addressing. Some architectures, including that of the Pentium, have a collection of two or more specialized sets (such as data and displacement). One advantage of this latter approach is that, for a fixed number of registers, a functional split requires fewer bits to be used in the instruction. For example, with two sets of eight registers, only 3 bits are required to identify a register; the opcode implicitly will determine which set of registers is being referenced.

- **Address range:** For addresses that reference memory, the range of addresses that can be referenced is related to the number of address bits. Because this imposes a severe limitation, direct addressing is rarely used. With displacement addressing, the range is opened up to the length of the address register. Even so, it is still convenient to allow rather large displacements from the register address, which requires a relatively large number of address bits in the instruction.
- **Address granularity:** For addresses that reference memory rather than registers, another factor is the granularity of addressing. In a system with 16- or 32-bit words, an address can reference a word or a byte at the designer's choice. Byte addressing is convenient for character manipulation but requires, for a fixed-size memory, more address bits.

Thus, the designer is faced with a host of factors to consider and balance. How critical the various choices are is not clear. As an example, we cite one study [CRAG79] that compared various instruction format approaches, including the use of a stack, general-purpose registers, an accumulator, and only memory-to-register approaches. Using a consistent set of assumptions, no significant difference in code space or execution time was observed.

Let us briefly look at how two historical machine designs balance these various factors.

PDP-8 One of the simplest instruction designs for a general-purpose computer was for the PDP-8 [BELL78b]. The PDP-8 uses 12-bit instructions and operates on 12-bit words. There is a single general-purpose register, the accumulator.

Despite the limitations of this design, the addressing is quite flexible. Each memory reference consists of 7 bits plus two 1-bit modifiers. The memory is divided into fixed-length pages of $2^7 = 128$ words each. Address calculation is based on references to page 0 or the current page (page containing this instruction) as determined by the page bit. The second modifier bit indicates whether direct or indirect addressing is to be used. These two modes can be used in combination, so that an indirect address is a 12-bit address contained in a word of page 0 or the current page. In addition, 8 dedicated words on page 0 are autoindex "registers." When an indirect reference is made to one of these locations, preindexing occurs.

Figure 11.4 shows the PDP-8 instruction format. There are a 3-bit opcode and three types of instructions. For opcodes 0 through 5, the format is a single-address memory reference instruction including a page bit and an indirect bit. Thus, there are only six basic operations. To enlarge the group of operations, opcode 7 defines a register reference or *microinstruction.* In this format, the remaining bits are used to encode additional operations. In general, each bit defines a specific operation (e.g., clear accumulator), and these bits can be combined in a single instruction. The microinstruction strategy was used as far back as the PDP-1 by DEC and is, in a sense, a forerunner of

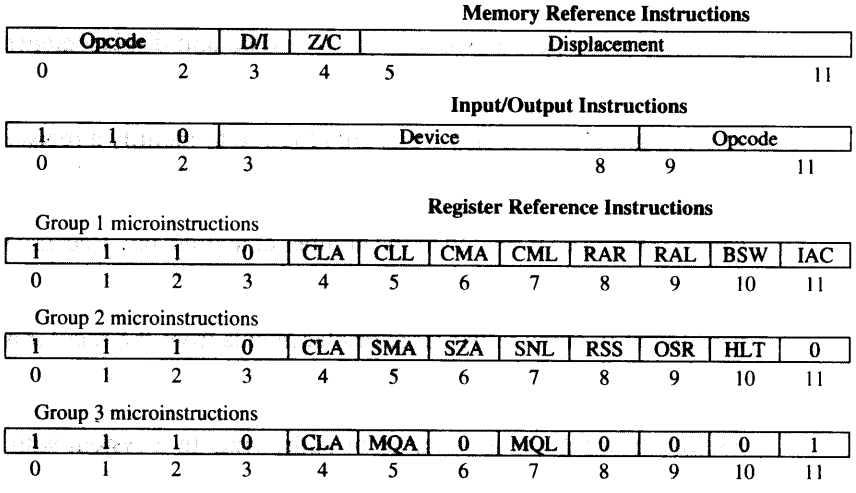


Figure 11.4 PDP-8 Instruction Formats

today's microprogrammed machines, to be discussed in Part Four. Opcode 6 is the I/O operation; 6 bits are used to select one of 64 devices, and 3 bits specify a particular I/O command.

The PDP-8 instruction format is remarkably efficient. It supports indirect addressing, displacement addressing, and indexing. With the use of the opcode extension, it supports a total of approximately 35 instructions. Given the constraints of a 12-bit instruction length, the designers could hardly have done better.

PDP-10 A sharp contrast to the instruction set of the PDP-8 is that of the PDP-10. The PDP-10 was designed to be a large-scale time-shared system, with an emphasis on making the system easy to program, even if additional hardware expense was involved.

Among the design principles that were employed in designing the instruction set were [BELL78c].

- **Orthogonality:** Orthogonality is a principle by which two variables are independent of each other. In the context of an instruction set, the term indicates that other elements of an instruction are independent of (not determined by) the opcode. The PDP-10 designers use the term to describe the fact that an address is always computed in the same way, independent of the opcode. This is in contrast to many machines, where the address mode sometimes depends implicitly on the operator being used.
- **Completeness:** Each arithmetic data type (integer, fixed-point, real) should have a complete and identical set of operations.

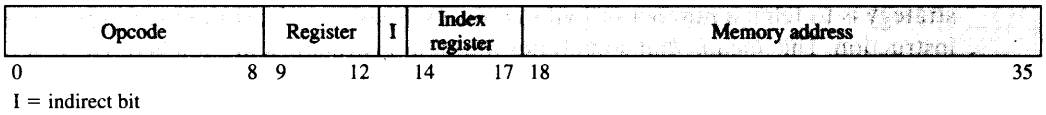


Figure 11.5 PDP-10 Instruction Format

- **Direct addressing:** Base plus displacement addressing, which places a memory organization burden on the programmer, was avoided in favor of direct addressing.

Each of these principles advances the main goal of ease of programming.

The PDP-10 has a 36-bit word length and a 36-bit instruction length. The fixed instruction format is shown in Figure 11.5. The opcode occupies 9 bits, allowing up to 512 operations. In fact, a total of 365 different instructions are defined. Most instructions have two addresses, one of which is one of 16 general-purpose registers. Thus, this operand reference occupies 4 bits. The other operand reference starts with an 18-bit memory address field. This can be used as an immediate operand or a memory address. In the latter usage, both indexing and indirect addressing are allowed. The same general-purpose registers are also used as index registers.

A 36-bit instruction length is true luxury. There is no need to do clever things to get more opcodes; a 9-bit opcode field is more than adequate. Addressing is also straightforward. An 18-bit address field makes direct addressing desirable. For memory sizes greater than 2^{18}, indirection is provided. For the ease of the programmer, indexing is provided for table manipulation and iterative programs. Also, with an 18-bit operand field, immediate addressing becomes attractive.

The PDP-10 instruction set design does accomplish the objectives listed earlier [LUND77]. It eases the task of the programmer or compiler at the expense of an inefficient utilization of space. This was a conscious choice made by the designers and therefore cannot be faulted as poor design.

Variable-Length Instructions

The examples we have looked at so far have used a single fixed instruction length, and we have implicitly discussed trade-offs in that context. But the designer may choose instead to provide a variety of instruction formats of different lengths. This tactic makes it easy to provide a large repertoire of opcodes, with different opcode lengths. Addressing can be more flexible, with various combinations of register and memory references plus addressing modes. With variable-length instructions, these many variations can be provided efficiently and compactly.

The principal price to pay for variable-length instructions is an increase in the complexity of the processor. Falling hardware prices, the use of microprogramming (discussed in Part Four), and a general increase in understanding the principles of processor design have all contributed to making this a small price to pay. However, we will see that RISC and superscalar machines can exploit the use of fixed-length instructions to provide improved performance.

The use of variable-length instructions does not remove the desirability of making all of the instruction lengths integrally related to the word length. Because the processor does not know the length of the next instruction to be fetched, a typical

strategy is to fetch a number of bytes or words equal to at least the longest possible instruction. This means that sometimes multiple instructions are fetched. However, as we shall see in Chapter 12, this is a good strategy to follow in any case.

PDP-11 The PDP-11 was designed to provide a powerful and flexible instruction set within the constraints of a 16-bit minicomputer [BELL70].

The PDP-11 employs a set of eight 16-bit general-purpose registers. Two of these registers have additional significance: One is used as a stack pointer for special-purpose stack operations, and one is used as the program counter, which contains the address of the next instruction.

Figure 11.6 shows the PDP-11 instruction formats. Thirteen different formats are used, encompassing zero-, one-, and two-address instruction types. The opcode can vary from 4 to 16 bits in length. Register references are 6 bits in length. Three bits identify the register, and the remaining 3 bits identify the addressing mode. The PDP-11 is endowed with a rich set of addressing modes. One advantage of linking the addressing mode to the operand rather than the opcode, as is sometimes done, is that any addressing mode can be used with any opcode. As was mentioned, this independence is referred to as *orthogonality.*

PDP-11 instructions are usually one word (16 bits) long. For some instructions, one or two memory addresses are appended, so that 32-bit and 48-bit instructions are part of the repertoire. This provides for further flexibility in addressing.

The PDP-11 instruction set and addressing capability are complex. This increases both hardware cost and programming complexity. The advantage is that more efficient or compact programs can be developed.

VAX Most architectures provide a relatively small number of fixed instruction formats. This can cause two problems for the programmer. First, addressing mode and opcode are not orthogonal. For example, for a given operation, one operand must come from a register and another from memory, or both from registers, and so on. Second, only a limited number of operands can be accommodated: typically up to two or three. Because some operations inherently require more operands, various strategies must be used to achieve the desired result using two or more instructions.

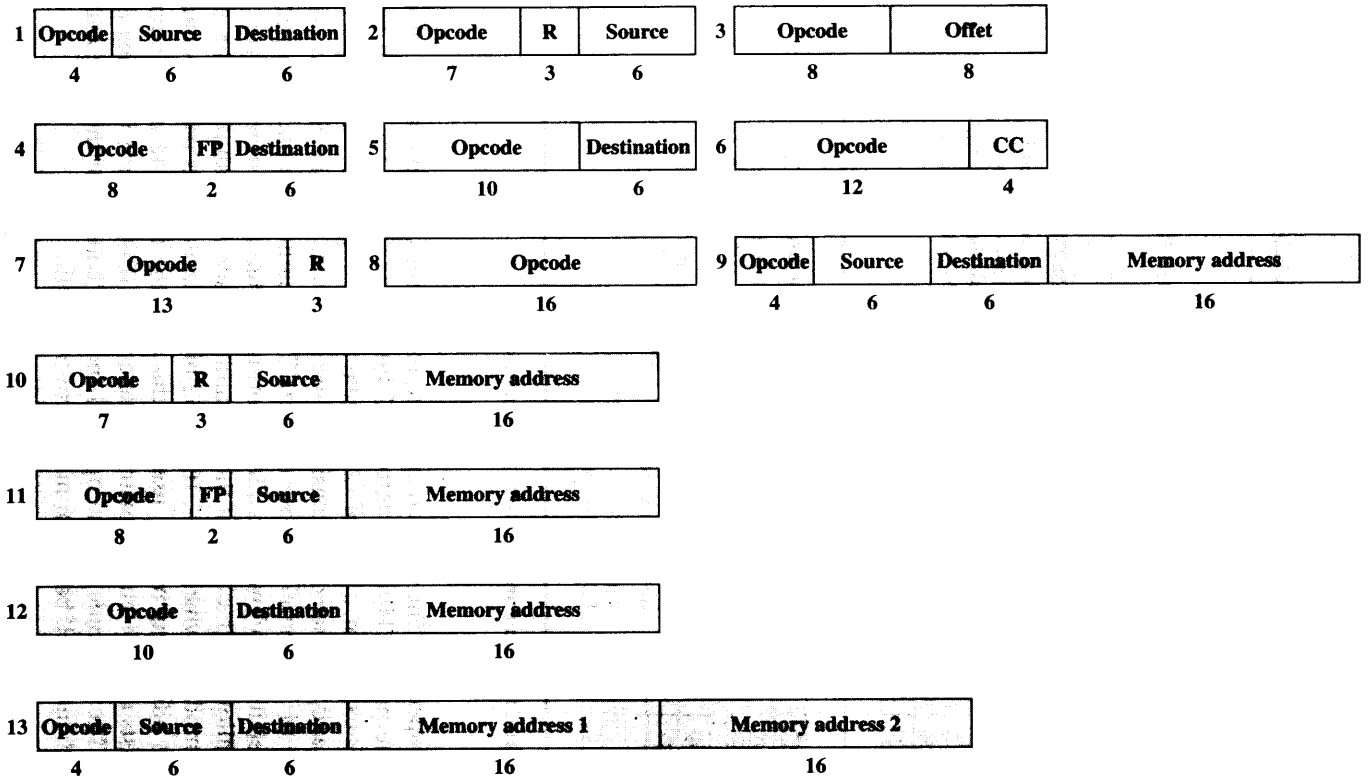
To avoid these problems, two criteria were used in designing the VAX instruction format [STRE78]:

