

# Chapter 1

## Introduction

### 1.1 Overview

This chapter discusses a brief introduction to the organization of a single-processor architecture, fundamental concepts in computer systems, and summarizes the main features of the Intel Architecture family from the 8/16-bit 8086/88 to the Intel's 64-bit architecture based on the Itanium processor. The notion of the software model as a set of resources available to programmers is discussed. The software model of the Intel 8086/88 processor and the Pentium 4 is reviewed. Some features of modern architectures exemplified by the Pentium 4 machine are highlighted and contrasted with the Intel 8086/88 machine.

### 1.2 Computer Organization

A single-processor computing system as shown in Fig. 1.2 contains three main components: the *cpu* (*central processing unit*), *main memory*, and *I/O devices*. The main function of the *cpu* is to interpret and execute programs currently resident in memory and fetched by the *cpu* instruction by instruction. The *cpu* is normally contained in a single chip on a 0.25" square of silicon. Internally the *cpu* is organized into three main components: at least one arithmetic logic unit (*alu*), a set of registers (*register file*), and a control unit. These components are interconnected through an internal bus and their coordinated activity provides the *cpu* with its entire functionality. The *alu* performs an array of arithmetic and logic operations.

At least one operation is performed at a time as part of the instruction currently in execution. Registers store operands or data fetched from memory or produced by the *alu*. All data transfer and computing activity within the *cpu* is controlled by the *control unit*; the control unit controls fetching instructions, selection of *alu* operations, selection of source and destination registers, selection of the appropriate *alu* operation, and the storing of results in memory if necessary.

The memory unit stores instructions and data. A section of memory referred to as Random Access Memory (RAM) is dedicated to the temporary storage of data and code for programs currently in execution. Another section of memory referred to as Read Only Memory (ROM) stores utility programs such as the bootstrap routine, I/O services routines, etc. Normally the *capacity* of memory is given in terms of the number of bytes it can store; this refers to the number of locations in bytes that are directly addressable; A *byte* typically regarded as the storage unit consists of 8 bits; therefore, most memory reference operations transfer data in terms of bytes. Applications may require data in 16-bit sizes and 2-byte data units are fetched from or stored in memory; 16-bit data is referred to as a *word*. Thus, applications that use 32 bits fetch 4 bytes or a *double word* from memory. A *quadword* refers to 8 bytes and 16 bytes of data are called a *paragraph*. This is the terminology used to address and fetch data from memory in the *Intel Architecture (IA)* family of processors. In general, if the size of the bus or register used to access memory is  $n$  bits, then the number of bytes directly addressable corresponds to an *address space* given as  $2^n$  locations. For a memory system where the minimum number of bits fetched per data unit is  $k$ , then the capacity of memory in terms of the total number of bits is  $2^n \times k$ . Therefore, as shown in Fig. 1.1, an  $i$ th entry consists of  $k$  bits representing a unit of data stored in memory at address  $i$ .

Typical units used to measure the capacity of memory are included in table 1.1.

Table 1.1: Prefixes used to measure memory capacity

1 Kilobyte	1 thousand Bytes	$2^{10} \approx 10^3$ bytes
1 Megabyte	1 thousand Kilobytes	$2^{20} \approx 10^6$ bytes
1 Gigabyte	1 thousand Megabytes	$2^{30} \approx 10^9$ bytes
1 Terabyte	1 thousand Gigabytes	$2^{40} \approx 10^{12}$ bytes
1 Petabyte	1 thousand Terabytes	$2^{50} \approx 10^{15}$ bytes
1 Exabyte	1 thousand Petabytes	$2^{60} \approx 10^{18}$ bytes

Current desktop machines network memory and I/O devices via a system of buses. A *bus* is set of parallel wires dedicated to transfer data, control, and address

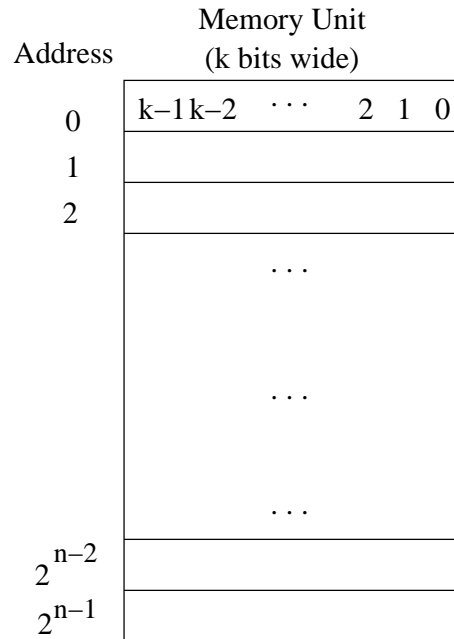


Figure 1.1: Memory organization

bits of information. Each wire in the bus is intended to support the transfer of one single bit of information at a time. Complete systems are integrated using three dedicated buses as shown in the block diagram in Fig. 1.3. Data traveling between the cpu, memory and I/O devices use the *data bus*. Control signals sent from the cpu to mainly I/O devices use the *control bus*. Information used to address specific locations in memory or even to access special I/O devices use the *address bus*.

### 1.3 Basic Concepts

Development of applications require mechanisms of interaction between the user and computing resources. Computer languages are needed to provide such interaction. The most common computer languages are referred to as *high level languages* (HLL). Examples of such languages include Fortran, Pascal, C, C++, etc. HLL programs require a *compilation* process to generate the machine code needed for its execution. Languages that require an *interpreter* are known as *script* languages. Examples of script languages include *perl*, *tcl/tk*, *java script*, etc. HLL are said to be *application-oriented* languages because the emphasis during development is on the application in terms of ease of programming and portability; ideally the user should not be concerned as to where programs are executed.

