Chapter 1

Single-processor Computer Systems

1.1 Introduction

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. 1.1 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 coordinates fetching instructions, selection of *alu* operations, selection of source and destination registers, selection of the appropriate *alu* operation, and if necessary, the storing of results in memory.

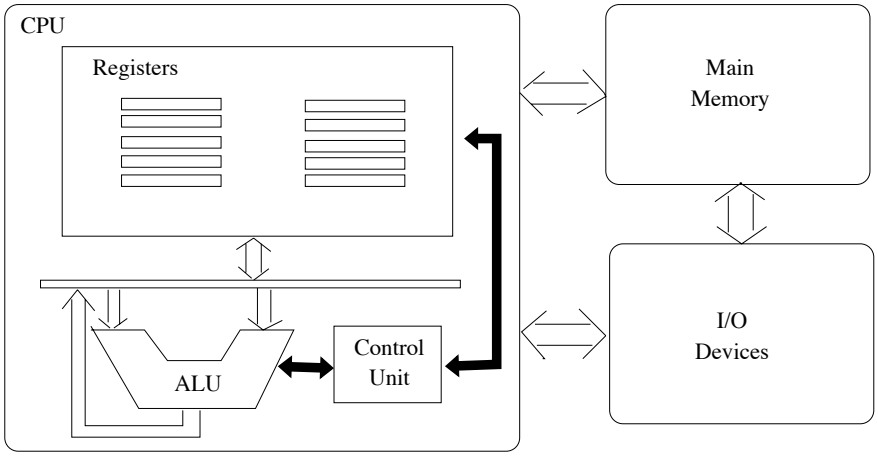


Figure 1.1: A single-processor computer system

Consider for example the execution of the operation $x = x + y$ where a typical assembly instruction such as *add ax, bx* engages the internal bus, the *alu*, and the registers *ax* and *bx* as illustrated in Fig. 1.2. Note that the values x and y have been previously loaded from memory into *ax* and *bx*. After the execution of the *add* instruction *ax* holds the result which in turn can be used by the next instruction to store it in memory or to be used by the *cpu* for subsequent operations.

Internal operations can be described using *register transfer notation* (RTN) to provide information in regard to which *alu* operations are selected and what registers are involved. The selection of operations and operands is carried out by control signals issued by the control unit. These control signals are synchronize to coordinate the transfer of data through a selected datapath as operations are being performed. Detailed computer design makes use of RTN-based specification for the efficient design of the control unit. For example the operation $x = x + y$ could be translated into one or more register transfer operations that include:

1. The transfer of data from the source registers to the internal bus under one clock cycle. The control signals associated with these transfers are labeled in

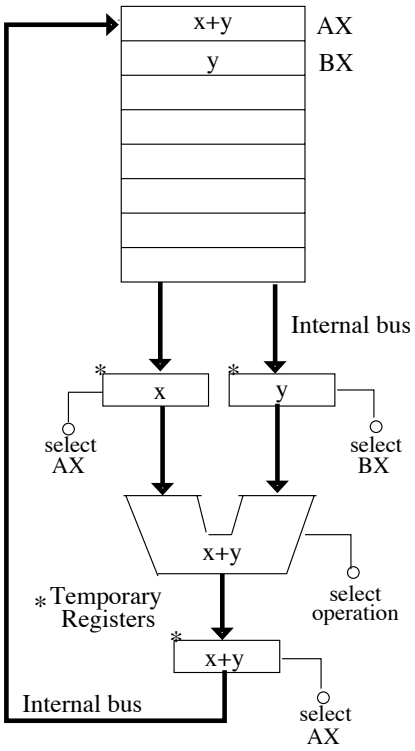


Figure 1.2: Execution of $x = x + y$.

Fig. 1.2 as "select AX" and "select BX" to select registers AX and BX as the sources of data,

2. A second transfer from the internal registers to the *alu* to the internal bus. A control signal label "select operation" can be used to trigger data into the ALU and to select the operation to be performed.

3. A third transfer from the internal bus to the final destination registers. This transfer is performed by a control signal labeled "select AX".

An RTN-based sequence can also be used to describe the internal resources needed for the hardware implementation of assembly language instructions without regard to the coordination and timing signals involved in the actual hardware implementation. Thus, the equation $x = x + y$, is implemented by the execution of *add ax, bx*, which can be described as:

$$ax \leftarrow ax + bx$$

to indicate which registers provide data, which *alu* operation is performed, and which register will hold the result.

Throughout this book we will use RTN simply to describe non-detailed register operations with the intention of familiarizing the reader with a process that very often involves:

1. A mathematical description of an operation,
2. The software implementation of such a mathematical description using assembly language, and
3. The internal operations at the register level that are involved in the hardware implementation of any assembly language instruction.

We will not be concerned with the synchronization of the timed internal activities performed by the control unit.