1. All instructions should have the "natural" number of operands.
2. All operands should have the same generality in specification.

The result is a highly variable instruction format. An instruction consists of a 1- or 2-byte opcode followed by from zero to six operand specifiers, depending on the opcode. The minimal instruction length is 1 byte, and instructions up to 37 bytes can be constructed. Figure 11.7 gives a few examples.

The VAX instruction begins with a 1-byte opcode. This suffices to handle most VAX instructions. However, as there are over 300 different instructions, 8 bits are not enough. The hexadecimal codes FD and FF indicate an extended opcode, with the actual opcode being specified in the second byte.

The remainder of the instruction consists of up to six operand specifiers. An operand specifier is, at minimum, a 1-byte format in which the leftmost 4 bits are the



Numbers below fields indicate bit length
Source and destination each contain a 3-bit addressing mode field and a 3-bit register number
FP indicates one of four floating-point registers
R indicates one of the general-purpose registers
CC is the condition code field

Figure 11.6 Instruction Formats for the PDP-11

Hexadecimal Format	Explanation	Assembler Notation and Description
8 bits 0 5	Opcode for RSB	RSB Return from subroutine
D 4 5 9	Opcode for CLRL Register R9	CLRL R9 Clear register R9
B 0 C 4 6 4 0 1 A B 1 9	Opcode for MOVW Word displacement mode, Register R4 356 in hexadecimal Byte displacement mode, Register R11 25 in hexadecimal	MOVW 356(R4), 25(R11) Move a word from address that is 356 plus contents of R4 to address that is 25 plus contents of R11
C 1 0 5 5 0 4 2 D F [ ]	Opcode for ADDL3 Short literal 5 Register mode R0 Index prefix R2 Indirect word relative (displacement from PC) Amount of displacement from PC relative to location A	ADDL3 #5, R0, @A[R2] Add 5 to a 32-bit integer in R0 and store the result in location whose address is sum of A and 4 times the contents of R2

Figure 11.7 Example of VAX Instructions

address mode specifier. The only exception to this rule is the literal mode, which is signaled by the pattern 00 in the leftmost 2 bits, leaving space for a 6-bit literal. Because of this exception, a total of 12 different addressing modes can be specified.

An operand specifier often consists of just one byte, with the rightmost 4 bits specifying one of 16 general-purpose registers. The length of the operand specifier can be extended in one of two ways. First, a constant value of one or more bytes may immediately follow the first byte of the operand specifier. An example of this is the displacement mode, in which an 8-, 16-, or 32-bit displacement is used. Second, an index mode of addressing may be used. In this case, the first byte of the operand specifier consists of the 4-bit addressing mode code of 0100 and a 4-bit index register identifier. The remainder of the operand specifier consists of the base address specifier, which may itself be one or more bytes in length.

The reader may be wondering, as the author did, what kind of instruction requires six operands. Surprisingly, the VAX has a number of such instructions. Consider

ADDP6 OP1, OP2, OP3, OP4, OP5, OP6

This instruction adds two packed decimal numbers. OP1 and OP2 specify the length and starting address of one decimal string; OP3 and OP4 specify a second string. These two strings are added and the result is stored in the decimal string whose length and starting location are specified by OP5 and OP6.

The VAX instruction set provides for a wide variety of operations and addressing modes. This gives a programmer, such as a compiler writer, a very powerful and flexible tool for developing programs. In theory, this should lead to efficient machine-language compilations of high-level language programs and, in general, to effective and efficient use of processor resources. The penalty to be paid for these benefits is the increased complexity of the processor compared with a processor with a simpler instruction set and format.

We return to these matters in Chapter 13, where we examine the case for very simple instruction sets.

11.4 PENTIUM AND POWERPC INSTRUCTION FORMATS

Pentium Instruction Formats

The Pentium is equipped with a variety of instruction formats. Of the elements described in this subsection, only the opcode field is always present. Figure 11.8 illustrates the general instruction format. Instructions are made up of from zero to four optional instruction prefixes, a 1- or 2-byte opcode, an optional address specifier (which consists of the ModR/m byte and the Scale Index byte) an optional displacement, and an optional immediate field.

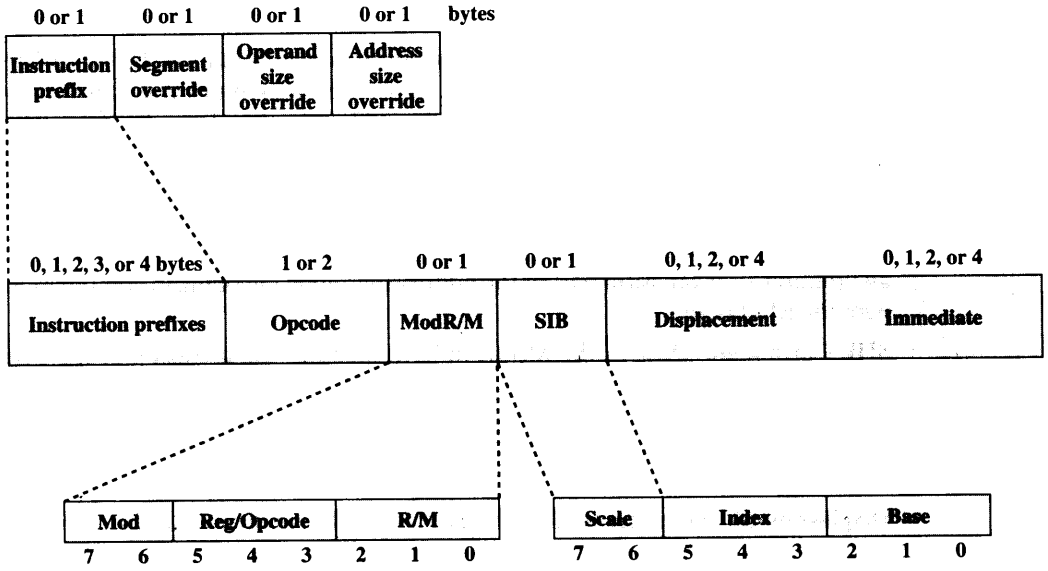


Figure 11.8 Pentium Instruction Format

Let us first consider the prefix bytes:

- **Instruction prefixes:** The instruction prefix, if present, consists of the LOCK prefix or one of the repeat prefixes. The LOCK prefix is used to ensure exclusive use of shared memory in multiprocessor environments. The repeat prefixes specify repeated operation of a string, which enables the Pentium to process strings much faster than with a regular software loop. There are five different repeat prefixes: REP, REPE, REPZ, REPNE, and REPNZ. When the absolute REP prefix is present, the operation specified in the instruction is executed repeatedly on successive elements of the string; the number of repetitions is specified in register CX. The conditional REP prefix causes the instruction to repeat until the count in CX goes to zero or until the condition is met.
- **Segment override:** Explicitly specifies which segment register an instruction should use, overriding the default segment-register selection generated by the Pentium for that instruction.
- **Address size:** The processor can address memory using either 16- or 32-bit addresses. The address size determines the displacement size in instructions and the size of address offsets generated during effective address calculation. One of these sizes is designated as default, and the address size prefix switches between 32-bit and 16-bit address generation.
- **Operand size:** An instruction has a default operand size of 16 or 32 bits, and the operand prefix switches between 32-bit and 16-bit operands.

The instruction itself includes the following fields:

- **Opcode:** One- or two-byte opcode. The opcode may also include bits that specify if data is byte- or full-size (16 or 32 bits depending on context), direction of data operation (to or from memory), and whether an immediate data field must be sign extended.
- **ModR/m:** This byte, and the next, provide addressing information. The ModR/m byte specifies whether an operand is in a register or in memory; if it is in memory, then fields within the byte specify the addressing mode to be used. The ModR/m byte consists of three fields: The Mod field (2 bits) combines with the r/m field to form 32 possible values: 8 registers and 24 indexing modes; the Reg/Opcode field (3 bits) specifies either a register number or three more bits of opcode information; the r/m field (3 bits) can specify a register as the location of an operand, or it can form part of the addressing-mode encoding in combination with the Mod field.
- **SIB:** Certain encoding of the ModR/m byte specifies the inclusion of the SIB byte to specify fully the addressing mode. The SIB byte consists of three fields: The Scale field (2 bits) specifies the scale factor for scaled indexing; the Index field (3 bits) specifies the index register; the Base field (3 bits) specifies the base register.
- **Displacement:** When the addressing-mode specifier indicates that a displacement is used, an 8-, 16-, or 32-bit signed integer displacement field is added.
- **Immediate:** Provides the value of an 8-, 16-, or 32-bit operand.

Several comparisons may be useful here. In the Pentium format, the addressing mode is provided as part of the opcode sequence rather than with each operand. Because only one operand can have address-mode information, only one memory operand can be referenced in an instruction. In contrast, the VAX carries the address-mode information with each operand, allowing memory-to-memory operations. The Pentium instructions are therefore more compact. However, if a memory-to-memory operation is required, the VAX can accomplish this in a single instruction.

The Pentium format allows the use of not only 1-byte, but also 2-byte and 4-byte offsets for indexing. Although the use of the larger index offsets results in longer instructions, this feature provides needed flexibility. For example, it is useful in addressing large arrays or large stack frames. In contrast, the IBM S/370 instruction format allows offsets no greater than 4K bytes (12 bits of offset information), and the offset must be positive. When a location is not in reach of this offset, the compiler must generate extra code to generate the needed address. This problem is especially apparent in dealing with stack frames that have local variables occupying in excess of 4K bytes. As [DEWA90] puts it, "Generating code for the 370 is so painful as a result of that restriction that there have even been compilers for the 370 that simply chose to limit the size of the stack frame to 4K bytes."

As can be seen, the encoding of the Pentium instruction set is very complex. This has to do partly with the need to be backward compatible with the 8086 machine and partly with a desire on the part of the designers to provide every possible assistance to the compiler writer in producing efficient code. It is a matter of some debate whether an instruction set as complex as this is preferable to the opposite extreme of the RISC instruction sets.