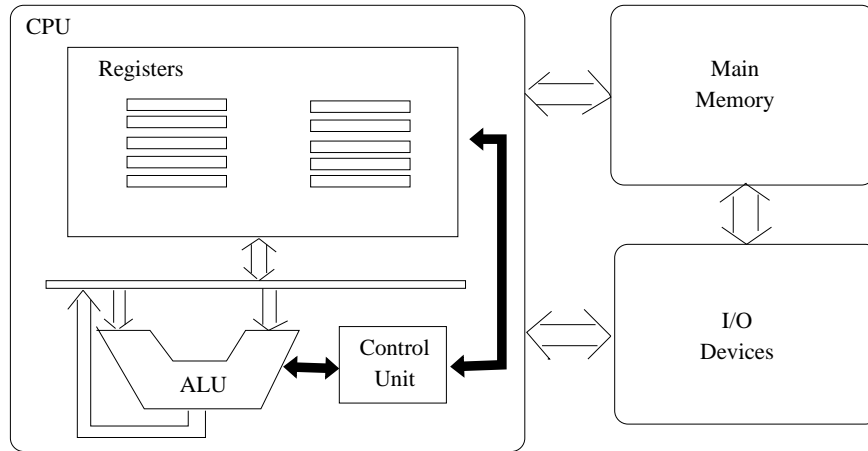


Figure 1.2: A single-processor computer system

A second class of languages are known as *assembly languages* (AL). Applications developed using an AL resort to an *instruction set* which is a list of predefined symbols to be used to specify operations and operands. An instruction set contains the following types of instructions: control, data transfer, arithmetic, logic, and I/O instructions. An AL program is a sequence of instructions which are executed one by one by the *cpu*. Each instruction is fetched, decoded and executed, until a final instruction is fetched that directs the *cpu* to execute a *halt* operation. The process fetching, decoding and executing each instruction is referred to as the *instruction cycle*. Assembly languages are said to be *architecture-oriented* because the programmer needs to be aware of the internal organization of the machine running the application. The programmer must be aware of the number and names of programmable registers, memory capacity and organization, ALU functions, and which type of operations are hardware-supported and which are not. Because ALs are associated with a particular architecture it can be argued that applications can be optimized in terms of the total number of cycles required and the internal resources utilized. The process to produce machine code is referred to as *assembly*. An *assembler* is needed as a tool that takes assembly language instructions and generates the machine code for its execution.

A third class of languages are *machine languages* (ML). ML were used with the introduction of simple computing machines (before the proliferation of modern computing machines) to enter manually instructions and data needed for the execution of simple arithmetic operations. Both HLL and AL generate machine code. The machine code is known as *binary code* because it uses two symbols (1 and 0) to represent information that can be stored in memory. Each assembly instruction is associated with a predetermined code packed into a binary representation formatted

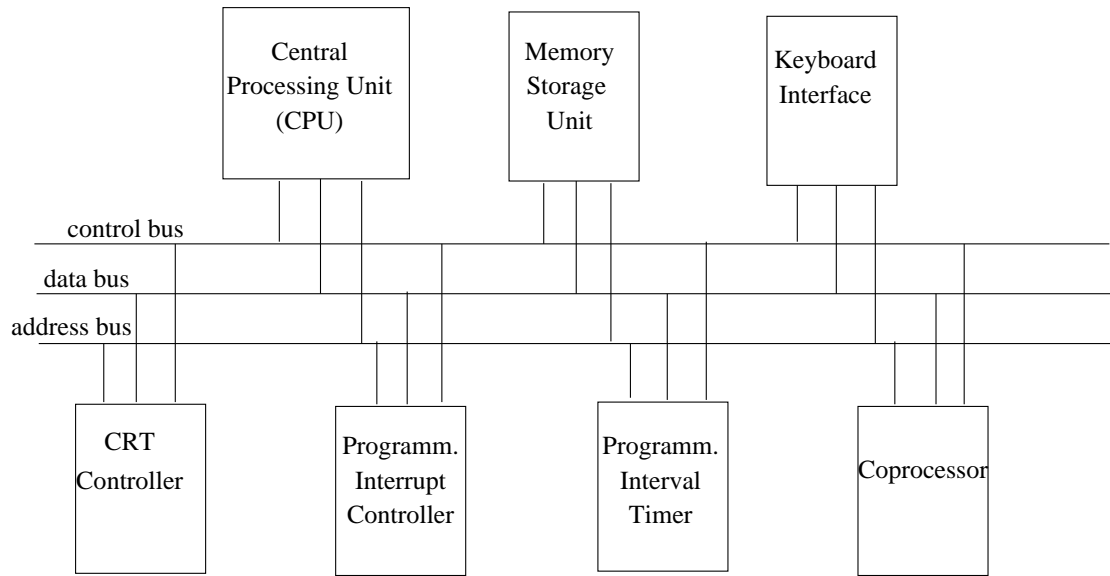


Figure 1.3: Block Diagram of a Single-processor System

into one or more bytes for execution. Formatted instructions are divided into several fields which are decoded by the control unit for internal coordination and synchronization. One of these fields corresponds to the operation code *opcode* which upon fetching the instruction from memory it is used by the *cpu* to select the appropriate ALU operation to execute. Other fields in the instruction format include code to select operands. Operand codes address internal register and/or memory locations where operands are located.

### 1.3.1 Development tools

HLL programs require a *compiler*. A compiler is a utility program that takes as input the HLL source code and generates an intermediate code referred to as *object* code. A set of related object programs generated through the compilation process are linked to generate the binary code of complex programs. In contrast *script* languages require an *interpreter* that executes HLL statements directly. Other languages such as Java use compilation to produce *bytecode* which in turn is used by an interpreter, a Java Virtual Machine (JVM), to execute it.

The assembly process of programs written in an assembly language is similar to the compilation process using a tool called *assembler*. The assembler normally uses two passes to generate an object code. Object programs are linked to form a final binary version of complex programs that may involve modules in HLL and assembly source code. Assembly languages are unique to specific architectures; however, hav-

ing the source code for a particular architecture, it is possible to generate the source code for a different architecture. Such process requires the use of a *cross-assembler* which takes as input the assembly source code for one processor and generates the assembly source code for a different processor.

The use of some of the tools involved in the development of applications is illustrated in Fig. 1.4. The blocks refer to files and the arcs refer to the tools that produced them. From the generation of source code with the use of an *editor*, followed by the use of compilers and/or assemblers to generate object files which are then fed to the *linker* along with access to *library* files to produce executable code. Once an executable file is available a *loader* brings the binary code to main memory for execution. If the output is not as expected then a *debugger* should be available to detect possible bugs and correct them by editing the source code repeating the entire process again.