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 in locations 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* with 2^n bytes. For a byte-oriented memory organization each byte in the address space corresponds to the minimum addressable item. For a memory system where the minimum number of bits fetched per data unit is k, then the capacity of the memory in terms of the total number of bits is $2^n \times k$. Therefore, as shown in Fig. 1.3, an ith entry consists of k bits representing a unit of data stored in memory at the ith address.

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

Current desktop machines do network memory and I/O devices via a system of buses. A *bus* is a set of parallel wires dedicated to transfering data, control, and address 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. 1.4. Data traveling

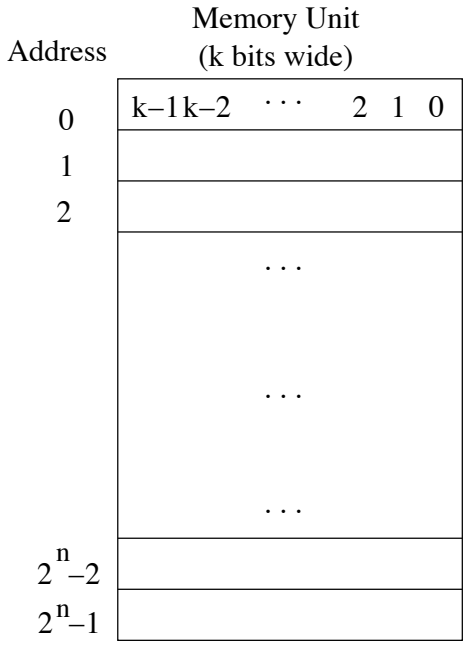


Figure 1.3: Memory organization

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*. The *address bus* is used to transferring address bits needed to access specific locations in memory or to access special I/O devices.

1.3 Basic Concepts

The development of applications requires 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.

Another class of languages is 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

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

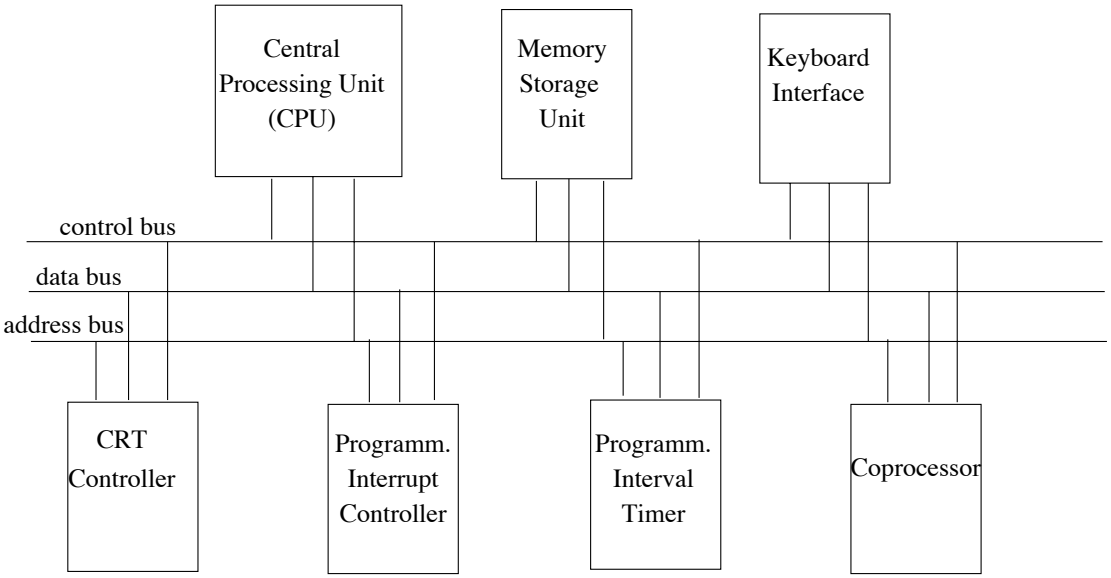


Figure 1.4: Block Diagram of a Single-processor System

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 of 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, the instruction set and whether or not some operations are hardware-supported. 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 of generating machine code is referred to as *assem-*

bly. An *assembler* is needed as a tool that takes assembly language instructions and generates the machine code for its execution.

Yet another class of languages include the *machine languages* (ML). ML were used with the introduction of simple computing machines (before the proliferation of modern computing machines) to manually enter instructions and data needed for the execution of basic 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 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 specifies the operation code *opcode* used by the control unit 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. As the name implies, JVM is a software-based architecture that can be enabled on any hardware platform making the bytecode of a program easily portable.

The assembly process of programs written in an assembly language is similar to the compilation process except that the tool uses is called an *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, having the source code for a particular architecture, it is possible to generate the source code for a different architecture. Such a 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. 1.5. 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.