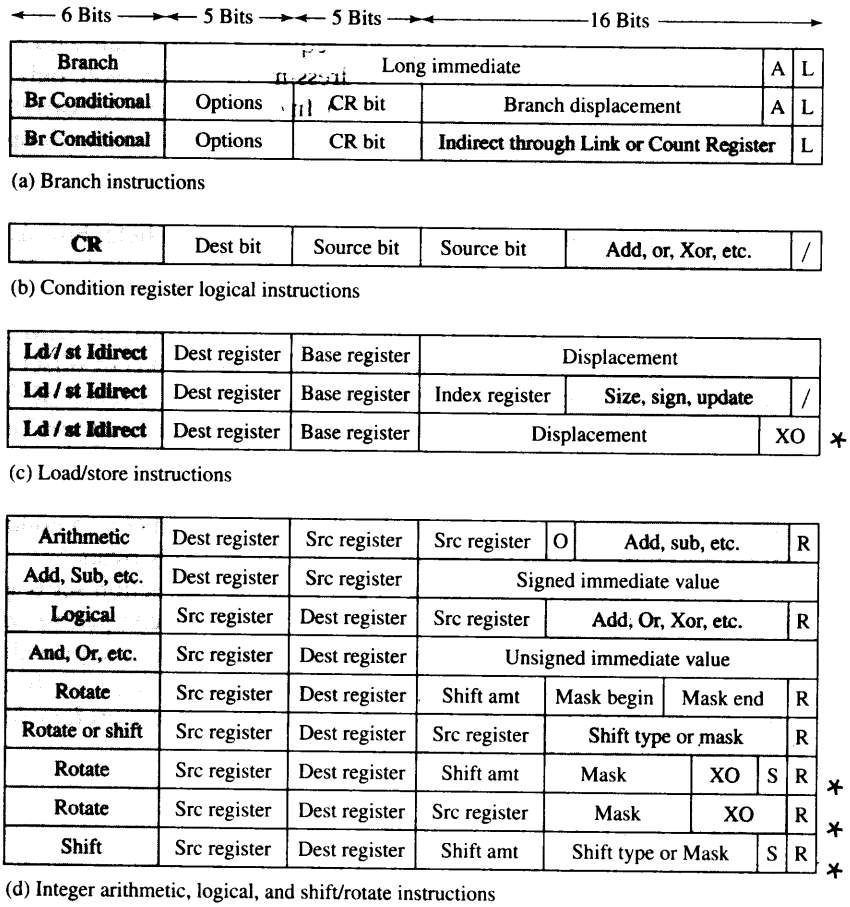
PowerPC Instruction Formats

All instructions in the PowerPC are 32 bits long and follow a regular format. The first 6 bits of an instruction specify the operation to be performed. In some cases, there is an extension to the opcode elsewhere in the instruction that specifies a particular subcase of an operation. In Figure 11.9, opcode bits are represented by the shaded portion of each format.

Note the regular structure of the formats, which eases the job of the instruction decode units. For all load/store, arithmetic, and logical instructions, the opcode is followed by two 5-bit register references, enabling 32 general-purpose registers to be used.

The branch instructions include a link (L) bit that indicates that the effective address of the instruction following the branch instruction is to be placed in the link register. Two forms of the instruction also include a bit (A) that indicates whether the addressing mode is absolute or PC relative. For the conditional branch instructions, the CR bit field specifies the bit to be tested in the condition register. The options field specifies the conditions under which the branch is to be taken. The following conditions may be specified:

- Branch always.
- Branch if count $\neq 0$ and condition is false.
- Branch if count $\neq 0$ and condition is true.
- Branch if count $= 0$ and condition is false.



A = Absolute or PC Relative * = 64-bit implementation only
L = Link to Subroutine
O = Record Overflow in XER
R = Record Conditions in CR1
XO = OPCode Extension
S = Part of Shift Amount Field

Figure 11.9 PowerPC Instruction Formats

- Branch if count = 0 and condition is true.
- Branch if count ≠ 0.
- Branch if count = 0.
- Branch if condition is false.
- Branch if condition is true.

Most instructions that result in a computation (arithmetic, floating-point arithmetic, logical) include a bit that indicates whether the result of the operation should be recorded in the condition register. As will be shown, this feature is useful for branch prediction processing.

Floating-point instructions have fields for three source registers. In many cases, only two source registers are used. A few instructions involve multiplication of two source registers and then addition or subtraction of a third source register. These composite instructions are included because of the frequency of their use. For example, the inner product that is part of many matrix operations can be implemented using multiply-adds.

11.5 RECOMMENDED READING

The references cited in Chapter 10 are equally applicable to the material of this chapter. [BLAA97] contains a detailed discussion of instruction formats and addressing modes. In addition, the reader may wish to consult [FLYN85] for a discussion and analysis of instruction set design issues, particularly those relating to formats.

BLAA97 Blaauw, G., and Brooks, F. *Computer Architecture: Concepts and Evolution.* Reading, MA: Addison-Wesley, 1997.

FLYN85 Flynn, M.; Johnson, J.; and Wakefield, S. "On Instruction Sets and Their Formats." *IEEE Transactions on Computers*, March 1985.

11.6 KEY TERMS, REVIEW QUESTIONS, AND PROBLEMS

Key Terms

autoindexing	immediate addressing	preindexing
base-register addressing	indexing	register addressing
direct addressing	indirect addressing	register indirect addressing
displacement addressing	instruction format	relative addressing
effective address	postindexing	word

Review Questions

11.1 Briefly define immediate addressing.
11.2 Briefly define direct addressing.
11.3 Briefly define indirect addressing.
11.4 Briefly define register addressing.
11.5 Briefly define register indirect addressing.
11.6 Briefly define displacement addressing.
11.7 Briefly define relative addressing.

11.8 What is the advantage of autoindexing?

11.9 What is the difference between postindexing and preindexing?

11.10 What facts go into determining the use of the addressing bits of an instruction?

11.11 What are the advantages and disadvantages of using a variable-length instruction format?

Problems

11.1 Given the following memory values and a one-address machine with an accumulator, what values do the following instructions load into the accumulator?

- Word 20 contains 40.
- Word 30 contains 50.
- Word 40 contains 60.
- Word 50 contains 70.

a. LOAD IMMEDIATE 20
b. LOAD DIRECT 20
c. LOAD INDIRECT 20
d. LOAD IMMEDIATE 30
e. LOAD DIRECT 30
f. LOAD INDIRECT 30

11.2 Let the address stored in the program counter be designated by the symbol X1. The instruction stored in X1 has an address part (operand reference) X2. The operand needed to execute the instruction is stored in the memory word with address X3. An index register contains the value X4. What is the relationship between these various quantities if the addressing mode of the instruction is (a) direct; (b) indirect; (c) PC relative; (d) indexed?

11.3 An address field in an instruction contains decimal value 14. Where is the corresponding operand located for

a. immediate addressing?
b. direct addressing?
c. indirect addressing?
d. register addressing?
e. register indirect addressing?

11.4 Consider a 16-bit processor in which the following appears in main memory, starting at location 200:

200	Load to AC	Mode
201	500	
202	Next instruction	

The first part of the first word indicates that this instruction loads a value into an accumulator. The Mode field specifies an addressing mode and, if appropriate, indicates a source register; assume that when used, the source register is R1, which has a value of 400. There is also a base register that contains the value 100. The value of 500 in location 201 may be part of the address calculation. Assume that location 399 contains the value 999, location 400 contains the value 1000, and so on. Determine the effective address and the operand to be loaded for the following address modes:

a. Direct	**d.** PC relative	**g.** Register indirect
b. Immediate	**e.** Displacement	**h.** Autoindexing with increment, using R1
c. Indirect	**f.** Register	

11.5 A PC-relative mode branch instruction is 3 bytes long. The address of the instruction, in decimal, is 256028. Determine the branch target address if the signed displacement in the instruction is −31.

11.6 A PC-relative mode branch instruction is stored in memory at address 620_{10}. The branch is made to location 530_{10}. The address field in the instruction is 10 bits long. What is the binary value in the instruction?

11.7 How many times does the processor need to refer to memory when it fetches and executes an indirect-address-mode instruction if the instruction is (a) a computation requiring a single operand; (b) a branch?

11.8 The IBM 370 does not provide indirect addressing. Assume that the address of an operand is in main memory. How would you access the operand?

11.9 In [COOK82], the author proposes that the PC-relative addressing modes be eliminated in favor of other modes, such as the use of a stack. What is the disadvantage of this proposal?

11.10 The Pentium includes the following instruction:

IMUL op1, op2, immediate

This instruction multiplies op2, which may be either register or memory, by the immediate operand value, and places the result in op1, which must be a register. There is no other three-operand instruction of this sort in the instruction set. What is the possible use of such an instruction? *Hint:* Consider indexing.

11.11 Consider a processor that includes a base with indexing addressing mode. Suppose an instruction is encountered that employs this addressing mode and specifies a displacement of 1970, in decimal. Currently the base and index register contain the decimal numbers 48022 and 8, respectively. What is the address of the operand?

11.12 Define: EA = (X)+ is the effective address equal to the contents of location X, with X incremented by one word length after the effective address is calculated; EA = −(X) is the effective address equal to the contents of location X, with X decremented by one word length before the effective address is calculated; EA = (X)− is the effective address equal to the contents of location X, with X decremented by one word length after the effective address is calculated. Consider the following instructions, each in the format (Operation Source Operand, Destination Operand), with the result of the operation placed in the destination operand.

a. OP X, (X)
b. OP (X), (X)+
c. OP (X)+, (X)
d. OP −(X), (X)
e. OP −(X), (X)+
f. OP (X)+, (X)+
g. OP (X)−, (X)

Using X as the stack pointer, which of these instructions can pop the top two elements from the stack, perform the designated operation (e.g., ADD source to destination and store in destination), and push the result back on the stack? For each such instruction, does the stack grow toward memory location 0 or in the opposite direction?

11.13 Assume a stack-oriented processor that includes the stack operations PUSH and POP. Arithmetic operations automatically involve the top one or two stack elements. Begin with an empty stack. What stack elements remain after the following instructions are executed?

```
PUSH  4
PUSH  7
PUSH  8
ADD
PUSH  10
SUB
MUL
```

11.14 Justify the assertion that a 32-bit instruction is probably much less than twice as useful as a 16-bit instruction.

11.15 Why was IBM's decision to move from 36 bits to 32 bits per word wrenching, and to whom?

11.16 Assume an instruction set that uses a fixed 16-bit instruction length. Operand specifiers are 6 bits in length. There are K two-operand instructions and L zero-operand instructions. What is the maximum number of one-operand instructions that can be supported?

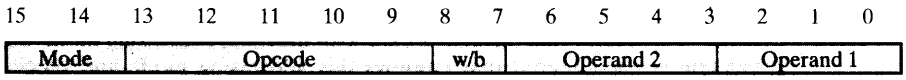
11.17 Design a variable-length opcode to allow all of the following to be encoded in a 36-bit instruction:

- instructions with two 15-bit addresses and one 3-bit register number
- instructions with one 15-bit address and one 3-bit register number
- instructions with no addresses or registers

11.18 Consider the results of Problem 10.6. Assume that M is a 16-bit memory address and that X, Y, and Z are either 16-bit addresses or 4-bit register numbers. The one-address machine uses an accumulator, and the two- and three-address machines have 16 registers and instructions operating on all combinations of memory locations and registers. Assuming 8-bit opcodes and instruction lengths that are multiples of 4 bits, how many bits does each machine need to compute X?