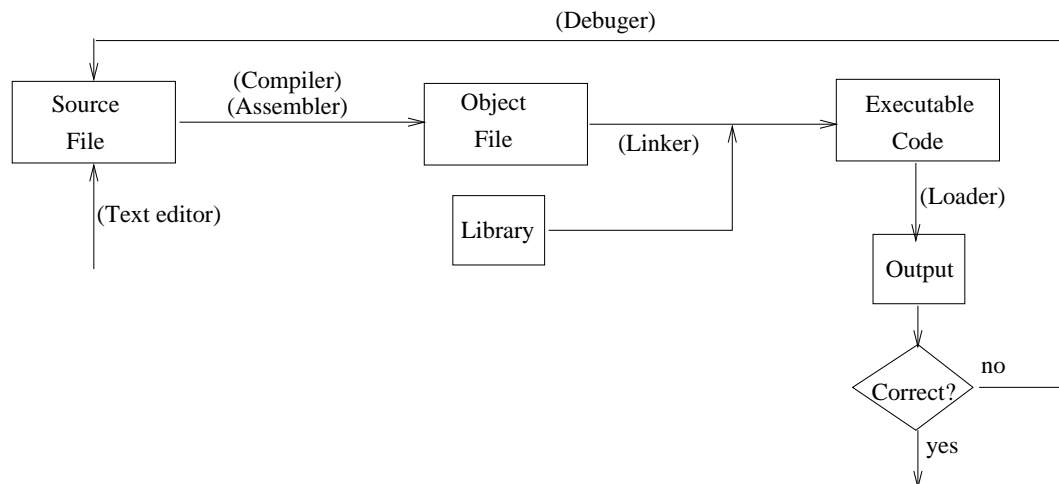


Figure 1.4: Tools used in the development of applications

### 1.3.2 Instruction Cycle

The instruction cycle corresponds to the number of steps the computer follows to execute a single instruction. The CPU first *fetches* an instruction from memory. This instruction is then *decoded*, i.e., each field of the instruction is examined to determine the type of operation the CPU will perform, the location, and retrieval of operands. After decoding the instruction, the CPU proceeds to *execute* it. The CPU repeats these three general steps for each instruction until either a *halt* instruction is executed or the last instruction has been fetched. Fetch, decode, and execute operations are synchronized with the *clock* pulses generated at a fixed frequency given in terms of

*cycles per second*. Clock speeds of Intel's Pentium 4 computers are in the order of 3.0 GHz, or, 3 billion cycles per second. Each instruction takes a given number of clock cycles to execute depending on its complexity and use of resources, i.e., registers, ALU, bus, and memory. The actual speed at which instructions are executed can be derived in terms of technology and architecture design. In terms of technology the increase in the clock frequency is one way to speed up instruction execution. Improvements in the architecture are another way to increase efficiency and speed. One such architectural design improvement is pipelining the instruction cycle. A *pipeline* architecture overlaps the execution of one instruction with the fetching of the next. The use of intermediate memory buffers referred to as *cache* memory is another important architectural design improvement. Caching instructions and data with temporal and spacial locality have the effect of increasing memory bandwidth resulting in data and instructions being fetched faster.

Some instructions are more complex than others and may involve additional access to memory to fetch operands. Other instructions contain immediate operands while others contain codes of registers holding operands or memory addresses where operands are stored. Invariably, every CPU is provided with a program counter (*pc*) register that contains the address, i.e., a *pointer* to the next instruction. Upon fetching a new instruction, the *pc* is incremented to point to the next instruction in the program. The increment corresponds to the size (in number of bytes) of the instruction just fetched. Consider for example the Intel 8086/88 where the program counter is a register identified as the *instruction pointer (ip)*; the following sequence of instructions illustrate the process involved in the execution of assembly programs:

```

mov ax, 0020h    ;a constant is placed in register ax
mov bx, 1000h    ;another constant is placed in register bx
mul bx           ; the contents of ax and bx are multiplied and
                 ;the results are stored in dx:ax

```

This short assembly program simply multiplies two 16-bit constants:  $32 \times 4096 = 2^5 \times 4096$ . The result is a 32-bit value that is stored in two 16-bit registers *dx:ax* which consists of the value 4096 shifted 5 times to the left. The computational process begins by loading (moving) the constant 0020h into the internal register *ax*. Likewise a second constant 1000h is loaded into register *bx*. The third instruction commands the *cpu* to multiply the contents of both registers. The assembly process consists of generating a binary version of the program, i.e., a pattern of 1's and 0's that when loaded into memory will be fetched, decoded and executed sequentially. Fig. 1.5 shows the binary code for each of the assembly instructions.

The *mov* instructions require three bytes; a field of four bits identifies the operation code (opcode) and a second field of four bits identifies the destination

opcode	reg	data	
1 0 1 1	1 0 0 0	0 0 1 0 0 0 0 0	0 0 0 0 0 0 0 0
mov	ax	low byte	high byte

a) MOV ax, 0020

opcode	reg	data	
1 0 1 1	1 0 1 1	0 0 0 0 0 0 0 0	0 0 0 1 0 0 0 0
mov	bx	low byte	high byte

b) MOV bx, 1000

opcode	mod	reg	
1 1 1 1 0 1 1 1	1 1	1 0 0	0 1 1
16-bit mult.	Reg.		bx

c) MUL bx

Figure 1.5: Binary code and instruction formats

operand; an immediate 16-bit field (data field) holds the constant to be copied to the destination register. The *mul* instruction requires two bytes. The first byte corresponds to the *opcode* which identifies a 16-bit multiplication; a 2-bit second field specifies that the multiplier is found in a register (register addressing mode) which is identified by the last 3-bit field. This small program can be loaded into memory using the *debug* tool under DOS as follows:

1. Open a DOS window and type the command *debug*. A “-” symbol is displayed indicating that *debug* is ready to accept user commands. To enter the program type the command *a* to assemble each program line. Terminate each line with a return (cr).

```
C:\>debug
-a
0AF9:0100 mov ax,0020
```



```

0AF9:0103 mov bx,1000
0AF9:0106 mul bx
0AF9:0108

```

2. To generate the binary code of the program, type the command *u* which stands for *unassemble*:

```

-u 100 106
0AF9:0100 B82000      MOV     AX,0020
0AF9:0103 BB0010      MOV     BX,1000
0AF9:0106 F7E3        MUL     BX

```

The range 100 — 106 given with the command *u* corresponds to the locations where our program was placed in memory. If no range is given *debug* will display the contents of the next 32 bytes. Note that the code for each instruction is given in hexadecimal notation and corresponds to the binary code shown in Fig. 1.5.

