A Guide to Programming Pentium/Pentium Pro Processors

Kai Li, Princeton University

The goal of this documentation is to provide a brief and concise documentation about Pentium PC architectures. It has a short description about the Intel Pentium and Pentium Pro processors and a brief introduction to assembly programming with the Gnu assembler.

Two useful reference books are *Pentium Pro Family Developers Manual, Volume 2: Programmer's Reference Manual, Intel Corporation, 1996,* and *Pentium Pro Family Developers Manual, Volume 3: Operating System Writer's Manual, Intel Corporation, 1996.* The on-line versions are available at http://www.x86.org/intel.doc/intelDocs.html.

1 Pentium/Prentium Pro Processor

1.1 Modes

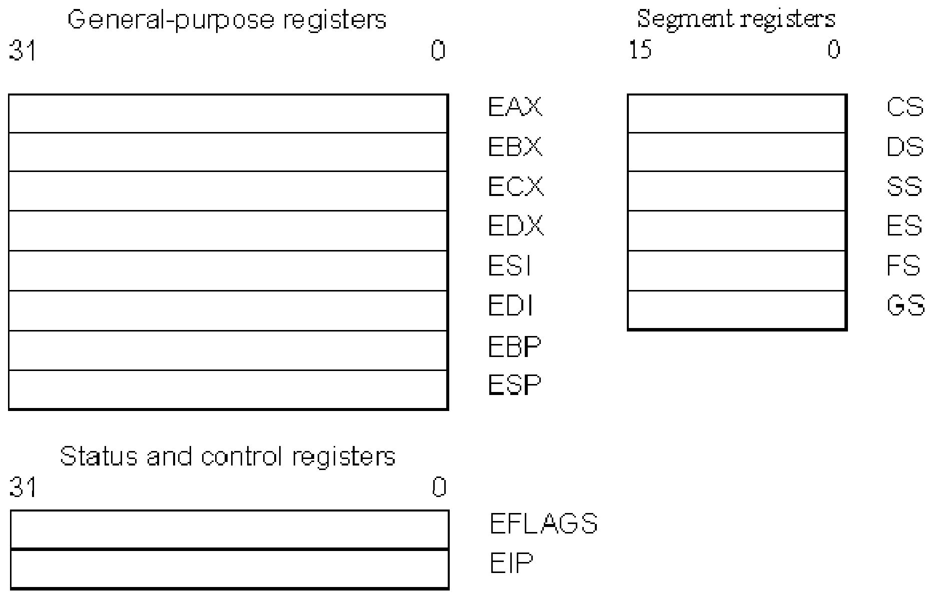
The *Pentium* and Pentium Pro processor has three operating modes:

- **Real-address mode.** This mode lets the processor to address "real" memory address. It can address up to 1Mbytes of memory (20-bit of address). It can also be called "unprotected" mode since operating system (such as DOS) code runs in the same mode as the user applications. Pentium and Prentium Pro processors have this mode to be compatible with early Intel processors such as 8086. The processor is set to this mode following by a power-up or a reset and can be switched to protected mode using a single instruction.
- **Protected mode.** This is the preferred mode for a modern operating system. It allows applications to use virtual memory addressing and supports multiple programming environment and protections.
- **System management mode.** This mode is designed for fast state snapshot and resumption. It is useful for power management.

There is also a virtual-8086 mode that allows the processor to execute 8086 code software in the protected, multi-tasking environment.

1.2 Register Set

There are three types of registers: general-purpose data registers, segment registers, and status and control registers. The following figure shows these registers:

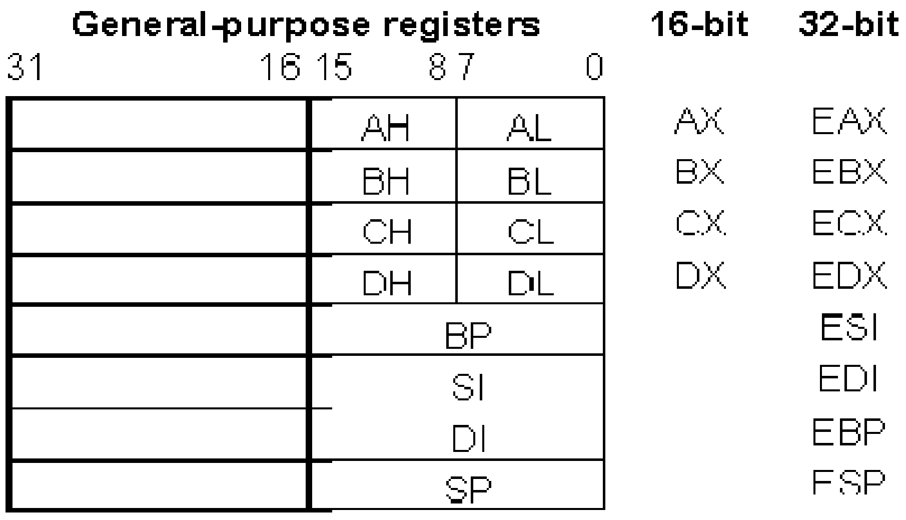


General-purpose Registers

The eight 32-bit general-purpose data registers are used to hold operands for logical and arithmetic operations, operands for address calculations and memory pointers. The following shows what they are used for:

- EAX—Accumulator for operands and results data.
- EBX—Pointer to data in the DS segment.
- ECX—Counter for string and loop operations.
- EDX—I/O pointer.
- ESI—Pointer to data in the segment pointed to by the DS register; source pointer for string operations.
- EDI—Pointer to data (or destination) in the segment pointed to by the ES register; destination pointer for string operations.
- ESP—Stack pointer (in the SS segment).
- EBP—Pointer to data on the stack (in the SS segment).

The following figure shows the lower 16 bits of the general-purpose registers can be used with the names AX, BX, CX, DX, BP, SP, SI, and DI (the names for the corresponding 32-bit ones have a prefix "E" for "extended"). Each of the lower two bytes of the EAX, EBX, ECX, and EDX registers can be referenced by the names AH, BH, CH, and DH (high bytes) and AL, BL, CL, and DL (low bytes).



Segment Registers

There are six segment registers that hold 16-bit segment selectors. A segment selector is a special pointer that identifies a segment in memory. The six segment registers are:

- CS: code segment register
- SS: stack segment register
- DS, ES, FS, GS: data segment registers

Four data segment registers provide programs with flexible and efficient ways to access data.

Modern operating system and applications use the (unsegmented) memory model— all the segment registers are loaded with the same segment selector so that all memory references a program makes are to a single linear-address space.

When writing application code, you generally create segment selectors with assembler directives and symbols. The assembler and/or linker then creates the actual segment selectors associated with these directives and symbols. If you are writing system code, you may need to create segment selectors directly. (A detailed description of the segment-selector data structure is given in Chapter 3, Protected-Mode Memory Management, of the Pentium Pro Family Developer’s Manual, Volume 3.)

Project 1 uses the real-address mode and needs to set up the segment registers properly.

EFLAGS Register

The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. The following shows the function of EFLAGS register bits:

Function	EFLAG Register bit or bits
ID Flag (ID)	21 (system)
Virtual Interrupt Pending (VIP)	20 (system)
Virtual Interrupt Flag (VIF)	19 (system)
Alignment check (AC)	18 (system)
Virtual 8086 Mode (VM)	17 (system)
Resume Flag (RF)	16 (system)
Nested Task (NT)	14 (system)
I/O Privilege Level (IOPL)	13 to 12 (system)
Overflow Flag (OF)	11 (system)
Direction Flag (DF)	10 (system)
Interrupt Enable Flag (IF)	9 (system)
Trap Flag (TF)	8 (system)
Sign Flag (SF)	7 (status)
Zero Flag (ZF)	6 (status)
Auxiliary Carry Flag (AF)	4 (status)
Parity Flag (PF)	2 (status)
Carry Flag (CF)	0 (status)

Bits 1, 3, 5, 15, and 22 through 31 of this register are reserved. To understand what these fields mean and how to use them, please see Section 3.6.3 and 3.6.4 in *Pentium Pro Family Developers Manual, Volume 2: Programmer's Reference Manual.*

EIP Register (Instruction Pointer)

The EIP register (or instruction pointer) can also be called "program counter." It contains the offset in the current code segment for the next instruction to be executed. It is advanced from one instruction boundary to the next in straight-line code or it is moved ahead or backwards by a number of instructions when executing JMP, Jcc, CALL, RET, and IRET instructions. The EIP cannot be accessed directly by software; it is controlled implicitly by control-transfer instructions (such as JMP, *Jcc*, CALL, and RET), interrupts, and exceptions. The EIP register can be loaded indirectly by modifying the value of a return instruction pointer on the procedure stack and executing a return instruction (RET or IRET).

Note that the value of the EIP may not match with the current instruction because of instruction prefetching. The only way to read the EIP is to execute a CALL instruction and then read the value of the return instruction pointer from the procedure stack.

The x86 processors also have control registers that are not used in project 1, and thus omitted in this document.