11.19 Is there any possible justification for an instruction with two opcodes?

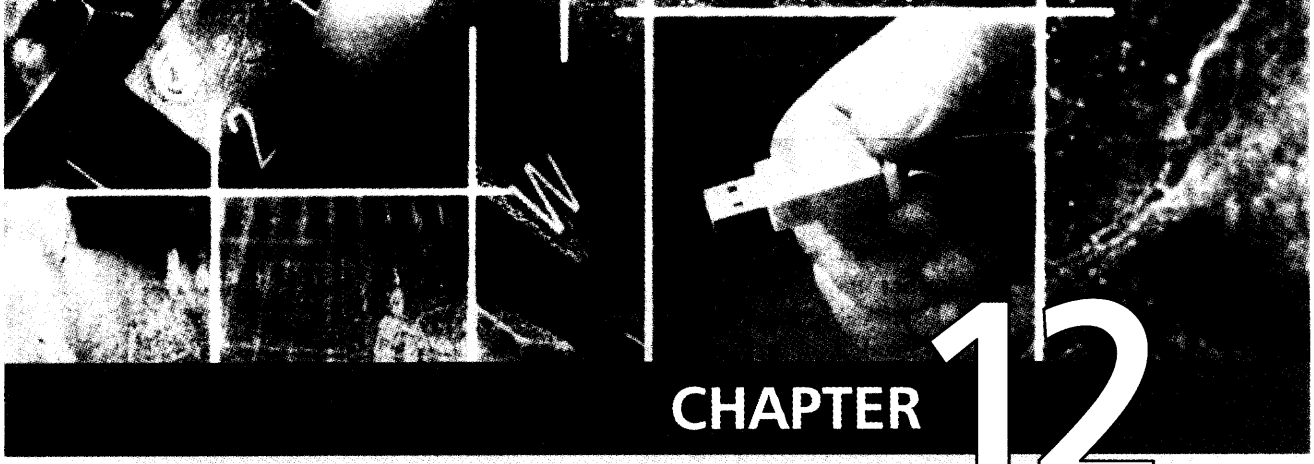
11.20 The 16-bit Zilog Z8001 has the following general instruction format:



The *mode* field specifies how to locate the operands from the *operand* fields. The *w/b* field is used in certain instructions to specify whether the operands are bytes or 16-bit words. The *operand 1* field may (depending on the *mode field* contents) specify one of 16 general-purpose registers. The *operand 2* field may specify any general-purpose registers except register 0. When the *operand 2* field is all zeros, each of the original opcodes takes on a new meaning.

a. How many opcodes are provided on the Z8001?

b. Suggest an efficient way to provide more opcodes and indicate the trade-off involved.



CHAPTER 12

PROCESSOR STRUCTURE AND FUNCTION

KEY POINTS

- A processor includes both user-visible registers and control/status registers. The former may be referenced, implicitly or explicitly, in machine instructions. User-visible registers may be general purpose or have a special use, such as fixed-point or floating-point numbers, addresses, indexes, and segment pointers. Control and status registers are used to control the operation of the processor. One obvious example is the program counter. Another important example is a program status word (PSW) that contains a variety of status and condition bits. These include bits to reflect the result of the most recent arithmetic operation, interrupt enable bits, and an indicator of whether the processor is executing in supervisor or user mode.
- Processors make use of instruction pipelining to speed up execution. In essence, pipelining involves breaking up the instruction cycle into a number of separate stages that occur in sequence, such as fetch instruction, decode instruction, determine operand addresses, fetch operands, execute instruction, and write operand result. Instructions move through these stages, as on an assembly line, so that in principle, each stage can be working on a different instruction at the same time. The occurrence of branches and dependencies between instructions complicates the design and use of pipelines.

This chapter discusses aspects of the processor not yet covered in Part Three and sets the stage for the discussion of RISC and superscalar architecture in Chapters 13 and 14.

We begin with a summary of processor organization. Registers, which form the internal memory of the processor, are then analyzed. We are then in a position to return to the discussion (begun in Section 3.2) of the instruction cycle. A description of the instruction cycle and a common technique known as instruction pipelining complete our description. The chapter concludes with an examination of some additional aspects of the Pentium and PowerPC organizations.

12.1 PROCESSOR ORGANIZATION

To understand the organization of the processor, let us consider the requirements placed on the processor, the things that it must do:

- **Fetch instruction:** The processor reads an instruction from memory (register, cache, main memory).
- **Interpret instruction:** The instruction is decoded to determine what action is required.

- **Fetch data:** The execution of an instruction may require reading data from memory or an I/O module.
- **Process data:** The execution of an instruction may require performing some arithmetic or logical operation on data.
- **Write data:** The results of an execution may require writing data to memory or an I/O module.

To do these things, it should be clear that the processor needs to store some data temporarily. It must remember the location of the last instruction so that it can know where to get the next instruction. It needs to store instructions and data temporarily while an instruction is being executed. In other words, the processor needs a small internal memory.

Figure 12.1 is a simplified view of a processor, indicating its connection to the rest of the system via the system bus. A similar interface would be needed for any of the interconnection structures described in Chapter 3. The reader will recall that the major components of the processor are an *arithmetic and logic unit* (ALU) and a *control unit* (CU). The ALU does the actual computation or processing of data. The control unit controls the movement of data and instructions into and out of the processor and controls the operation of the ALU. In addition, the figure shows a minimal internal memory, consisting of a set of storage locations, called *registers.*

Figure 12.2 is a slightly more detailed view of the processor. The data transfer and logic control paths are indicated, including an element labeled *internal processor bus.* This element is needed to transfer data between the various registers and the ALU because the ALU in fact operates only on data in the internal processor memory. The figure also shows typical basic elements of the ALU. Note the similarity between the internal structure of the computer as a whole and the internal structure of the processor. In both cases, there is a small collection of major elements (computer: processor, I/O, memory; processor: control unit, ALU, registers) connected by data paths.

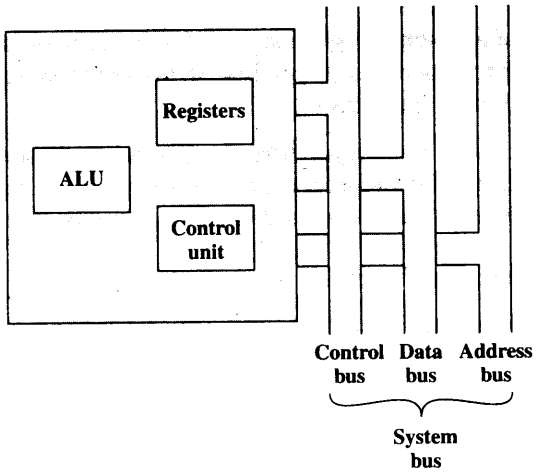


Figure 12.1 The CPU with the System Bus

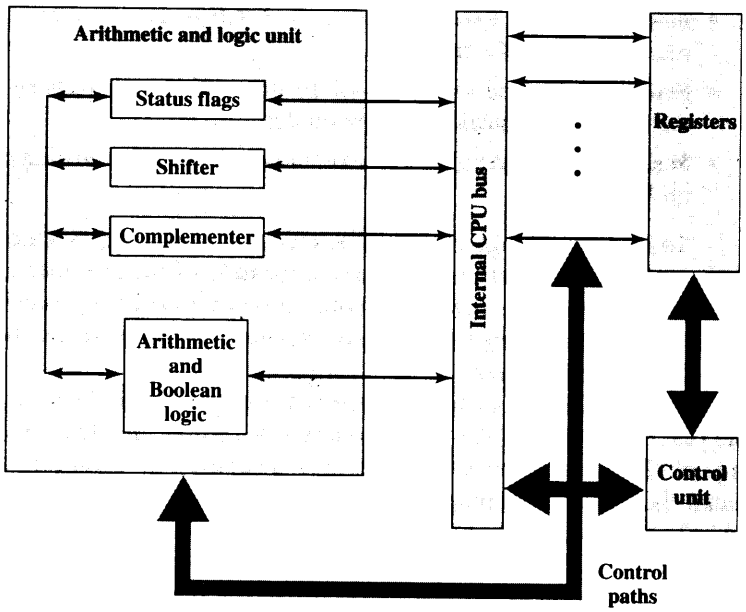


Figure 12.2 Internal Structure of the CPU

12.2 REGISTER ORGANIZATION

As we discussed in Chapter 4, a computer system employs a memory hierarchy. At higher levels of the hierarchy, memory is faster, smaller, and more expensive (per bit). Within the processor, there is a set of registers that function as a level of memory above main memory and cache in the hierarchy. The registers in the processor perform two roles:

- **User-visible registers:** Enable the machine- or assembly language programmer to minimize main memory references by optimizing use of registers.
- **Control and status registers:** Used by the control unit to control the operation of the processor and by privileged, operating system programs to control the execution of programs.

There is not a clean separation of registers into these two categories. For example, on some machines the program counter is user visible (e.g., Pentium), but on many it is not. For purposes of the following discussion, however, we will use these categories.

User-Visible Registers

A user-visible register is one that may be referenced by means of the machine language that the processor executes. We can characterize these in the following categories:

- General purpose
- Data
- Address
- Condition codes

General-purpose registers can be assigned to a variety of functions by the programmer. Sometimes their use within the instruction set is orthogonal to the operation. That is, any general-purpose register can contain the operand for any opcode. This provides true general-purpose register use. Often, however, there are restrictions. For example, there may be dedicated registers for floating-point and stack operations.

In some cases, general-purpose registers can be used for addressing functions (e.g., register indirect, displacement). In other cases, there is a partial or clean separation between data registers and address registers. **Data registers** may be used only to hold data and cannot be employed in the calculation of an operand address. **Address registers** may themselves be somewhat general purpose, or they may be devoted to a particular addressing mode. Examples include the following:

- **Segment pointers:** In a machine with segmented addressing (see Section 8.3), a segment register holds the address of the base of the segment. There may be multiple registers: for example, one for the operating system and one for the current process.
- **Index registers:** These are used for indexed addressing and may be autoindexed.
- **Stack pointer:** If there is user-visible stack addressing, then typically there is a dedicated register that points to the top of the stack. This allows implicit addressing; that is, push, pop, and other stack instructions need not contain an explicit stack operand.

There are several design issues to be addressed here. An important issue is whether to use completely general-purpose registers or to specialize their use. We have already touched on this issue in the preceding chapter because it affects instruction set design. With the use of specialized registers, it can generally be implicit in the opcode which type of register a certain operand specifier refers to. The operand specifier must only identify one of a set of specialized registers rather than one out of all the registers, thus saving bits. On the other hand, this specialization limits the programmer's flexibility.

Another design issue is the number of registers, either general purpose or data plus address, to be provided. Again, this affects instruction set design because more registers require more operand specifier bits. As we previously discussed, somewhere between 8 and 32 registers appears optimum [LUND77]. Fewer registers result in more memory references; more registers do not noticeably reduce memory references (e.g., see [WILL90]). However, a new approach, which finds advantage in the use of hundreds of registers, is exhibited in some RISC systems and is discussed in Chapter 13.

Finally, there is the issue of register length. Registers that must hold addresses obviously must be at least long enough to hold the largest address. Data registers should be able to hold values of most data types. Some machines allow two contiguous registers to be used as one for holding double-length values.