3. To check the initial contents of the registers use the command *r*:

```

-r
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0AF9  ES=0AF9  SS=0AF9  CS=0AF9  IP=0100  NV UP EI PL NZ NA PO NC
0AF9:0100 B82000      MOV     AX,0020

```

Note that the contents of the *ip* register point to the first instruction *mov ax,0020h*.

4. The command *t* will allow the execution trace of each instruction. The contents of all visible registers are displayed and so it is the status of the program, i.e., the values of the flags in the register flag. A detailed explanation of the Intel 8086/88 architecture is given in the next section.

```

-t

AX=0020  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0AF9  ES=0AF9  SS=0AF9  CS=0AF9  IP=0103  NV UP EI PL NZ NA PO NC
0AF9:0103 BB0010      MOV     BX,1000
-t

AX=0020  BX=1000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0AF9  ES=0AF9  SS=0AF9  CS=0AF9  IP=0106  NV UP EI PL NZ NA PO NC

```

```

0AF9:0106 F7E3          MUL      BX
-t

AX=0000  BX=1000  CX=0000  DX=0002  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0AF9  ES=0AF9  SS=0AF9  CS=0AF9  IP=0108   OV UP EI PL ZR NA PE CY
0AF9:0108 0080FF02      ADD      [BX+SI+02FF],AL      DS:12FF=75
-q

C:\>

```

While the first *mov* instruction is executed the *ip* register is incremented by 3 to point to the next *mov* instruction. Note also that the constant 0020 is placed in the *ax* register. The second *t* command executes the second *mov* instruction and the contents of *ip* are now 0106 and point to the *mul* instruction. The constant value 1000 is now in the *bx* register. As shown in Fig. 1.5 the *mul* instruction requires two bytes and after execution the concatenation of registers *dx* and *ax* show the expected 32-bit result. The command *q* is used to exit *debug*.

## 1.4 Intel Processors

The 8086 is considered the first of the *Intel Architecture* (IA) family of processors followed by a smaller and more cost effective version: the Intel 8088. Notable predecessors of the IA family are the 4004 microprocessor designed in 1969 and the subsequent 8-bit versions, the 8080 and the 8085. One distinctive feature of the IA family of processors is *upward compatibility* by which object code created for early machines (starting in 1978) will still execute on the newest members of the IA family. The 8086 has 16-bit registers and a 16-bit external data bus. A 20-bit addressing provides  $2^{20} \approx 1$ -Megabyte of directly addressable memory space. This is referred to as *real mode* memory organization. The 8088 is identical except for a smaller external data bus of 8 bits. These processors introduced *segmentation*, where 16-bit registers can act as pointers to memory segments of up to 64 KBytes in size. This form of address partitioning is referred to as a *segmented model*. Four segment registers can be used to hold 20-bit base addresses of the currently active memory segments; therefore, up to 256 KBytes ( $4 \times 64\text{K}$ ) can be addressed without switching between segments. For example, the full address in terms of segment and offset values where the instruction *mov ax, 0020h* is found in the previous example is 0AF9 : 0100.

The 80186/80188 family of embedded microprocessors was introduced in 1982. The original 80186/80188 integrated an enhanced 8086/8088 CPU with six commonly used system peripherals. The 80C186/80C188 introduced in 1987 is an en-

hanced 8086/8088 CPU redesigned as a static, stand-alone module known as the 80C186 Modular Core.

The 80286 processor appeared on the market also in 1982. The 80286 began the trend toward high performance architectures including the 80386, the 80486, and the Pentium family. A 16-bit external data bus transfers both 8-bit and 16-bit data between the *cpu*, I/O devices and memory. The address bus of the 80286 is 24 bits which allows up to  $2^{24} \approx 16$  Megabytes of directly addressable space. The Intel 80286 processor introduced the *protected mode* into the IA family as an extension of the real mode operation associated with a 1 Mbyte addressability in earlier microprocessors. *Protected mode* uses the contents of a segment register as a *selector* or pointer into a descriptor table. A *descriptor* provides the 24-bit base address allowing 1) a maximum physical memory size of up to 16 Megabytes, 2) support for virtual memory management on a segment swapping basis, and 3) various protection mechanisms. These mechanisms include segment limit checking, read-only and execute-only segment options, and up to four privilege levels to protect operating system code from application or user programs. Furthermore, hardware task switching and the local descriptor tables allow the operating system to protect application or user programs from each other.

The 80386 is the first of the IA-32 family which introduced 32-bit registers for use both as operands for calculations and for addressing. The external data bus carries up to 32 bits of data. The lower half of each 32-bit register retained the properties of one of the 16-bit registers of the earlier 16-bit generation of microprocessors to provide complete upward compatibility. The 32-bit addressing is supported with an external 32-bit address bus allowing up to  $2^{32} \approx 4$  Gigabytes of directly addressable memory space. Large segments in combination with paging allowed the implementation of a protected *flat model* addressing system. In contrast to the segmented model, a flat protected model allows the programmer to treat the entire memory space as a single segment.

The 80486 expanded the 80386 processors instruction decode and execution units into five pipelined stages, where each stage operates in parallel with the others on up to five instructions in different stages of execution. Each stage can do its work on one instruction in one clock cycle. Consequently when all stages are busy, the 80486 can execute as fast as five instruction per CPU clock cycle. An 8-KByte on-chip level-1 (L1) cache was added to the Intel 486 processor to increase the percent of instructions that could execute at the scalar rate of one per clock. Since cache is an intermediate memory buffer between main memory and the cpu, memory access instructions execute faster if operands were in the L1 cache. The 80486 is also provided with an external 32-bit wide bus; hence, the directly addressable memory space is the same as the one provided by the 80386.

The Intel Pentium processor added a second execution pipeline to achieve super-scalar performance (two pipelines, known as *u* and *v*, together can execute two instructions per clock). The on-chip L1 cache has also been doubled, with 8 KBytes devoted to code, and another 8 KBytes devoted to data. The data cache supports an efficient write-back mode, as well as the write-through mode used by the 80486 processor. Branch prediction with an on-chip branch table was added to increase performance in looping constructs. The main registers are still 32 bits, but the internal data paths of 128 and 256 bits increases internal data bandwidth, The external data bus has been increased to 64 bits while the address bus remained at 32 bits.