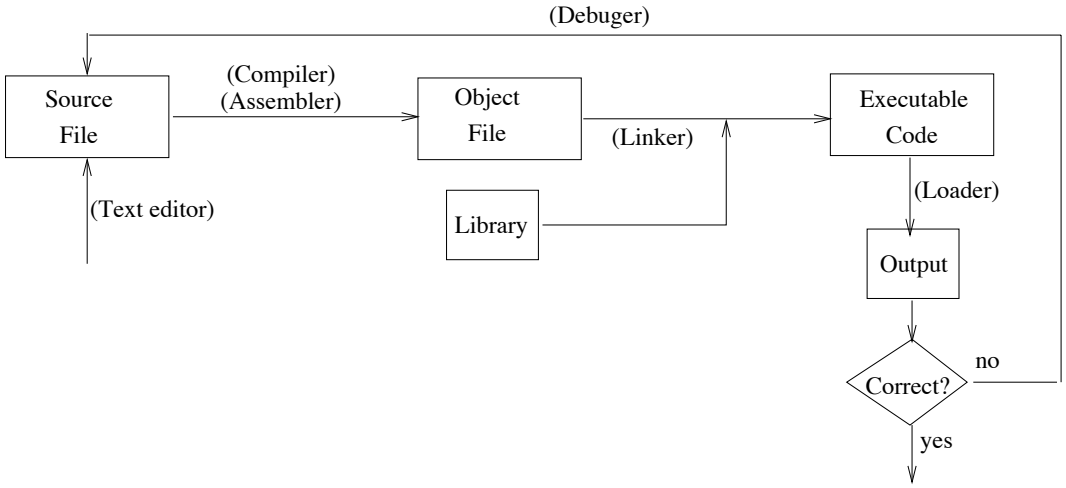


Figure 1.5: Tools used in the development of applications

1.3.2 Instruction Cycle

The instruction cycle refers 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 depends on two factors, the technology used, and the 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.

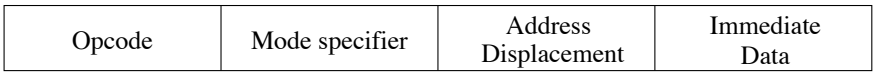
1.3.3 Instruction formats

Some instructions are more complex than others and may involve additional references 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. Fig. 1.6 shows the general instruction format used by Intel processors. The format shows up to four fields but the actual number of fields used depends on the particular instruction format. Instructions that require operands stored in memory need to be fetched during execution. These are called *memory reference instructions* and may require up to four fields where the addressing mode field specifies either a register or an address field in the instruction that contains the *effective address*.

Appendix A details the instruction formats and encoding used for the Intel instruction set. Consider for example a small 16-bit program for the I8086/88 processor where the program counter is a register identified as the *instruction pointer* (*ip*) ; the following sequence of instructions illustrates 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 concatenated 16-bit registers *dx:ax* that will hold a value that consists of 4096 shifted 5 times to the left. The computational process begins by loading (moving) the constant $0020h(= 32)$ into the internal register ax. Likewise a second constant $1000h(= 4096)$ is loaded into register bx. The third instruction commands the *alu* to multiply the contents of

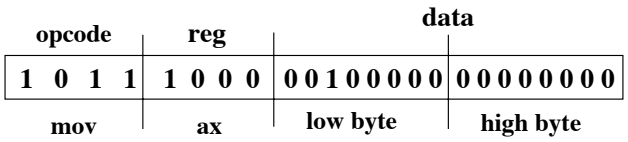


Opcode:	The opcode may require one or two bytes Smaller encoding fields may be defined within the opcode field. These fields define such information as register encoding, conditional test performed, or sign extension of immediate byte.
Mode specifier:	The mode specifier consists of two bytes: *mod r/m:* Almost all instructions that refer to a register and/or memory operand have a register and/or address mode byte that consists of the mod field, the reg field, and the *r/m* field. *sib*: This byte stands for *scale, index, base* and it is used to fully specify the manner in which an effective address is computed.
Address displacement:	If the selected addressing mode specifies a displacement, the displacement value is placed immediately following the *mod r/m* byte or *sib* byte. The displacement can be 8, 16, or 32 bits long.
Immediate data:	If the instruction specifies an immediate operand, the immediate value follows any displacement bytes. An immediate operand, if specified, is always the last field of the instruction and can take 8, 16 or 32 bits.

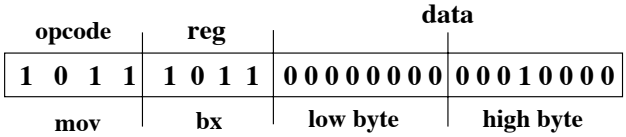
Figure 1.6: Instruction format

both registers. The assembly process generates a binary version of the program, i.e., a pattern of 1's and 0's formatted into instructions that when loaded into memory will be fetched, decoded and executed sequentially. Fig. 1.7 shows the binary code for each of the assembly instructions.