A final category of registers, which is at least partially visible to the user, holds **condition codes** (also referred to as *flags*). Condition codes are bits set by the processor hardware as the result of operations. For example, an arithmetic operation may

Table 12.1 Condition Codes

Advantages	Disadvantages
1. Because condition codes are set by normal arithmetic and data movement instructions, they should reduce the number of COMPARE and TEST instructions needed.	1. Condition codes add complexity, both to the hardware and software. Condition code bits are often modified in different ways by different instructions, making life more difficult for both the microprogrammer and compiler writer.
2. Conditional instructions, such as BRANCH are simplified relative to composite instructions, such as TEST AND BRANCH.	2. Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections.
3. Condition codes facilitate multiway branches. For example, a TEST instruction can be followed by two branches, one on less than or equal to zero and one on greater than zero.	3. Often condition code machines must add special non-condition-code instructions for special situations anyway, such as bit checking, loop control, and atomic semaphore operations.
	4. In a pipelined implementation, condition codes require special synchronization to avoid conflicts.

produce a positive, negative, zero, or overflow result. In addition to the result itself being stored in a register or memory, a condition code is also set. The code may subsequently be tested as part of a conditional branch operation.

Condition code bits are collected into one or more registers. Usually, they form part of a control register. Generally, machine instructions allow these bits to be read by implicit reference, but the programmer cannot alter them.

Many processors, including those based on the IA-64 architecture and the MIPS processors, do not use condition codes at all. Rather, conditional branch instructions specify a comparison to be made and act on the result of the comparison, without storing a condition code. Table 12.1, based on [DERO87], lists key advantages and disadvantages of condition codes.

In some machines, a subroutine call will result in the automatic saving of all user-visible registers, to be restored on return. The processor performs the saving and restoring as part of the execution of call and return instructions. This allows each subroutine to use the user-visible registers independently. On other machines, it is the responsibility of the programmer to save the contents of the relevant user-visible registers prior to a subroutine call, by including instructions for this purpose in the program.

Control and Status Registers

There are a variety of processor registers that are employed to control the operation of the processor. Most of these, on most machines, are not visible to the user. Some of them may be visible to machine instructions executed in a control or operating system mode.

Of course, different machines will have different register organizations and use different terminology. We list here a reasonably complete list of register types, with a brief description.

Four registers are essential to instruction execution:

- **Program counter (PC):** Contains the address of an instruction to be fetched
- **Instruction register (IR):** Contains the instruction most recently fetched
- **Memory address register (MAR):** Contains the address of a location in memory
- **Memory buffer register (MBR):** Contains a word of data to be written to memory or the word most recently read

Not all processors have internal registers designated as MAR and MBR, but some equivalent buffering mechanism is needed whereby the bits to be transferred to the system bus are staged and the bits to be read from the data bus are temporarily stored.

Typically, the processor updates the PC after each instruction fetch so that the PC always points to the next instruction to be executed. A branch or skip instruction will also modify the contents of the PC. The fetched instruction is loaded into an IR, where the opcode and operand specifiers are analyzed. Data are exchanged with memory using the MAR and MBR. In a bus-organized system, the MAR connects directly to the address bus, and the MBR connects directly to the data bus. User-visible registers, in turn, exchange data with the MBR.

The four registers just mentioned are used for the movement of data between the processor and memory. Within the processor, data must be presented to the ALU for processing. The ALU may have direct access to the MBR and user-visible registers. Alternatively, there may be additional buffering registers at the boundary to the ALU; these registers serve as input and output registers for the ALU and exchange data with the MBR and user-visible registers.

Many processor designs include a register or set of registers, often known as the *program status word* (PSW), that contain status information. The PSW typically contains condition codes plus other status information. Common fields or flags include the following:

- **Sign:** Contains the sign bit of the result of the last arithmetic operation.
- **Zero:** Set when the result is 0.
- **Carry:** Set if an operation resulted in a carry (addition) into or borrow (subtraction) out of a high-order bit. Used for multiword arithmetic operations.
- **Equal:** Set if a logical compare result is equality.
- **Overflow:** Used to indicate arithmetic overflow.
- **Interrupt enable/disable:** Used to enable or disable interrupts.
- **Supervisor:** Indicates whether the processor is executing in supervisor or user mode. Certain privileged instructions can be executed only in supervisor mode, and certain areas of memory can be accessed only in supervisor mode.

A number of other registers related to status and control might be found in a particular processor design. There may be a pointer to a block of memory containing additional status information (e.g., process control blocks). In machines using vectored interrupts, an interrupt vector register may be provided. If a stack is used to implement certain functions (e.g., subroutine call), then a system stack pointer is

needed. A page table pointer is used with a virtual memory system. Finally, registers may be used in the control of I/O operations.

A number of factors go into the design of the control and status register organization. One key issue is operating system support. Certain types of control information are of specific utility to the operating system. If the processor designer has a functional understanding of the operating system to be used, then the register organization can to some extent be tailored to the operating system.

Another key design decision is the allocation of control information between registers and memory. It is common to dedicate the first (lowest) few hundred or thousand words of memory for control purposes. The designer must decide how much control information should be in registers and how much in memory. The usual trade-off of cost versus speed arises.

Example Microprocessor Register Organizations

It is instructive to examine and compare the register organization of comparable systems. In this section, we look at two 16-bit microprocessors that were designed at about the same time: the Motorola MC68000 [STRI79] and the Intel 8086 [MORS78]. Figures 12.3a and b depict the register organization of each; purely internal registers, such as a memory address register, are not shown.

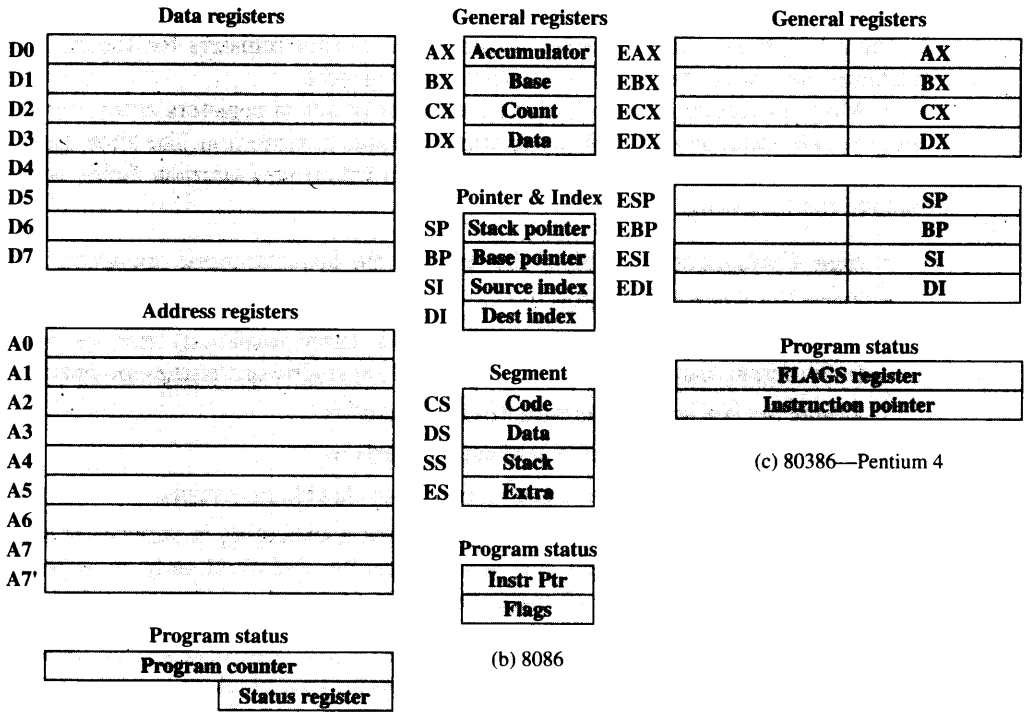


Figure 12.3 Example Microprocessor Register Organizations

The MC68000 partitions its 32-bit registers into eight data registers and nine address registers. The eight data registers are used primarily for data manipulation and are also used in addressing as index registers. The width of the registers allows 8-, 16-, and 32-bit data operations, determined by opcode. The address registers contain 32-bit (no segmentation) addresses; two of these registers are also used as stack pointers, one for users and one for the operating system, depending on the current execution mode. Both registers are numbered 7, because only one can be used at a time. The MC68000 also includes a 32-bit program counter and a 16-bit status register.

The Motorola team wanted a very regular instruction set, with no special-purpose registers. A concern for code efficiency led them to divide the registers into two functional components, saving one bit on each register specifier. This seems a reasonable compromise between complete generality and code compaction.

The Intel 8086 takes a different approach to register organization. Every register is special purpose, although some registers are also usable as general purpose. The 8086 contains four 16-bit data registers that are addressable on a byte or 16-bit basis, and four 16-bit pointer and index registers. The data registers can be used as general purpose in some instructions. In others, the registers are used implicitly. For example, a multiply instruction always uses the accumulator. The four pointer registers are also used implicitly in a number of operations; each contains a segment offset. There are also four 16-bit segment registers. Three of the four segment registers are used in a dedicated, implicit fashion, to point to the segment of the current instruction (useful for branch instructions), a segment containing data, and a segment containing a stack, respectively. These dedicated and implicit uses provide for compact encoding at the cost of reduced flexibility. The 8086 also includes an instruction pointer and a set of 1-bit status and control flags.

The point of this comparison should be clear. There is no universally accepted philosophy concerning the best way to organize processor registers [TOON81]. As with overall instruction set design and so many other processor design issues, it is still a matter of judgment and taste.

A second instructive point concerning register organization design is illustrated in Figure 12.3c. This figure shows the user-visible register organization for the Intel 80386 [ELAY85], which is a 32-bit microprocessor designed as an extension of the 8086.[1] The 80386 uses 32-bit registers. However, to provide upward compatibility for programs written on the earlier machine, the 80386 retains the original register organization embedded in the new organization. Given this design constraint, the architects of the 32-bit processors had limited flexibility in designing the register organization.

12.3 INSTRUCTION CYCLE

In Section 3.2, we described the processor's instruction cycle (Figure 3.9). To recall, an instruction cycle includes the following subcycles:

[1]Because the MC68000 already uses 32-bit registers, the MC68020 [MACG84], which is a full 32-bit architecture, uses the same register organization.

- **Fetch:** Read the next instruction from memory into the processor.
- **Execute:** Interpret the opcode and perform the indicated operation.
- **Interrupt:** If interrupts are enabled and an interrupt has occurred, save the current process state and service the interrupt.

We are now in a position to elaborate somewhat on the instruction cycle. First, we must introduce one additional subcycle, known as the indirect cycle.

The Indirect Cycle

We have seen, in Chapter 11, that the execution of an instruction may involve one or more operands in memory, each of which requires a memory access. Further, if indirect addressing is used, then additional memory accesses are required.

We can think of the fetching of indirect addresses as one more instruction subcycle. The result is shown in Figure 12.4. The main line of activity consists of alternating instruction fetch and instruction execution activities. After an instruction is fetched, it is examined to determine if any indirect addressing is involved. If so, the required operands are fetched using indirect addressing. Following execution, an interrupt may be processed before the next instruction fetch.

Another way to view this process is shown in Figure 12.5, which is a revised version of Figure 3.12. This illustrates more correctly the nature of the instruction cycle. Once an instruction is fetched, its operand specifiers must be identified. Each input operand in memory is then fetched, and this process may require indirect addressing. Register-based operands need not be fetched. Once the opcode is executed, a similar process may be needed to store the result in main memory.

Data Flow

The exact sequence of events during an instruction cycle depends on the design of the processor. We can, however, indicate in general terms what must happen. Let us assume that a processor that employs a memory address register (MAR), a memory buffer register (MBR), a program counter (PC), and an instruction register (IR).