The Intel Pentium Pro processor introduced a three-way super-scalar architecture, which means that can execute three instructions per CPU clock. It does this by incorporating even more parallelism than the Pentium processor. The Pentium Pro processor provides Dynamic Execution (micro-data flow analysis, out-of-order execution, superior branch prediction, and speculative execution) in a super-scalar implementation. Three instruction decode units work in parallel to decode instructions into smaller operations called *micro-ops*. Micro-ops are queued into an instruction pool, and if they are free from interdependencies, can be executed out of order by the five parallel execution units (two integer, two FPU and one memory interface unit). The Retirement Unit retires completed micro-ops in their original program order. The power of the Pentium Pro processor is further enhanced by its caches: it has the same two on-chip 8-KByte L1 caches as does the Pentium processor, and also has a 256-KByte L2 cache that is in the same package as the CPU, using a dedicated 64-bit *backside* full clock speed bus. The L2 cache supports up to 4 concurrent accesses. The Pentium Pro processor also has an expanded 36-bit address bus, giving a maximum physical address space of 64 GBytes. The Intel Pentium Pro is the first member of the *P6 family* of microprocessors.

The Pentium II processor added MMX instructions to the Pentium Pro processor architecture. The Pentium II processor expanded the L1 data cache and L1 instruction cache to 16 KBytes each. The Pentium II processor has L2 cache sizes of 256 KBytes, 512 KBytes and 1 MByte or 2 MByte. A half-clock speed backside bus connects the L2 cache to the processor. The Pentium II Xeon combined characteristics of previous generations of the IA architecture to include 4-way, 8-way (and up) scalability and 2 Mbyte 2L cache running on a full-clock speed backside bus. The Intel Celeron focused the IA-32 architecture on the desktop and value PC market. It has an integrated 128 Kbyte L2 cache.

The Pentium III processor introduced the Streaming SIMD Extensions (SSE). SSE expands MMX technology by providing 128-bit registers and the ability to perform SIMD operations on packed single-precision floating point values.

The Pentium 4 processor was introduced at 1.5GHz in November of 2000. It features the Intel NetBurst micro-architecture at significantly higher clock rates. The Pentium 4 processor enables real-time MPEG2 video encoding and near real-time MPEG4 encoding, allowing efficient video editing and video conferencing. The Pentium 4 works with 144 additional 128-bit Single Instruction Multiple Data (SIMD) instructions called SSE2 (Streaming SIMD Extension 2) that improves performance for multi-media, content creation, scientific, and engineering applications. The NetBurst micro-architecture features a renaming logic to map the set of logical IA-32 registers onto the processor's 128-entry register file.

The Intel Xeon processor is also based on the Intel Netburst micro-architecture. This family of IA-32 processors is designed for use in server systems and high-performance workstations. The intel Xeon has the same advance features of the Pentium 4 processor.

The Intel Pentium M is a low power mobile high-performance processor. It features a on die, a primary 32 Kbyte cache and 32 Kbyte write-back data cache, and a 1 Mbyte second level cache. To reduce the number of mispredictions, the processor features provide advance branch prediction. The Data Pre-fetch Logic fetches data to the second-level cache before a cache request to the first-level data cache occurs.

The latest Intel processors includes the Itanium family. The Itanium is a 64-bit architecture with explicit parallelism at the instruction level. This feature is referred to as *Explicitly Parallel Instruction Computing* (EPIC). The Itanium is a joint effort between Hewlett Packard and Intel; it features three levels of cache, level 1 (L1) with 32KB, level 2 (L2) with 96KB, and level 3 (L3) with 4MB. The Intel Itanium 2 with a 1.7 Ghz. clock, also features three cache levels with 16K, 256K, and 9MB, respectively. The Itanium family provides a software layer referred to as the IA32 Execution Layer to support a dynamic translation of IA32-based applications. The Itanium architecture defines 128 general purpose 64-bit registers, 128 floating-point 82-bit registers, 64 predict 1-bit registers to control conditional execution and conditional branches, up to 128 special purpose 64-bit registers (application registers), and a 64-bit instruction pointer register.

Table 1.2 compares some characteristics of the most relevant and successful Intel processors.

## 1.5 Software Model

The *software model* refers to the *visible* part of the system architecture that is available to the programmer to develop applications. Therefore, a software model of

Table 1.2: Key features of selected Intel processors

Processor	Year Intr.	Clock Freq.	Trans./ Die	Register Sizes	Data Bus	Address Space
8086	June 1978	8Mhz	29K	16	16	1 MB (20)
80286	Feb. 1982	12.5 Mhz	134K	16	16	16 MB (24)
80386 DX	Oct. 1985	16 Mhz	275K	32	32	4 GB (32)
80486 DX	April 1989	25 Mhz	1.2M	32 80 FPU	32	4 GB (32)
Pentium	Mar. 1993	60 Mhz	3.1M	32 80 FPU	64	4 GB (32)
Pentium 4	Nov. 2000	1.5Ghz	42M	32 80 FPU 64 MMX 128 XMM	64	64G (36)
Pentium 4	Feb. 2004	3.40Ghz	178M	32 80 FPU 64 MMX 128 XMM	64	64G (36)
Pentium M	Mar. 2003	1.6Ghz	77M	32 80 FPU 64 MMX 128 XMM	64	64G (36)
Intel Itanium	May 2001	800 MHz	25M		64	50
Intel Itanium 2	July 2002	1.7 Ghz	220M		64	50

a given system consists of information regarding the number and names of logical registers that can be used, the size and organization of memory directly addressable, size of internal and external address and data buses, and addressing modes supported.

Consider the architecture of the Intel 8086/88 processor as shown in Fig. 1.6. The CPU features two separate processing units: an Execution Unit (EU) and a Bus Interface Unit (BIU). The BIU and the EU interface via an instruction pre-fetch queue. The EU executes instructions; the BIU fetches instructions, reads operands and writes results. The two units can operate independently of one another and are able, under most circumstances, to overlap instruction fetches and execution and exemplifies an early application of pipelining. Whenever the EU requires another opcode byte, it takes the byte out of the pre-fetch queue. The 16-bit Arithmetic Logic Unit (ALU) performs 8-bit or 16-bit arithmetic and logical operations. It provides for data movement between registers, memory and I/O space. The architecture features 14 basic registers grouped as general registers, segment registers, pointer registers and status and control registers. The four 16-bit *general-purpose*