1.3 Addressing

Bit and Byte Order

Pentium and Pentium-Pro processors use "little endian" as their byte order. This means that the bytes of a word are numbered starting from the least significant byte and that the least significant bit starts of a word starts in the least significant byte.

Data Types

The Pentium/Pentium Pro provides four data types: a byte (8 bits), a word (16 bits), a doubleword (32 bits), and a quadword (64 bits). Note that a doubleword is equivalent to "long" in Gnu assembler.

Memory Addressing

One can use either flat memory model or segmented memory mode. With the flat memory model, memory appears to a program as a single, continuous address space, called a linear address space. Code (a program’s instructions),
data, and the procedure stack are all contained in this address space. The linear address space is byte addressable, with addresses running contiguously from 0 to 2^{32-1}.

With the segmented memory mode, memory appears to a program as a group of independent address spaces called segments. When using this model, code, data, and stacks are typically contained in separate segments. To address a byte in a segment, a program must issue a logical address, which consists of a segment selector and an offset. (A logical address is often referred to as a far pointer.) The segment selector identifies the segment to be accessed and the offset identifies a byte in the address space of the segment. The programs running on a Pentium Pro
processor can address up to 16,383 segments of different sizes and types. Internally, all the segments that are defined for a system are mapped into the processor’s linear address space. So, the processor translates each logical address into a linear address to access a memory location. This translation is transparent to the application program.

1.4 Processor Reset

A cold boot or a warm boot can reset the CPU. A cold boot is powering up a system whereas a warm boot means that when three keys CTRL-ALT-DEL are all pressed together, the keyboard BIOS will set a special flag and resets the CPU.

Upon reset, the processor sets itself to real-mode with interrupts disabled and key registers set to a known state. For example, the state of the EFLAGS register is 00000002H and the memory is unchanged. Thus, the memory will contain garbage upon a cold boot. The CPU will jump to the BIOS (Basic Input Output Services) to load the bootstrap loader program from the diskette drive or the hard disk and begins execution of the loader. The BIOS loads the bootstrap loader into the fixed address 0:7C00 and jumps to the starting address.

2 Assembly Programming

It often takes a while to master the techniques to program in assembly language for a particular machine. On the other hand, it should not take much time to assembly programming for Pentium or Pentium Pro processors if you are familiar with another processor.

This section assumes that you are already familiar with Gnu assembly syntax (learned from the course *Introduction to Programming Systems* or its equivalent). The simplest way to learn assembly programming is to compile a simple C program into its assembly source code as a template. For example, gcc -S -c foo.c will compile foo.c its assembly source foo.s. The source code will tell you common opcodes, directives and addressing syntax.

The goal of this section is to answer some frequently encountered questions and provide pointers to related documents.

2.1 Memory operands

Pentium and Pentium Pro processors use segmented memory architecture. It means that the memory locations are referenced by means of a segment selector and an offset:

- The segment selector specifies the segment containing the operand, and
- The offset (the number of bytes from the beginning of the segment to the first byte of the operand) specifies the linear or effective address of the operand.

The segment selector can be specified either implicitly or explicitly. The most common method of specifying a segment selector is to load it in a segment register and then allow the processor to select the register implicitly, depending on the type of operation being performed. The processor automatically chooses a segment according to the following rules:

- Code segment register CS for instruction fetches
- Stack segment register SS for stack pushes and pops as well as references using ESP or EBP as a base register
- Data segment register DS for all data references except when relative to stack or string destination
- Data segment register ES for the destinations of string instructions

The offset part of the memory address can be specified either directly as a static value (called a *displacement*) or through an address computation made up of one or more of the following components:

- Displacement—An 8-, 16-, or 32-bit value.
- Base—The value in a general-purpose register.
- Index—The value in a general-purpose register except EBP.
- Scale factor—A value of 2, 4, or 8 that is multiplied by the index value.

An effective address is computed by:

$$Offset = Base + (Index \times Scale) + displacement$$

The offset which results from adding these components is called an *effective address* of the selected segment. Each of these components can have either a positive or negative (2's complement) value, with the exception of the scaling factor.