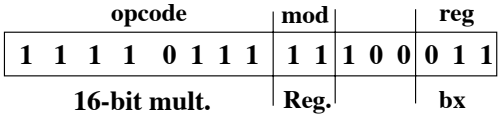
The *mov* instruction require three bytes; a field of four bits identifies the operation code (opcode) and a second field of four bits identifies the destination operand with a format *w, reg*, where $w = 1$ indicates that all 16 bits are used, and *reg* is the code assigned to the register used. If $reg = 000$ then *ax* is used; if $reg = 011$ then *bx* is used; an immediate 16-bit field (data field) holds the constant to be copied to the destination register. This is the *immediate-to-register* short format of the *mov* instruction. The *mul* instruction requires two bytes. The first byte corresponds to the *opcode* which identifies a 16-bit multiplication; a two-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



a) MOV ax, 0020



b) MOV bx, 1000



c) MUL bx

Figure 1.7: Binary code and instruction formats

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. 1.7.

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 is the status of the program, given by 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. 1.7, 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 (4×64K) 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 enhanced 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. 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* which divides the addressable space into more than one segment, a flat protected model allows the programmer to treat the entire memory space as a single segment.

The IA 32-bit family introduced many features that characterized modern processor design. Some of these features include the following:

1. *A large number of internal registers.* The execution of typical instructions require the involvement of more than one register which are used for temporary storage of data within the *cpu*, as illustrated in Fig. 1.2. To boost the execution of complex or pipelined instructions a large number of internal registers are needed. The set of internal files available for execution of instructions is referred to as the *register file* in most modern processors.

2. *Multiple cache units.* A *cache* unit refers to a relatively small set of memory cells within the *cpu* chip that acts as a buffer to store data or instructions which are also stored in main memory but outside the *cpu* chip. If a data item or instruction is requested and resides in cache then a *hit* occurs; otherwise, a *miss* occurs and the item is fetched from main memory. Since accessing main memory is slower than accessing cache then the use of a cache unit results in faster access of instructions or data items. The design of modern processors must address issues involving not only increasing the hit ratio but the number of cache units organized in multiple levels, and the size and purpose of each unit at each level. Since all cached data is also found in main memory, both sets of data must must be maintained consistent when write operations occur. Common techniques to maintain data consistency are *write through* and *write*

back.

3. *Pipelining.* This technique refers to the organization of the *cpu* in several staged units such that when all of them are engaged several instructions are in execution simultaneously. A typical organization pipelines the instruction cycle as shown in Fig. 1.8 with five stages that streamline the execution of up to five instructions when the pipe is full. Note that the instruction cycle is augmented with the fetching of operands for which one unit is created as well as a unit to store results in memory.

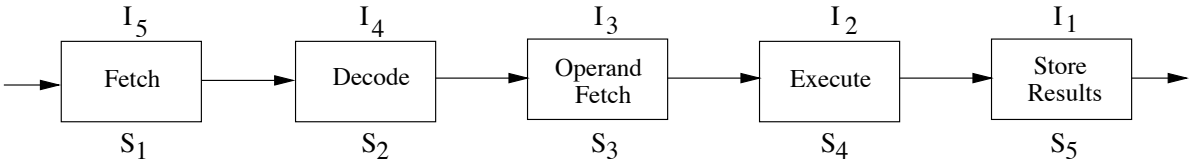


Figure 1.8: A 5-stage pipeline organization

Each stage is labeled $I_i, i = 1, \ldots 5$ to indicate that when a program segment with 5 instructions fills the pipe, instruction I_1 is storing results completing its cycle while I_5 is being fetched and thus, beginning its instruction cycle. Assuming a program has no jump instructions the flow of work through a 5-stage pipeline is described in terms of clock cycles in Fig. 1.9

Stage/cycles	1	2	3	4	5	6	7	8	9
S_1	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	I_9
S_2		I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8
S_3			I_1	I_2	I_3	I_4	I_5	I_6	I_7
S_4				I_1	I_2	I_3	I_4	I_5	I_6
S_5					I_1	I_2	I_3	I_4	I_5

Figure 1.9: Time-space execution diagram of a 5-stage pipeline

Following instruction I_1 through the pipe observe that during clock cycle 1, I_1 is being fetched at stage S_1. During cycle 2, I_1 travels to S_2 to be decoded while I_2 is simultaneously being fetched. During cycle 3, I_1 is at stage S_3 in the process of fetching its operands. Note that during this cycle two subsequent instructions are already in the previous two stages. During cycle 4, I_1 is executed by stage S_4, and finally, in S_5 results are being stored. During clock cycle 5 the pipeline is full. From now on there is one instruction executed per clock cycle. A non-pipelined processor would need 5 cycles to execute one

instruction. Resulting in a pipeline execution time five times faster than the non-pipeline execution time.

4. *Branch prediction.* This feature allows the processor to begin fetching and executing instructions long before the previous branch outcomes are certain. An optimal use of linear pipelines is intrinsic to the way a program in execution is organized. A program not containing jump instructions will increase its throughput in linear proportion to the number of stages in the pipeline. However, a general purpose architecture is expected to execute jump instructions randomly. A delayed execution results each time a jump instruction is executed as all instructions already in the pipe must be flushed out before a new instruction is allowed to enter the pipe. Branch prediction has the purpose of minimizing this delay by pre-fetching the set of instructions in the targeted address. Note that a conditional jump must go through the execution stage to determine whether the condition is true or not. If true, the pre-fetched buffer provides the next set of instructions; otherwise, the instructions already in the pipeline continue their execution through the pipeline.

