -- Many (most?) of the instructions have exactly 2 operands.
    If there are 2 operands, then one of them will be required
    to use register mode, and the other will have no restrictions
    on its addressing mode.

 -- There are most often ways of specifying the same instruction
    for 8-, 16-, or 32-bit oeprands.  I left out the 16-bit ones
    to reduce presentation of the instruction set.  Note that
    on a 32-bit machine, with newly written code, the 16-bit form
    will never be used.


Meanings of the operand specifications:
   reg - register mode operand, 32-bit register
   reg8 - register mode operand, 8-bit register
   r/m - general addressing mode, 32-bit
   r/m8 - general addressing mode, 8-bit
```

```
immed - 32-bit immediate is in the instruction
immed8 - 8-bit immediate is in the instruction
m - symbol (label) in the instruction is the effective address


Data Movement
-------------

   mov   reg, r/m                ; copy data
         r/m, reg
         reg, immed
         r/m, immed

   movsx reg, r/m8               ; sign extend and copy data

   movzx reg, r/m8               ; zero extend and copy data

   lea   reg, m                  ; get effective address
      (A newer instruction, so its format is much restricted
       over the other ones.)

       EXAMPLES:

       mov EAX, 23  ; places 32-bit 2's complement immediate 23
                    ; into register EAX
       movsx ECX, AL  ; sign extends the 8-bit quantity in register
                      ; AL to 32 bits, and places it in ECX
       mov [esp], -1  ; places value -1 into memory, address given
                      ; by contents of esp
       lea EBX, loop_top ; put the address assigned (by the assembler)
                         ; to label loop_top into register EBX



Integer Arithmetic
------------------

   add   reg, r/m                ; two's complement addition
         r/m, reg
         reg, immed
         r/m, immed

   inc   reg                     ; add 1 to operand
         r/m

   sub   reg, r/m                ; two's complement subtraction
         r/m, reg
         reg, immed
         r/m, immed

   dec   reg                     ; subtract 1 from operand
         r/m

   neg   r/m                     ; get additive inverse of operand

   mul   eax, r/m                ; unsigned multiplication
                                 ; edx||eax <- eax * r/m

   imul   r/m                    ; 2's comp. multiplication
                                 ; edx||eax <- eax * r/m
          reg, r/m               ; reg <- reg * r/m
```

```
          reg, immed              ; reg <- reg * immed

   div    r/m                     ; unsigned division
                                  ; does edx||eax / r/m
                                  ; eax <- quotient
                                  ; edx <- remainder

   idiv    r/m                    ; 2's complement division
                                  ; does edx||eax / r/m
                                  ; eax <- quotient
                                  ; edx <- remainder

   cmp    reg, r/m                ; sets EFLAGS based on
          r/m, immed              ; second operand - first operand
          r/m8, immed8
          r/m, immed8             ; sign extends immed8 before subtract


      EXAMPLES:

      neg [eax + 4]     ; takes doubleword at address eax+4
                        ;   and finds its additive inverse, then places
                        ;   the additive inverse back at that address
                        ;   the instruction should probably be
                        ;      neg  dword ptr [eax + 4]

      inc ecx           ; adds one to contents of register ecx, and
                        ;   result goes back to ecx


Logical
-------

   not    r/m                     ; logical not

   and    reg, r/m                ; logical and
          reg8, r/m8
          r/m, reg
          r/m8, reg8
          r/m, immed
          r/m8, immed8

   or     reg, r/m                ; logical or
          reg8, r/m8
          r/m, reg
          r/m8, reg8
          r/m, immed
          r/m8, immed8

   xor    reg, r/m                ; logical exclusive or
          reg8, r/m8
          r/m, reg
          r/m8, reg8
          r/m, immed
          r/m8, immed8

   test   r/m, reg                ; logical and to set EFLAGS
          r/m8, reg8
          r/m, immed
```

```
          r/m8, immed8
```

```
      EXAMPLES:

      and edx, 00330000h   ; logical and of contents of register
                           ;   edx (bitwise) with 0x00330000,
                           ;   result goes back to edx
```

## Floating Point Arithmetic

Since the newer architectures have room for floating point hardware on chip, Intel defined a simple-to-implement extension to the architecture to do floating point arithmetic. In their usual zeal, they have included MANY instructions to do floating point operations.

The mechanism is simple. A set of 8 registers are organized and maintained (by hardware) as a stack of floating point values. ST refers to the stack top. ST(1) refers to the register within the stack that is next to ST. ST and ST(0) are synonyms.

There are separate instructions to test and compare the values of floating point variables.

```
   finit                        ; initialize the FPU

   fld   m32                    ; load floating point value
         m64
         ST(i)

   fldz                         ; load floating point value 0.0

   fst   m32                    ; store floating point value
         m64
         ST(i)

   fstp  m32                    ; store floating point value
         m64                    ;   and pop ST
         ST(i)

   fadd  m32                    ; floating point addition
         m64
         ST, ST(i)
         ST(i), ST

   faddp ST(i), ST              ; floating point addition
                                ;   and pop ST

         ETC. (see p.201-202)
```

## I/O

---
The only instructions which actually allow the reading and
writing of I/O devices are priviledged. The OS must handle
these things. But, in writing programs that do something
useful, we need input and output. Therefore, there are some
simple macros defined to help us do I/O.

These are used just like instructions.

```
   put_ch  r/m           ; print character in the least significant
                         ;   byte of 32-bit operand

   get_ch  r/m           ; character will be in AL

   put_str m             ; print null terminated string given
                         ; by label m
```

Control Instructions
---
These are the same control instructions that all started with
the character 'b' in SASM.

```
   jmp   m               ; unconditional jump
   jg    m               ; jump if greater than 0
   jge   m               ; jump if greater than or equal to 0
   jl    m               ; jump if less than 0
   jle   m               ; jump if less than or equal to 0
```

ETC. (see p. 205)