	Intel	**AT&T (Gnu Syntax)**
Immediate operands	Undelimited e.g.: `push 4` `mov ebx, d00ah`	Preceded by "$" e.g.:`push $4` `movl $0xd00a, %eax`
Register operands	Undelimited e.g.: `eax`	Preceded by "%" e.g.: `%eax`
Argument order (e.g. adds the address of C variable "foo" to register EAX)	Dest, source [, source2] e.g.: `add eax, _foo`	Source, [source,] dest e.g.: `addl $_foo, %eax`
Single-size operands	Implicit with register name, **byte ptr**, **word ptr**, or **dword ptr** e.g.: `mov al, foo`	opcode{b,w,l} e.g.: `movb foo, %al`
Address a C variable "foo"	`[_foo]`	`_foo`
Address memory pointed by a register (e.g. EAX)	`[eax]`	`(%eax)`
Address a variable offset by a value in the register	`[eax + _foo]`	`_foo(%eax)`
Address a value in an array "foo" of 32-bit integers	`[eax*4+foo]`	`_foo(,%eax,4)`
Equivalent to C code *(p+1)	If EAX holds the value of p, then `[eax+1]`	`1(%eax)`

2.2 Instruction Syntax

There are two conventions about their syntax and representations: Intel and AT&T. Most documents including those at http://www.x86.org use the Intel convention, whereas the Gnu assembler uses the AT&T convention. The main differences are:

In addition, with the AT&T syntax, the name for a long JUMP is `ljmp` and long CALL is `lcall`.

Section 6-6 of *Pentium Pro Family Developers Manual, Volume 2: Programmer's Reference Manual* has a complete list of the Pentium Pro instructions. Section 11 provides the detailed description for each instruction.
The instruction names obviously use the Intel convention and you need to convert them to the AT&T syntax.

2.3 Assembler Directives

The Gnu assemler directives are machine independent, so your knowledge about assembly programming applies.
All directive names begin with a period "." and the rest are letters in lower case. Here are some examples of commonly used directives:

> `.ascii "string"` defines an ASCII string `"string"`
>
> `.byte 10, 13, 0` defines three bytes
>
> `.word 0x0456, 0x1234` defines two words
>
> `.long 0x001234, 0x12345` defines two long words
>
> `.equ STACK_SEGMENT, 0x9000` sets symbol `STACK_SEGMENT` the value `0x9000`
>
> `.globl symbol` makes "`symbol`" global (useful for defining global labels and procedure names)
>
> `.code16` tells the assembler to insert the appropriate override prefixes so the code will run in real mode.

When using directives to define a string, bytes or a word, you often want to make sure that they are aligned to 32-bit long word by padding additional bytes.

2.4 Inline Assembly

The most basic format of inline assembly code into your the assembly code generated by the gcc compiler is to use

```
      asm( "assembly-instruction" );
```

where `assembly-instruction` will be inlined into where the `asm` statement is. This is a very convenient way to inline assembly instructions that require no registers. For example, you can use

```
      asm( "cli" );
```

to clear interrupts and

```
      asm( "sti" );
```

to enable interrupts.

The general format to write inline assembly code in C is:

```
      asm( "statements": output_regs: input_regs: used_regs);
```

where `statements` are the assembly instructions. If there are more than one instruction, you can use "`\n\t`" to separate them to make them look pretty. "`input_regs`" tells gcc compiler which C variables move to which registers. For example, if you would like to load variable "foo" into register EAX and "bar" into register ECX, you would say

```
      : "a" (foo), "c" (bar)
```

gcc uses single letters to represent all registers:

Single Letters	**Registers**
`a`	`eax`
`b`	`ebx`
`c`	`ecx`
`d`	`edx`
`S`	`esi`
`D`	`edi`
`I`	constant value (0 to 31)
`q`	allocate a register from EAX, EBX, ECX, EDX
`r`	allocate a register from EAX, EBX, ECX, EDX, ESI, EDI

Note that you cannot specify register AH or AL this way. You need to get to EAX first and then go from there.

"`output_regs`" provides output registers. A convenient way to do this is to let gcc compiler to pick the registers for you. You need to say "`=q`" or "`=r`" to let gcc compiler pick registers for you. You can refer to the first allocated register with "`%0`", second with "`%1`", and so on, in the assembly instructions. If you refer to the registers in the input register list, you simply say "`0`" or "`1`" without the "`%`" prefix.

"`used_regs`" lists the registers that are used (or clobbered) in the assembly code.

To understand exactly how to do this, please try to use gcc to compile a piece of C code containing the following inline assembly:

```
asm ("leal (%1,%1,4), %0"
     : "=r" (x_times_5)
     : "r" (x) );
```

and

```
asm ("leal (%0,%0,4), %0"
    : "=r" (x)
    : "0" (x) );
```

Also, to avoid the gcc compiler's optimizer to remove the assembly code, you can put in keyword volitale to ensure your inline. Here are some macro code examples:

```
#define disable() __asm__ __volatile__ ("cli");

#define enable() __asm__ __volatile__ ("sti");
```

to disable and enable interrupts.

To see more information, you can read another guide that has more information about inline assembly.