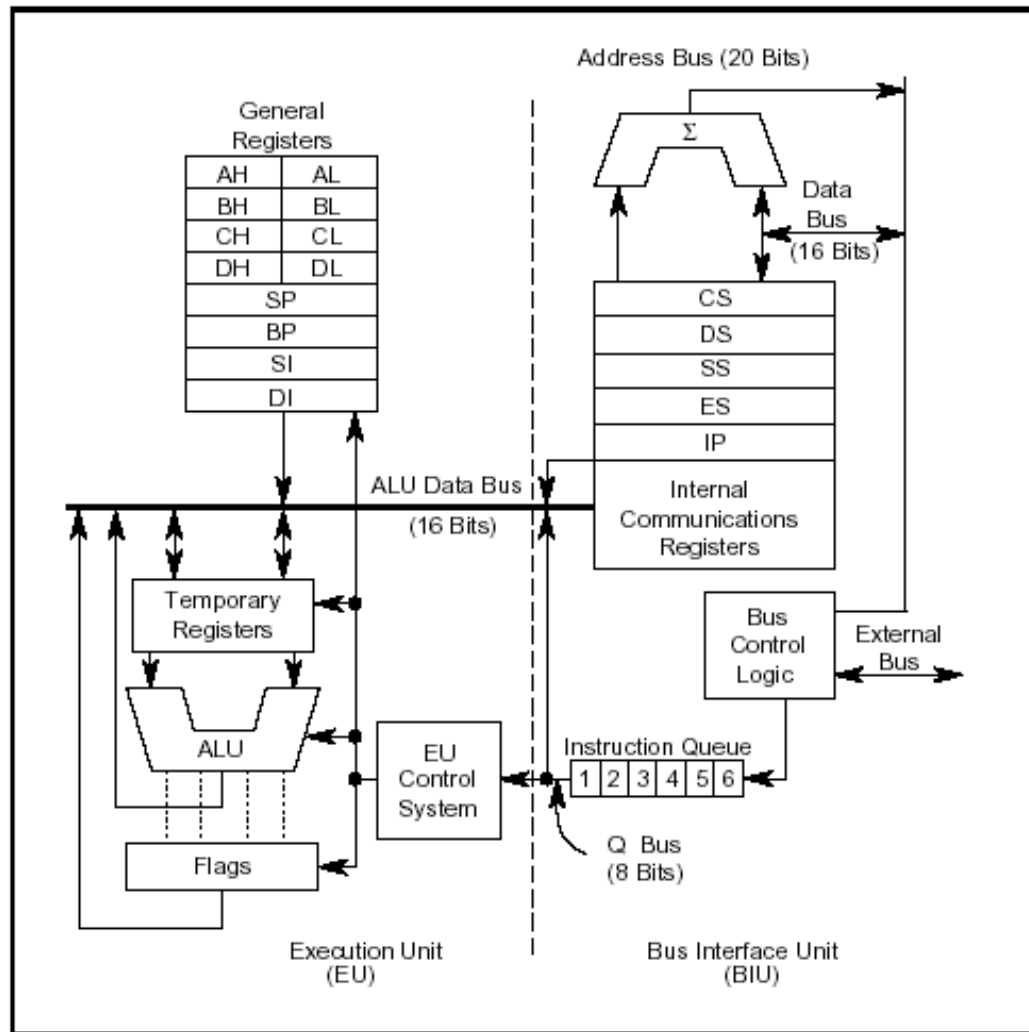


Figure 1.6: CPU block diagram of the 8086

registers (AX, BX, CX and DX) can be used as operands for most arithmetic operations as either 8- or 16-bit units. The four 16-bit *pointer registers* (SI, DI, BP and SP) can be used in arithmetic operations and in accessing memory-based variables. Four 16-bit *segment registers* (CS, DS, SS and ES) allow simple memory partitioning to aid modular programming. The *status and control* registers consist of the Instruction Pointer (IP), and the Processor Status Word (PSW) register, which contains flag bits that control the flow of the program during its execution. The set of visible registers, the size of internal and external buses, and the memory capacity and organization are relevant information that must be available to the assembler programmer, and therefore, it constitutes the *software model* of the Intel 8086/88 machine.



The 1 Megabyte of memory accessed via a 20-bit address bus in the I8086/88 is organized as follows:

00000 — 00400	Interrupt vector table array of addresses used by the cpu when programs are interrupted
00400 — 9FFFF	DOS data area Software Bios – routines to manage the keyboard, console, printer and time-of-day clock (IO.SYS file) DOS kernel (MSDOS.SYS file) Device drivers (CONFIG.SYS file) COMMAND.COM – interprets commands, loads and executes programs RAM for applications
A0000	EGA/VGA graphics buffer
B0000	MDA text buffer (monochrome display adaptor)
B8000	CGA/EGA/VGA text buffer
C0000	reserved
F0000	ROM Bios – diagnostics, configuration software and low-level I/O used by DOS

Note that the range from 00000h to BFFFFh is assigned for RAM, and from C0000h to FFFFFh for ROM storage. The rest is used for video display, disk controller, BIOS, etc.

While the software model of the Intel 8086/88 processor corresponds closely to the actual hardware implementation, that is not the case for current complex processor designs. The description of the NetBurst microarchitecture of the Pentium 4 as shown in Fig. 1.7 illustrates some of the current architecture issues implemented in modern processors. Some of these features in the Pentium 4 include two levels of instruction cache (L1) and (L2). The Trace Cache is the primary level (L1) from which most instructions in a program are fetched and executed. Only when there is a L1 miss does the NetBurst microarchitecture fetch and decode instructions from L2 cache. The L2 cache stores both instructions and data that cannot fit in the Execution Trace Cache and the L1 data cache. In a sharp contrast with the Instruction Queue in the 8086/88 architecture, the NetBurst provides the Instruction Table Lookahead Buffer (ITLB) to translate instruction logical addresses given to it into physical addresses needed to access the L2 cache.