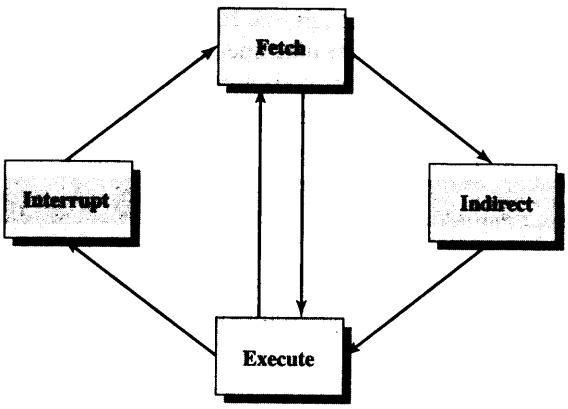


Figure 12.4 The Instruction Cycle

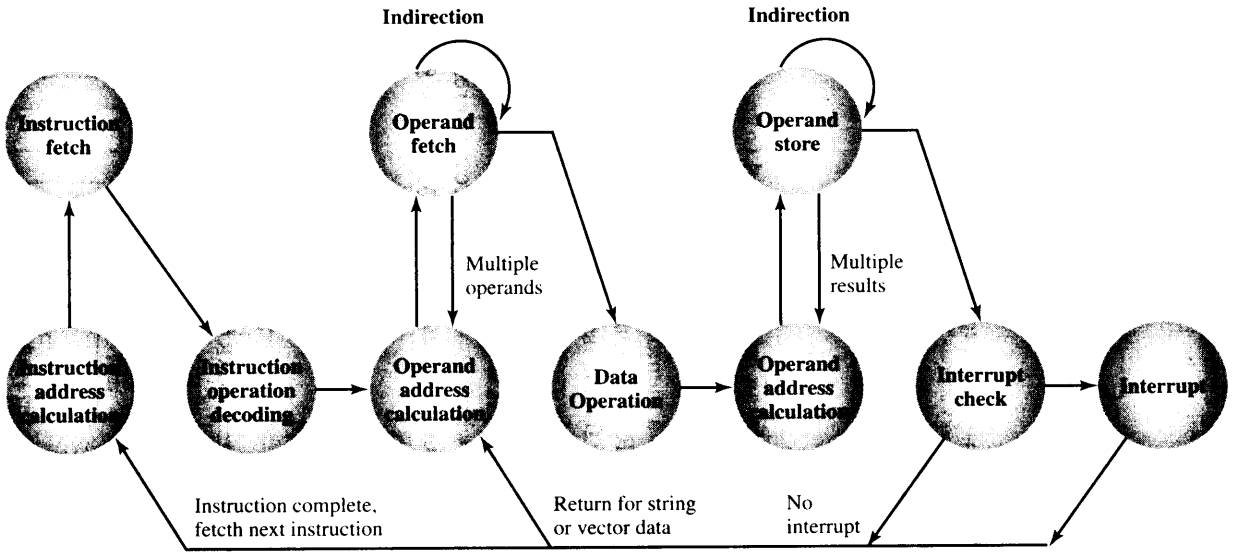


Figure 12.5 Instruction Cycle State Diagram

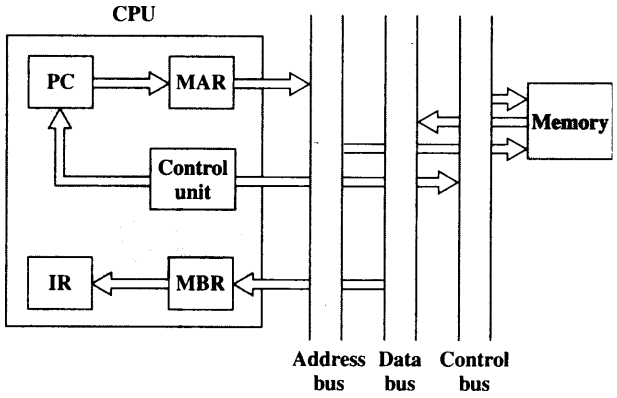


Figure 12.6 Data Flow, Fetch Cycle

During the *fetch cycle,* an instruction is read from memory. Figure 12.6 shows the flow of data during this cycle. The PC contains the address of the next instruction to be fetched. This address is moved to the MAR and placed on the address bus. The control unit requests a memory read, and the result is placed on the data bus and copied into the MBR and then moved to the IR. Meanwhile, the PC is incremented by 1, preparatory for the next fetch.

Once the fetch cycle is over, the control unit examines the contents of the IR to determine if it contains an operand specifier using indirect addressing. If so, an *indirect cycle* is performed. As shown in Figure 12.7, this is a simple cycle. The rightmost N bits of the MBR, which contain the address reference, are transferred to the MAR. Then the control unit requests a memory read, to get the desired address of the operand into the MBR.

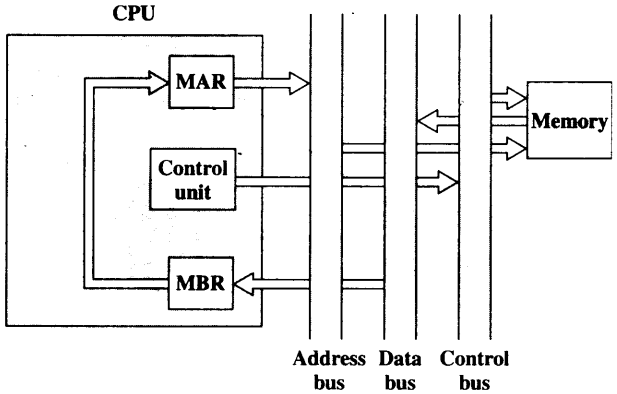


Figure 12.7 Data Flow, Indirect Cycle

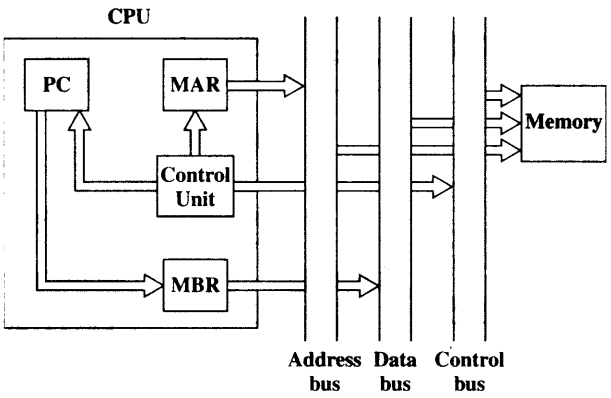


Figure 12.8 Data Flow, Interrupt Cycle

The fetch and indirect cycles are simple and predictable. The *execute cycle* takes many forms; the form depends on which of the various machine instructions is in the IR. This cycle may involve transferring data among registers, read or write from memory or I/O, and/or the invocation of the ALU.

Like the fetch and indirect cycles, the *interrupt cycle* is simple and predictable (Figure 12.8). The current contents of the PC must be saved so that the processor can resume normal activity after the interrupt. Thus, the contents of the PC are transferred to the MBR to be written into memory. The special memory location reserved for this purpose is loaded into the MAR from the control unit. It might, for example, be a stack pointer. The PC is loaded with the address of the interrupt routine. As a result, the next instruction cycle will begin by fetching the appropriate instruction.

12.4 INSTRUCTION PIPELINING

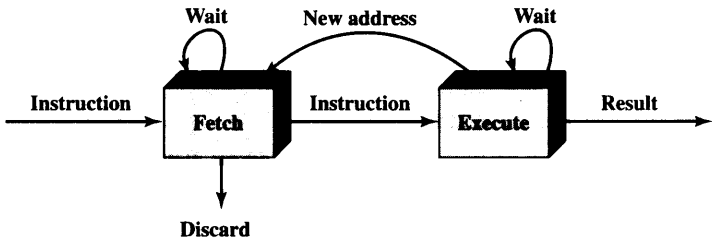
As computer systems evolve, greater performance can be achieved by taking advantage of improvements in technology, such as faster circuitry. In addition, organizational enhancements to the processor can improve performance. We have already seen some examples of this, such as the use of multiple registers rather than a single accumulator, and the use of a cache memory. Another organizational approach, which is quite common, is instruction pipelining.

Pipelining Strategy

Instruction pipelining is similar to the use of an assembly line in a manufacturing plant. An assembly line takes advantage of the fact that a product goes through various stages of production. By laying the production process out in an assembly line, products at various stages can be worked on simultaneously. This process is also referred to as *pipelining,* because, as in a pipeline, new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.

To apply this concept to instruction execution, we must recognize that, in fact, an instruction has a number of stages. Figure 12.5, for example, breaks the

(a) Simplified view



(b) Expanded view

Figure 12.9 Two-Stage Instruction Pipeline

instruction cycle up into 10 tasks, which occur in sequence. Clearly, there should be some opportunity for pipelining.

As a simple approach, consider subdividing instruction processing into two stages: fetch instruction and execute instruction. There are times during the execution of an instruction when main memory is not being accessed. This time could be used to fetch the next instruction in parallel with the execution of the current one. Figure 12.9a depicts this approach. The pipeline has two independent stages. The first stage fetches an instruction and buffers it. When the second stage is free, the first stage passes it the buffered instruction. While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch and buffer the next instruction. This is called *instruction prefetch* or *fetch overlap*.

It should be clear that this process will speed up instruction execution. If the fetch and execute stages were of equal duration, the instruction cycle time would be halved. However, if we look more closely at this pipeline (Figure 12.9b), we will see that this doubling of execution rate is unlikely for two reasons:

1. The execution time will generally be longer than the fetch time. Execution will involve reading and storing operands and the performance of some operation. Thus, the fetch stage may have to wait for some time before it can empty its buffer.
2. A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched.

Guessing can reduce the time loss from the second reason. A simple rule is the following: When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction. Then, if the branch is not taken, no time is lost. If the branch is taken, the fetched instruction must be discarded and a new instruction fetched.

While these factors reduce the potential effectiveness of the two-stage pipeline, some speedup occurs. To gain further speedup, the pipeline must have more stages. Let us consider the following decomposition of the instruction processing.

- **Fetch instruction (FI):** Read the next expected instruction into a buffer.
- **Decode instruction (DI):** Determine the opcode and the operand specifiers.
- **Calculate operands (CO):** Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of address calculation.
- **Fetch operands (FO):** Fetch each operand from memory. Operands in registers need not be fetched.
- **Execute instruction (EI):** Perform the indicated operation and store the result, if any, in the specified destination operand location.
- **Write operand (WO):** Store the result in memory.

With this decomposition, the various stages will be of more nearly equal duration. For the sake of illustration, let us assume equal duration. Using this assumption, Figure 12.10 shows that a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.