5. *multiple alu's.* Several processors include multiple arithmetic logic units to provide parallelism at the instruction level. Coupled with pipelined configurations, each *alu* is provided with instructions and operands that have been fetched from memory and exist in the pipeline resulting in the parallel execution of several instructions increasing consequently the pipeline throughput.

The 80486 expanded the 80386'S 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. Consequently when all stages are busy, the 80486 can execute as fast as five instruction per 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. 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 it 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 (MultiMedia eXtension) instructions to the Pentium Pro processor architecture. MMX instructions were intended to replace special multimedia coprocessors and speed up audio and video processing. 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, 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. A later 3.07 Ghz version of the Pentium 4 introduced *hyperthreading* intended to speed up execution by creating two threads of control which divide the work in two parts that can run in parallel.

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 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 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. The I8086 model and the Pentium 4 model are discussed in the next

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

two sections to contrast the need to represent inherent internal resources and make them available to the programmer.

1.5.1 The I8086 model

Consider the architecture of the Intel 8086/88 processor as shown in Fig. 1.10. The CPU features two separate processing units: an Execution Unit (EU) and a Bus Interface Unit (BIU). The BIU and the EU interact via an internal ALU data bus and 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 the fetching of the next instruction with the processing of the instruction currently through the EU. Such overlapping 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 *alu* performs 8-bit or 16-bit arithmetic and logical operations. It provides for data movement between registers,

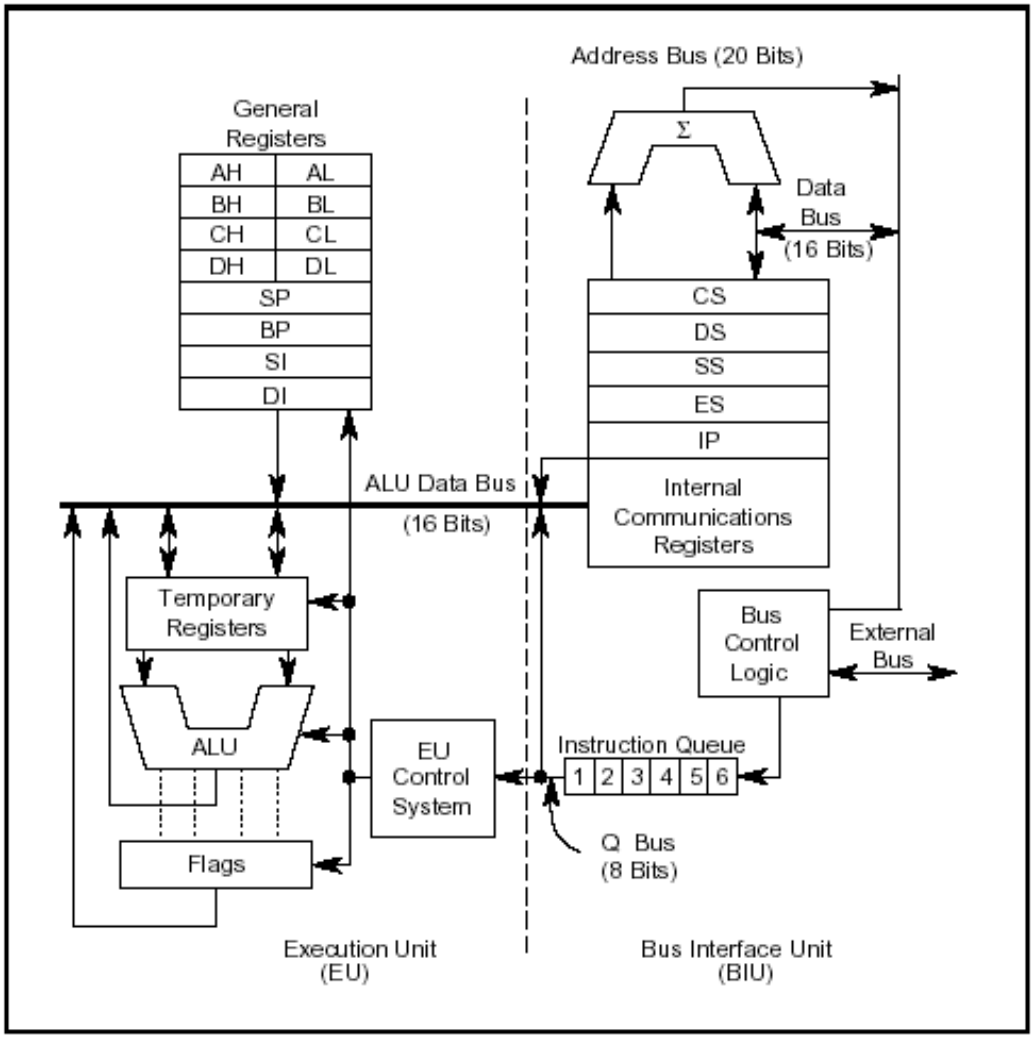


Figure 1.10: CPU block diagram of the 8086

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* 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 in modular programming as programs can be organized into sections or segments that are associated with sections in memory addressed by the corresponding segment register. The *status and control* registers consist of the Instruction Pointer (*ip*), and the Processor Status Word (*PSW*) register, which contains flag bits that regulate the flow of the program during

its execution. The format of the 16-bit flag register as shown in Fig. 1.11 assigns bits to two types of flags, *status* flags and *control* flags which are described as follows:

CF: The *carry flag* is a status flag that holds the carry after an addition operation or the borrow after a subtraction. This is the status of the outgoing most significant bit in the result.

PF: The *parity flag* is a status flag that contains a logic 0 for odd parity and a logic 1 for even parity. An odd parity occurs when the number of 1's in a binary word is odd. An even parity results when the number of 1's is even. If the binary value represented is 0 then an even parity is indicated and $PF = 1$.

AF: This is the *auxiliary carry flag* and it is also a status flag that holds a *half-carry* bit after an addition or a *half-borrow* bit after subtraction. This carry bit is produced out of bit 3 in the *al* register and was primarily used for BCD operations.

ZF: The *zero flag* is also a status flag that indicates if the result of an arithmetic or logic operation is zero. If $ZF = 1$ then the result is zero; otherwise, the result is not zero.

SF: As a status flag the *sign flag* holds the arithmetic sign of the result after an arithmetic or logic operation. If $SF = 1$, the result is negative; otherwise, the result is positive.

TF: The *trap flag* is a control flag that if enabled ($TF = 1$) a running program will be interrupted each time an instruction is executed. This is a trap feature that provides a *single-step* capability for debugging a program during its execution.

IF: The *interrupt flag* controls (control flag) an input pin labeled *INTR*. If $IF = 1$ then *INTR* is enabled, that is, any external interrupt signal will be acknowledged and processed; otherwise, all external interrupts are ignored as *INTR* is disabled.

DF: The *direction flag* is a control flag that determines either to increment or decrement the contents of an index register, controlling the up or down direction of a *pointer* to a section in memory. If $DF = 0$ the index register is automatically incremented; otherwise, it is decremented.

OF: The *overflow flag* is a status flag to indicate the occurrence of an overflow. An overflow occurs ($OF = 1$) if two numbers of the same sign are added and the resulting value is of the opposite sign. The overflow flag is used for signed operations and simply indicates that a result is too big to be represented with the available number of bits.

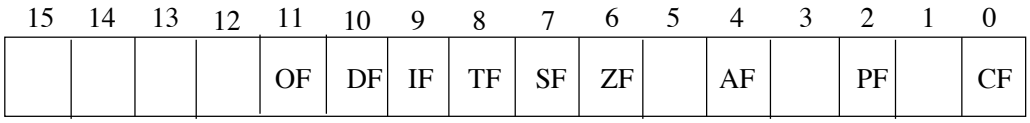


Figure 1.11: I8086/88 Flag register

Table 1.3: Allocation of 1 Megabyte of memory in the I8086/88 processor

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

The 1 Megabyte of memory accessed via a 20-bit address bus in the I8086/88 is organized as shown in Table 1.3. 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.

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.

1.5.2 The Pentium 4 Model

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. 1.12 illustrates some of the current architecture features implemented in modern processors. Some of these features 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, i.e, the instruction is not found in L1, does the NetBurst micro-architecture 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. The front-end BTB at level 2 and the trace-cache BTB at level 1 provide such functionality.