Branch prediction allows the processor to begin fetching and executing instructions long before the previous branch outcomes are certain. The front-end BTB at level 2 and the trace-cache BTB at level 1 provide such functionality. A salient feature of modern processors is the register file shown as the *integer register file* because



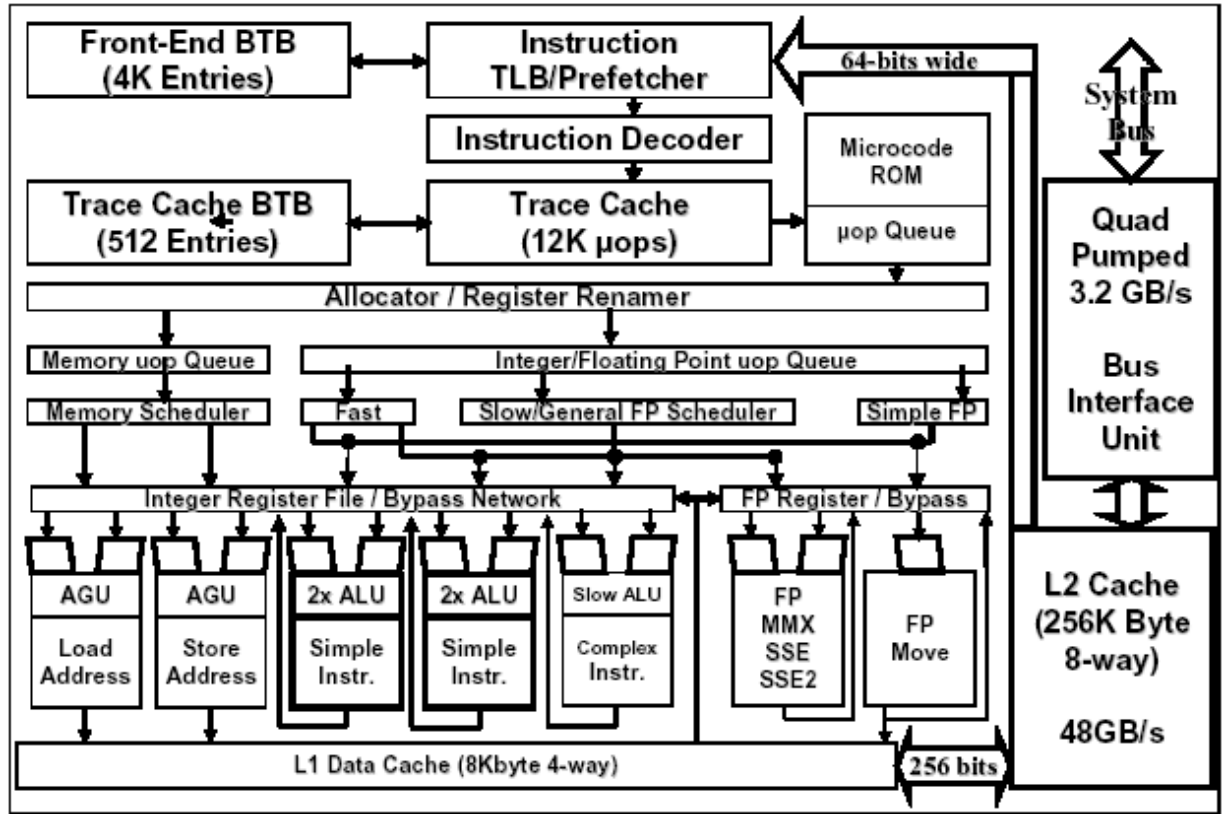


Figure 1.7: NetBurst microarchitecture of the Pentium 4

the same chip also includes the floating point unit. The physical register file consists of 128 entries; a register renaming logic maps the logical IA-32 registers given by the software model onto the internal register file. The purpose of a systematic mapping is to eliminate possible conflicts with the existence of several unique instances of registers such as *eax* in the pipeline at one time.

The software model of the IA 32-bit family provide 16 logical registers for use in general system and application programming. As in the case of the Intel 8086/88, these registers can be grouped as follows:

1. *General-purpose data registers*: the primary general purpose registers are *eax*, *ebx*, *ecx*, and *edx*, available to store data and as operands for the ALU operations.
2. *Pointer registers*: Two of these registers *edi*, *esi* are mainly used as index registers. Register *edi* is normally used to point to a *destination* memory buffer, and *esi* is normally used to point to a memory buffer used a source

of data. The pair *ebp*, *esp* are also used as pointers to memory but with the intended purpose of accessing the stack.

3. *Segment registers*, there are six registers (*cs*, *ds*, *es*, *ss*, *fs*, *gs*) designed to hold up to six segment selectors that are used in the generation of physical addresses in protected mode, and
4. *Status and control registers*, which are registers used to record and report any modification of the state of the processor and of the program in execution.