Several comments are in order: The diagram assumes that each instruction goes through all six stages of the pipeline. This will not always be the case. For example, a load instruction does not need the WO stage. However, to simplify the pipeline hardware, the timing is set up assuming that each instruction requires all six stages. Also, the diagram assumes that all of the stages can be performed in parallel. In particular, it is assumed that there are no memory conflicts. For example, the FI, FO, and WO stages involve a memory access. The diagram implies that all these accesses can

Time →

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO	EI	WO					
Instruction 5					FI	DI	CO	FO	EI	WO				
Instruction 6						FI	DI	CO	FO	EI	WO			
Instruction 7							FI	DI	CO	FO	EI	WO		
Instruction 8								FI	DI	CO	FO	EI	WO	
Instruction 9									FI	DI	CO	FO	EI	WO

Figure 12.10 Timing Diagram for Instruction Pipeline Operation

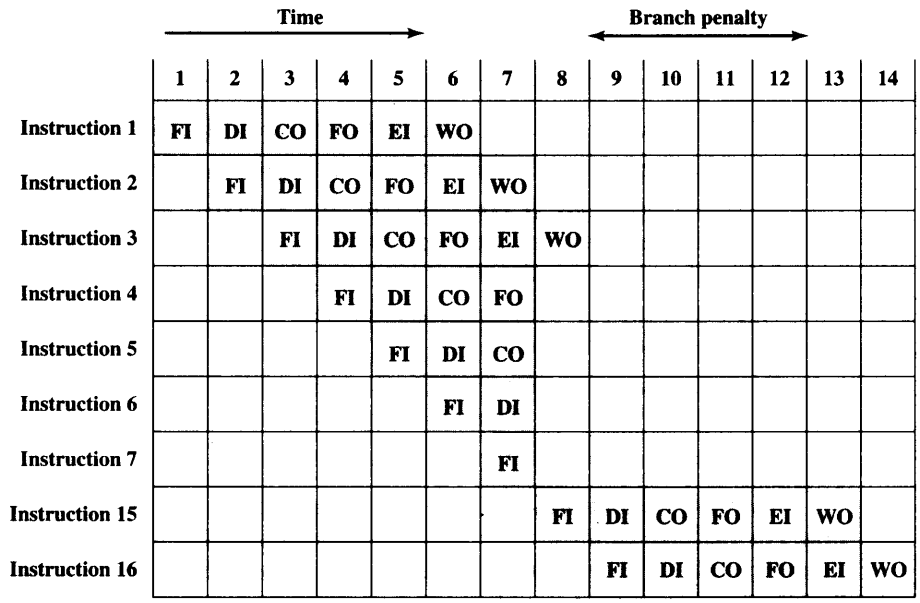


Figure 12.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

occur simultaneously. Most memory systems will not permit that. However, the desired value may be in cache, or the FO or WO stage may be null. Thus, much of the time, memory conflicts will not slow down the pipeline.

Several other factors serve to limit the performance enhancement. If the six stages are not of equal duration, there will be some waiting involved at various pipeline stages, as discussed before for the two-stage pipeline. Another difficulty is the conditional branch instruction, which can invalidate several instruction fetches. A similar unpredictable event is an interrupt. Figure 12.11 illustrates the effects of the conditional branch, using the same program as Figure 12.10. Assume that instruction 3 is a conditional branch to instruction 15. Until the instruction is executed, there is no way of knowing which instruction will come next. The pipeline, in this example, simply loads the next instruction in sequence (instruction 4) and proceeds. In Figure 12.10, the branch is not taken, and we get the full performance benefit of the enhancement. In Figure 12.11, the branch is taken. This is not determined until the end of time unit 7. At this point, the pipeline must be cleared of instructions that are not useful. During time unit 8, instruction 15 enters the pipeline. No instructions complete during time units 9 through 12; this is the performance penalty incurred because we could not anticipate the branch. Figure 12.12 indicates the logic needed for pipelining to account for branches and interrupts.

Other problems arise that did not appear in our simple two-stage organization. The CO stage may depend on the contents of a register that could be altered by a previous instruction that is still in the pipeline. Other such register and memory conflicts could occur. The system must contain logic to account for this type of conflict.

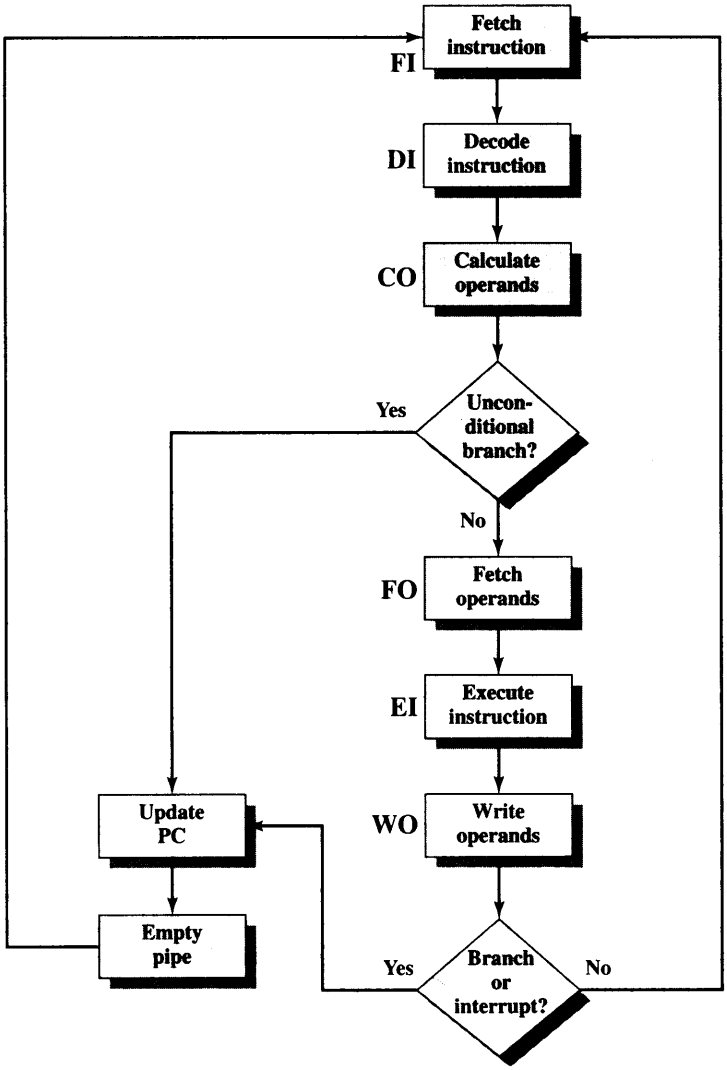


Figure 12.12 Six-Stage CPU Instruction Pipeline

To clarify pipeline operation, it might be useful to look at an alternative depiction. Figures 12.10 and 12.11 show the progression of time horizontally across the figures, with each row showing the progress of an individual instruction. Figure 12.13 shows same sequence of events, with time progressing vertically down the figure, and each row showing the state of the pipeline at a given point in time. In Figure 12.13a (which corresponds to Figure 12.10), the pipeline is full at time 6, with 6 different instructions in various stages of execution, and remains full through time 9; we assume that instruction I9 is the last instruction to be executed. In Figure 12.13b, (which corresponds to Figure 12.11), the pipeline is full at times 6 and 7. At time 7, instruction 3 is in the execute stage and executes a branch to instruction 15. At this point, instructions I4 through I7

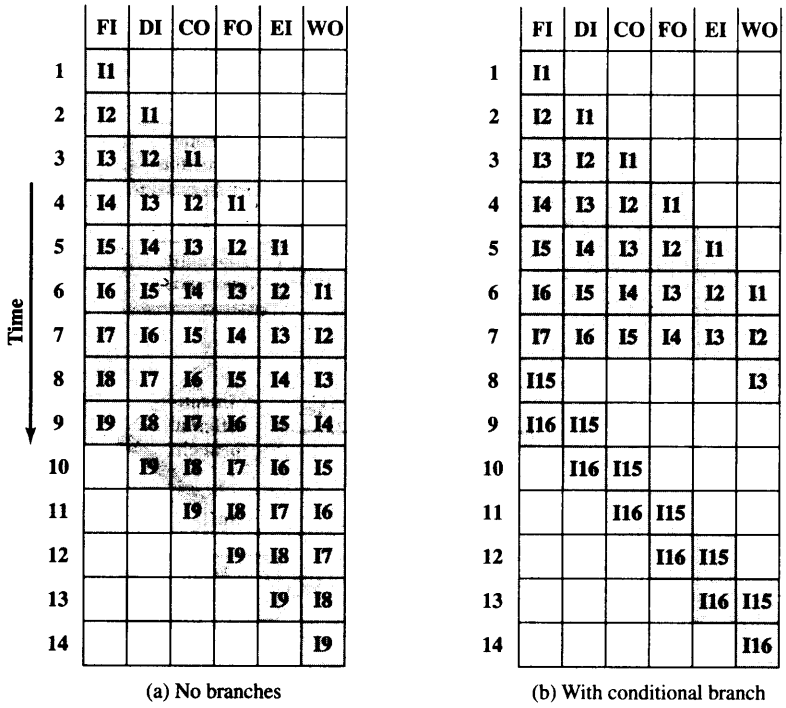


Figure 12.13 An Alternative Pipeline Depiction

are flushed from the pipeline, so that at time 8, only two instructions are in the pipeline, I3 and I15.

From the preceding discussion, it might appear that the greater the number of stages in the pipeline, the faster the execution rate. Some of the IBM S/360 designers pointed out two factors that frustrate this seemingly simple pattern for high-performance design [ANDE67a], and they remain elements that designer must still consider:

1. At each stage of the pipeline, there is some overhead involved in moving data from buffer to buffer and in performing various preparation and delivery functions. This overhead can appreciably lengthen the total execution time of a single instruction. This is significant when sequential instructions are logically dependent, either through heavy use of branching or through memory access dependencies.
2. The amount of control logic required to handle memory and register dependencies and to optimize the use of the pipeline increases enormously with the number of stages. This can lead to a situation where the logic controlling the gating between stages is more complex than the stages being controlled.

Instruction pipelining is a powerful technique for enhancing performance but requires careful design to achieve optimum results with reasonable complexity.

Pipeline Performance

In this subsection, we develop some simple measures of pipeline performance and relative speedup (based on a discussion in [HWAN93]). The cycle time τ of an instruction pipeline is the time needed to advance a set of instructions one stage through the pipeline; each column in Figures 12.10 and 12.11 represents one cycle time. The cycle time can be determined as

$$\tau = \max_i[\tau_i] + d = \tau_m + d \qquad 1 \leq i \leq k$$

where

τ_i = time delay of the circuitry in the ith stage of the pipeline

τ_m = maximum stage delay (delay through stage which experiences the largest delay)

k = number of stages in the instruction pipeline

d = time delay of a latch, needed to advance signals and data from one stage to the next

In general, the time delay d is equivalent to a clock pulse and $\tau_m \gg d$. Now suppose that n instructions are processed, with no branches. Let $T_{k,n}$ be the total time required for a pipeline with k stages to execute n instructions. Then

$$T_{k,n} = [k + (n - 1)]\tau \tag{12.1}$$

A total of k cycles are required to complete the execution of the first instruction, and the remaining $n - 1$ instructions require $n - 1$ cycles.[2] This equation is easily verified from Figure 12.10. The ninth instruction completes at time cycle 14:

$$14 = [6 + (9 - 1)]$$

Now consider an processor with equivalent functions but no pipeline, and assume that the instruction cycle time is $k\tau$. The speedup factor for the instruction pipeline compared to execution without the pipeline is defined as

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)} \tag{12.2}$$

Figure 12.14a plots the speedup factor as a function of the number of instructions that are executed without a branch. As might be expected, at the limit ($n \rightarrow \infty$), we have a k-fold speedup. Figure 12.14b shows the speedup factor as a function of the number of stages in the instruction pipeline.[3] In this case, the speedup factor approaches the number of instructions that can be fed into the pipeline without branches. Thus, the larger the number of pipeline stages, the greater the potential for speedup. However, as a practical matter, the potential gains of additional pipeline

[2]We are being a bit sloppy here. The cycle time will only equal the maximum value of τ when all the stages are full. At the beginning, the cycle time may be less for the first one or few cycles.

[3]Note that the x-axis is logarithmic in Figure 12.14a and linear in Figure 12.14b.