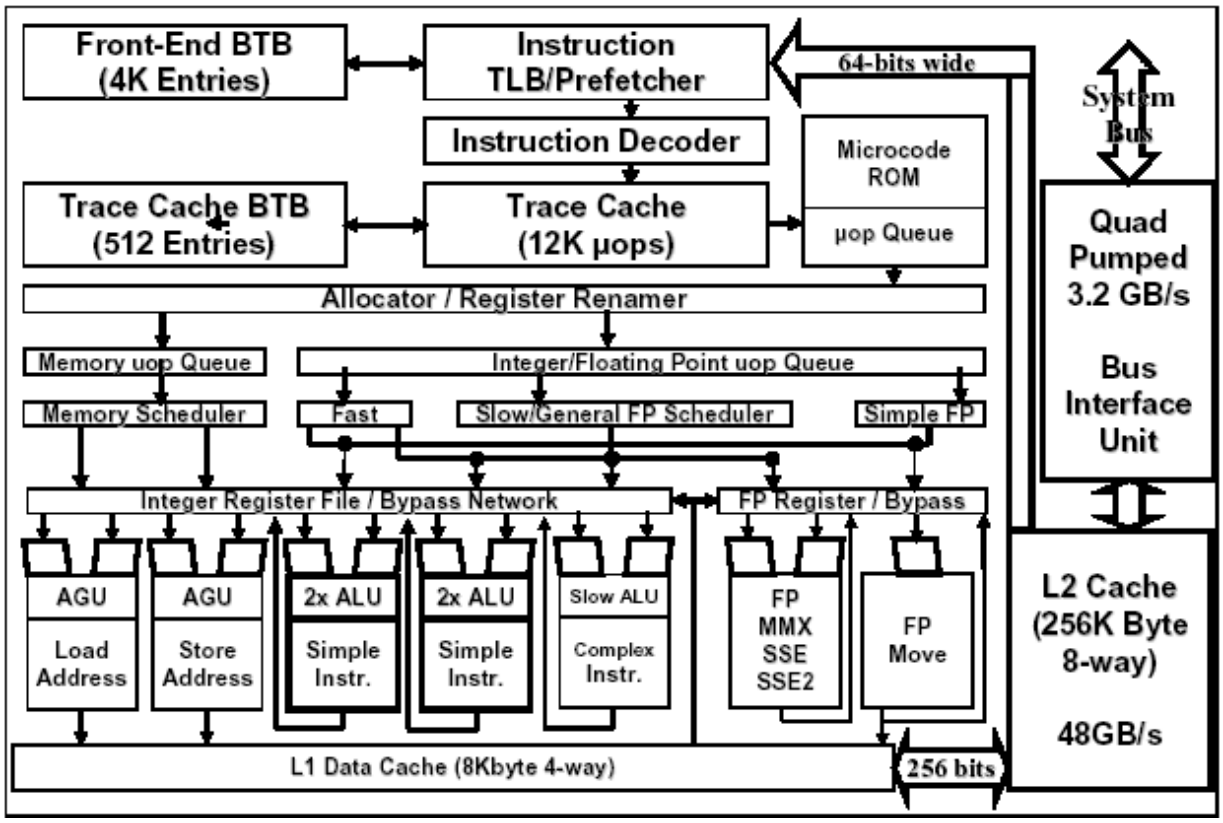


Figure 1.12: NetBurst microarchitecture of the Pentium 4

The register file is shown as the *integer register file* because 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 provides 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. These registers can also hold 8-bit and 16-bit values for 8-bit and 16-bit applications or simply data that fits these sizes in 32-bit applications. However, the main idea is to preserve upward compatibility with 16-bit processors that deal with 8-bit and 16-bit data items. The use of the lower 16-bit or 8-bit part of the register does not affect the remaining contents of the register.

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*) that are 16 bit in size. Segment registers were designed to hold up to six segment selectors that are used in the generation of physical addresses in protected mode.

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. The *eip* register holds the address of the next instruction in the normal program sequence. It is modified to point to a target address if the current instruction is a jump instruction. The *eflags* register is the extended version of the *flags* register but includes additional flags to control the execution of programs in virtual or protected mode.

Fig. 1.13 shows the names and a brief description of the specific purpose of each register. 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. As shown in Fig. 1.13 some registers can be used for specific purposes . A short description of each register is given as follows:

EAX: Historically this register is referred to as the *accumulator* and it is used to hold operands as well as results from the *alu* operations; special purposes of the accumulator include its role in multiplication, division, and extended bit instructions.

The multi-purpose use of the accumulator has been extended in the IA 32-bit family as a pointer to memory as well.

EBX: It is used as a general purpose register and it also holds 8 and 16-bit data. This register is identified as a base pointer because it is also used as a pointer to address memory data through a 16-bit and 32-bit address bus.

ECX: This is also a multi-purpose register that includes addressing memory; it is especially used as a counter for string and loop operations. It is also used as a counter of the number of one-bit shifts that rotate and shift instructions must perform.

EDX: Another multipurpose register but with dedicated tasks that include holding part of the results of multiplication and division instructions. Another dedicated purpose of this register is as I/O pointer as it can be used to hold the port numbers associated with I/O devices.

ESI: This register is referred to as the *source* index register because it addresses sections of memory from which data is read when string instructions are used.
It is also used as a multi-purpose register.

EDI: This is the *destination* index register. It addresses sections of memory into which data is written when string instructions are used. It is also used as a multi-purpose register.

ESP: It is referred to as *the stack pointer* because it is used to address a section of memory structured as a stack.

EBP: Used as a pointer to memory for data transfers. It is also used to access a section of the stack called *stack frame* where parameters passed via a *call* instruction are located.

EIP: This register contains a pointer to the next sequential instruction of a program in execution. The contents may be modified if the current instruction is a jump.

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. The format is shown in Fig. 1.14.