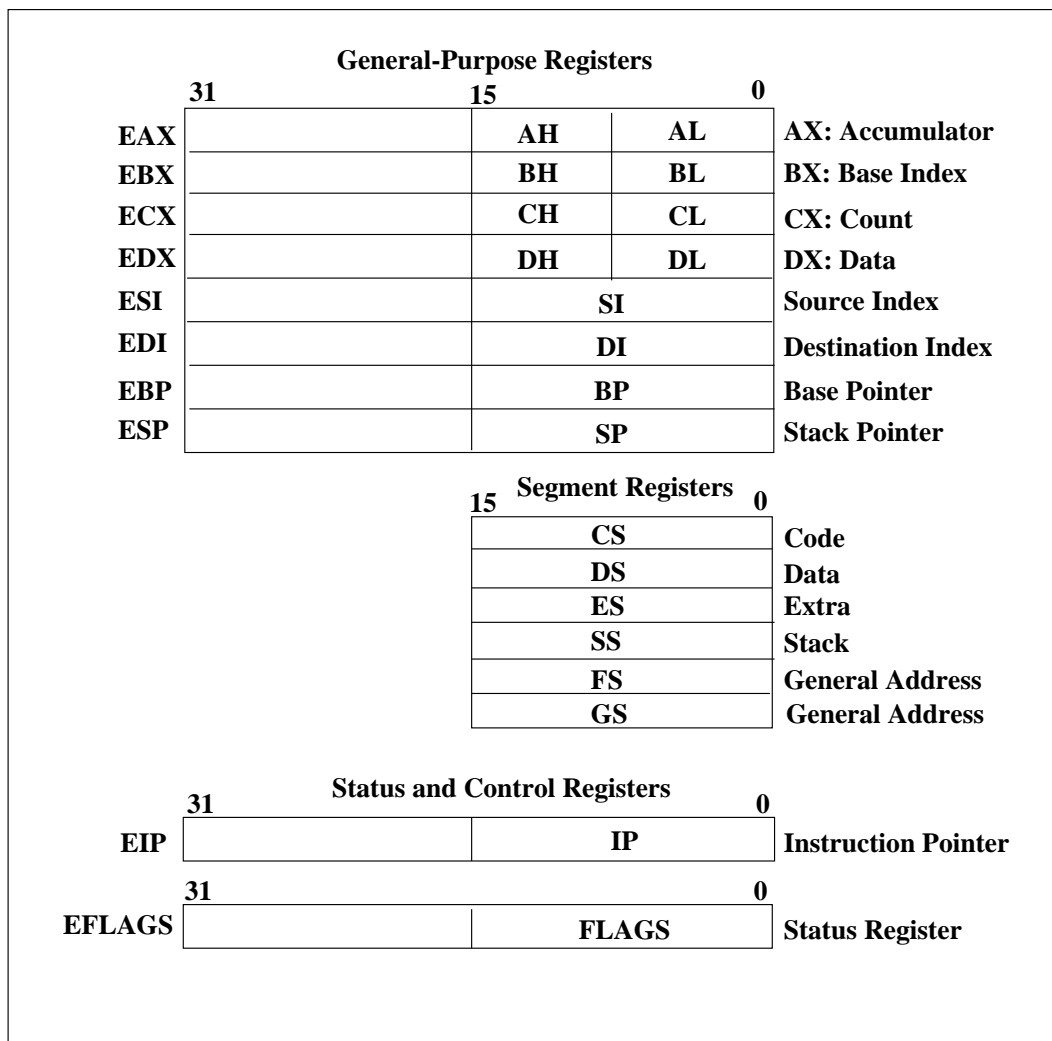


Figure 1.8: Visible set of registers for the IA family

Some registers can be used for specific purposes which is shown in Fig. 1.8 A short summary of the special purpose of each register is given as follows:

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).

EIP — Pointer to the next instruction

EFLAGS — This is a 32-bit register that consists of a collection of one-bit flags that control and describe the general execution state of a program in execution. For reference the *eflags* register is shown in Fig. 1.9.

The lower 16 bits of the general-purpose registers map directly to the 16-bit and 8-bit register set found in the 8086 and 80286 processors. Fig. 1.8 shows the names and a brief description of the specific purpose of each register.

## 1.6 Exercises

1. Give a short definition of the following:
  - (a) BIU
  - (b) Assembler
  - (c) Compiler
  - (d) Cross assembler
  - (e) Machine Language
  - (f) Loader
  - (g) Instruction cycle
2. Explain the function of the following:

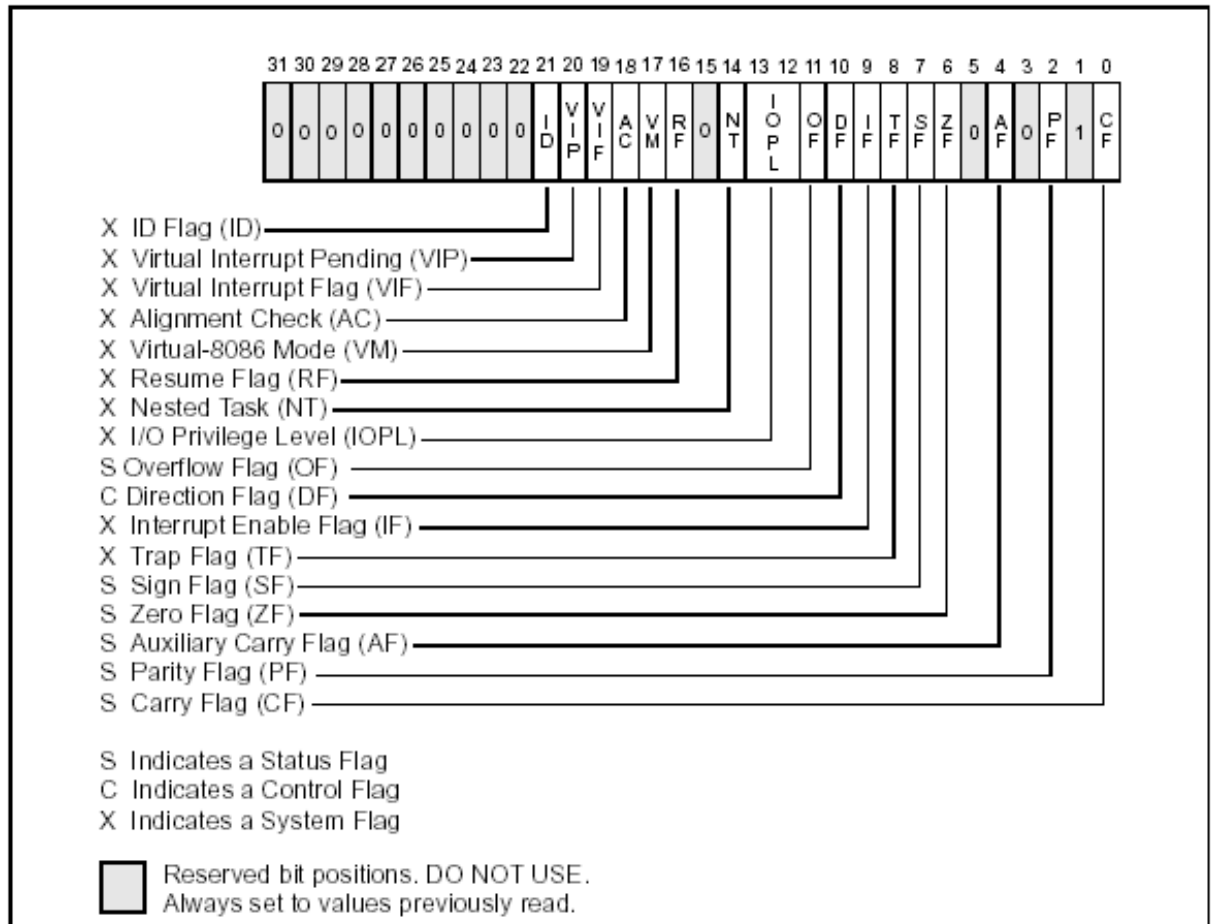


Figure 1.9: Eflags register

- (a) Program counter
  - (b) Instruction pointer
3. How many bytes are in the following data types?
- (a) quadword
  - (b) doubleword
  - (c) word
  - (d) byte
4. For the number  $n$  of bits specified determine the maximum number of memory locations (in Megabytes) that can be addressed directly:
- (a)  $n = 20$

(b)  $n = 24$

(c)  $n = 32$

(d)  $n = 42$

5. What is the purpose of the term *software model*?
6. Name and describe the functions of the following set of registers for the Pentium software model.
  - (a) General purpose registers
  - (b) Segment registers
  - (c) Pointer registers
  - (d) Status and control registers