The first 16 bits contain the same flags for the I8086/88. In fact, from the I8086/88 to the Pentium 4, all flag registers are upward compatible. The remaining flags in Fig. 1.14 are briefly described as follows:

IOPL: This is a two-bit I/O privilege level flag as it requires 2 bits to describe up to four privilege levels for I/O devices. If $IOPL = 00$ corresponds to the

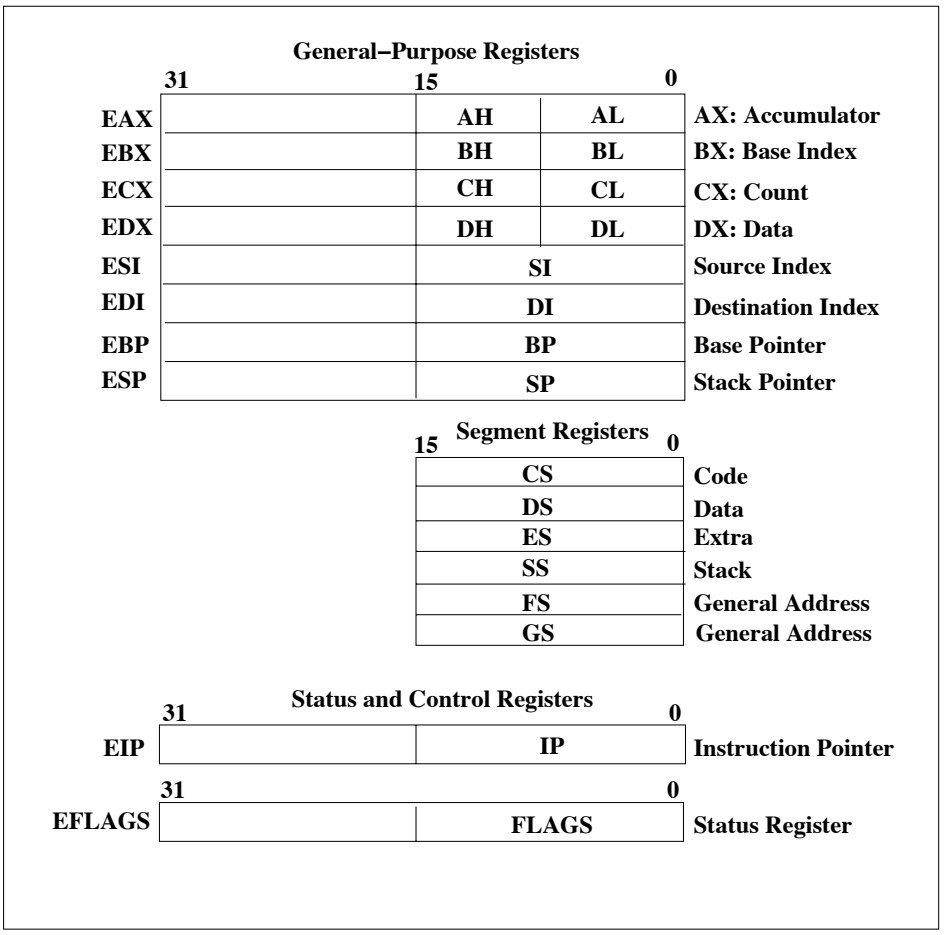


Figure 1.13: Visible set of registers for the IA family

highest or most trusted level, and $IOPL = 11$ corresponds to the lowest or least trusted level.

NT: Indicates that the current task is nested.

RF: The *resumption flag* controls the resumption of execution after the next instruction.

VM A virtual mode system is selected if the *virtual mode flag* is set. A virtual mode allows multiple DOS memory partitions and each partition can be regarded as one more task in a multitasking environment.

AC: This is the *alignment check flag* which is set if the a word or doubleword is addressed on a non-word or non-doubleword boundary.

VIF: The *virtual interrupt flag* is a virtual copy of the interrupt flag (IF).

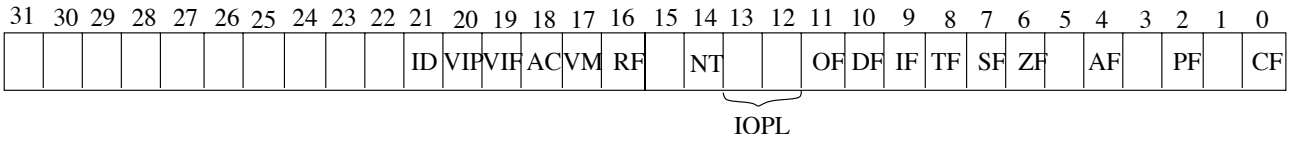


Figure 1.14: Eflags register

VIP: The *virtual interrupt pending flag* is used in a multitasking environment to provide information to the operation system about virtual interrupt flags.

ID: The *identification flag* is set to indicate that the *CPUID* instruction is supported. This instruction provides information such as version number and manufacturer.

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:
   (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. If a manufacturer advertises the following number of bytes as the memory capacity of certain computer systems, indicate what are the number of bits required to address memory and the actual memory capacity in number of bytes the systems have:

   (a) 1 Gigabyte
   (b) 520 Megabytes
   (c) 250 Megabytes
   (d) 120 Megabytes
   (e) 60 Megabytes

6. Single out and explain three features of modern IA processors.

7. What is the purpose of the term *software model*?

8. 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

9. Comment on the main differences between a flat protected model and a segmented model.