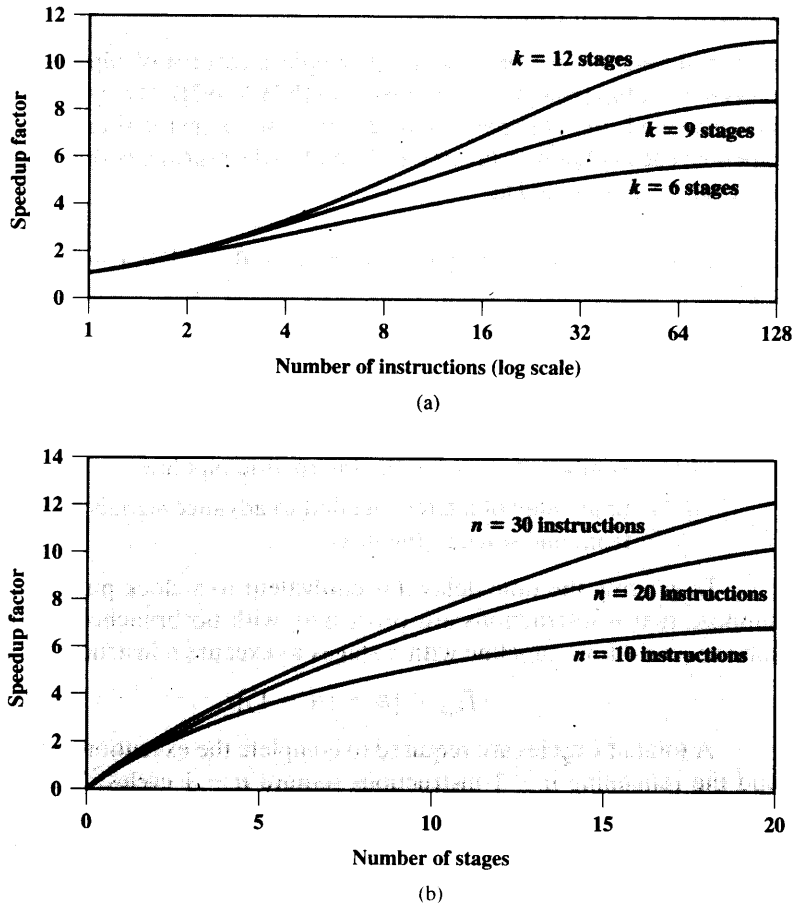


Figure 12.14 Speedup Factors with Instruction Pipelining

stages are countered by increases in cost, delays between stages, and the fact that branches will be encountered requiring the flushing of the pipeline.

Dealing with Branches

One of the major problems in designing an instruction pipeline is assuring a steady flow of instructions to the initial stages of the pipeline. The primary impediment, as we have seen, is the conditional branch instruction. Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not.

A variety of approaches have been taken for dealing with conditional branches:

- Multiple streams
- Prefetch branch target
- Loop buffer
- Branch prediction
- Delayed branch

Multiple Streams A simple pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and may make the wrong choice. A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams. There are two problems with this approach:

- With multiple pipelines there are contention delays for access to the registers and to memory.
- Additional branch instructions may enter the pipeline (either stream) before the original branch decision is resolved. Each such instruction needs an additional stream.

Despite these drawbacks, this strategy can improve performance. Examples of machines with two or more pipeline streams are the IBM 370/168 and the IBM 3033.

Prefetch Branch Target When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed. If the branch is taken, the target has already been prefetched.

The IBM 360/91 uses this approach.

Loop Buffer A loop buffer is a small, very-high-speed memory maintained by the instruction fetch stage of the pipeline and containing the n most recently fetched instructions, in sequence. If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the next instruction is fetched from the buffer. The loop buffer has three benefits:

1. With the use of prefetching, the loop buffer will contain some instruction sequentially ahead of the current instruction fetch address. Thus, instructions fetched in sequence will be available without the usual memory access time.
2. If a branch occurs to a target just a few locations ahead of the address of the branch instruction, the target will already be in the buffer. This is useful for the rather common occurrence of IF–THEN and IF–THEN–ELSE sequences.
3. This strategy is particularly well suited to dealing with loops, or iterations; hence the name *loop buffer*. If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched from memory only once, for the first iteration. For subsequent iterations, all the needed instructions are already in the buffer.

The loop buffer is similar in principle to a cache dedicated to instructions. The differences are that the loop buffer only retains instructions in sequence and is much smaller in size and hence lower in cost.

Figure 12.15 gives an example of a loop buffer. If the buffer contains 256 bytes, and byte addressing is used, then the least significant 8 bits are used to index the buffer. The remaining most significant bits are checked to determine if the branch target lies within the environment captured by the buffer.

Among the machines using a loop buffer are some of the CDC machines (Star-100, 6600, 7600) and the CRAY-1. A specialized form of loop buffer is available on the Motorola 68010, for executing a three-instruction loop involving the

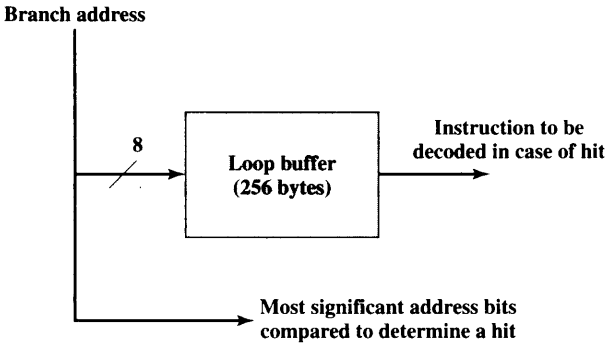


Figure 12.15 Loop Buffer

DBcc (decrement and branch on condition) instruction (see Problem 12.14). A three-word buffer is maintained, and the processor executes these instructions repeatedly until the loop condition is satisfied.

Branch Prediction Various techniques can be used to predict whether a branch will be taken. Among the more common are the following:

- Predict never taken
- Predict always taken
- Predict by opcode
- Taken/not taken switch
- Branch history table

The first three approaches are static: They do not depend on the execution history up to the time of the conditional branch instruction. The latter two approaches are dynamic: They depend on the execution history.

The first two approaches are the simplest. These either always assume that the branch will not be taken and continue to fetch instructions in sequence, or they always assume that the branch will be taken and always fetch from the branch target. The 68020 and the VAX 11/780 use the predict-never-taken approach. The VAX 11/780 also includes a feature to minimize the effect of a wrong decision. If the fetch of the instruction after the branch will cause a page fault or protection violation, the processor halts its prefetching until it is sure that the instruction should be fetched.

Studies analyzing program behavior have shown that conditional branches are taken more than 50% of the time [LILJ88], and so if the cost of prefetching from either path is the same, then always prefetching from the branch target address should give better performance than always prefetching from the sequential path. However, in a paged machine, prefetching the branch target is more likely to cause a page fault than prefetching the next instruction in sequence, and so this performance penalty should be taken into account. An avoidance mechanism may be employed to reduce this penalty.

The final static approach makes the decision based on the opcode of the branch instruction. The processor assumes that the branch will be taken for certain branch opcodes and not for others. [LILJ88] reports success rates of greater than 75% with this strategy.

Dynamic branch strategies attempt to improve the accuracy of prediction by recording the history of conditional branch instructions in a program. For example, one or more bits can be associated with each conditional branch instruction that reflect the recent history of the instruction. These bits are referred to as a taken/not taken switch that directs the processor to make a particular decision the next time the instruction is encountered. Typically, these history bits are not associated with the instruction in main memory. Rather, they are kept in temporary high-speed storage. One possibility is to associate these bits with any conditional branch instruction that is in a cache. When the instruction is replaced in the cache, its history is lost. Another possibility is to maintain a small table for recently executed branch instructions with one or more history bits in each entry. The processor could access the table associatively, like a cache, or by using the low-order bits of the branch instruction's address.

With a single bit, all that can be recorded is whether the last execution of this instruction resulted in a branch or not. A shortcoming of using a single bit appears in the case of a conditional branch instruction that is almost always taken, such as a loop instruction. With only one bit of history, an error in prediction will occur twice for each use of the loop: once on entering the loop, and once on exiting.

If two bits are used, they can be used to record the result of the last two instances of the execution of the associated instruction, or to record a state in some other fashion. Figure 12.16 shows a typical approach (see Problem 12.13 for other

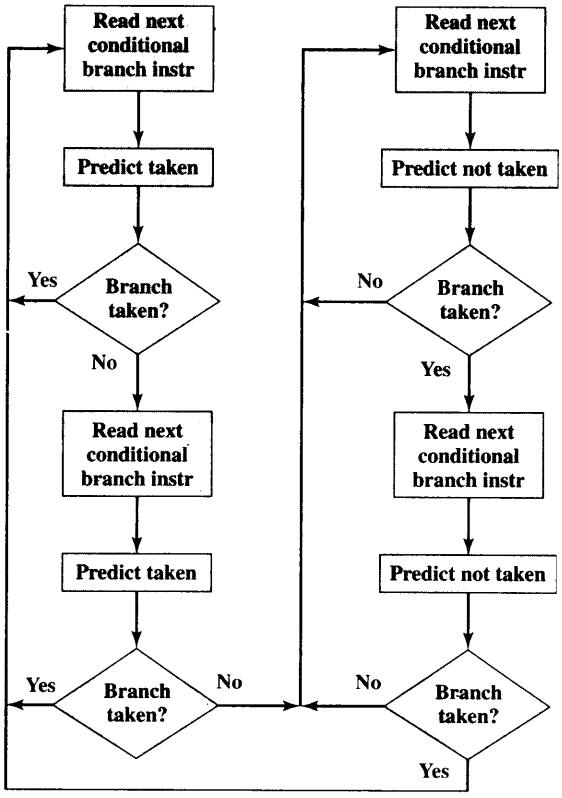


Figure 12.16 Branch Prediction Flowchart

possibilities). Assume that the algorithm starts at the upper left-hand corner of the flowchart. As long as each succeeding conditional branch instruction that is encountered is taken, the decision process predicts that the next branch will be taken. If a single prediction is wrong, the algorithm continues to predict that the next branch is taken. Only if two successive branches are not taken does the algorithm shift to the right-hand side of the flowchart. Subsequently, the algorithm will predict that branches are not taken until two branches in a row are taken. Thus, the algorithm requires two consecutive wrong predictions to change the prediction decision.

The decision process can be represented more compactly by a finite-state machine, shown in Figure 12.17. The finite-state machine representation is commonly used in the literature.

The use of history bits, as just described, has one drawback: If the decision is made to take the branch, the target instruction cannot be fetched until the target address, which is an operand in the conditional branch instruction, is decoded. Greater efficiency could be achieved if the instruction fetch could be initiated as soon as the branch decision is made. For this purpose, more information must be saved, in what is known as a branch target buffer, or a branch history table.

The branch history table is a small cache memory associated with the instruction fetch stage of the pipeline. Each entry in the table consists of three elements: the address of a branch instruction, some number of history bits that record the state of use of that instruction, and information about the target instruction. In most proposals and implementations, this third field contains the address of the target instruction. Another possibility is for the third field to actually contain the target instruction. The trade-off is clear: Storing the target address yields a smaller table but a greater instruction fetch time compared with storing the target instruction [RECH98].

Figure 12.18 contrasts this scheme with a predict-never-taken strategy. With the former strategy, the instruction fetch stage always fetches the next sequential address. If a branch is taken, some logic in the processor detects this and instructs that the next instruction be fetched from the target address (in addition to flushing the pipeline). The

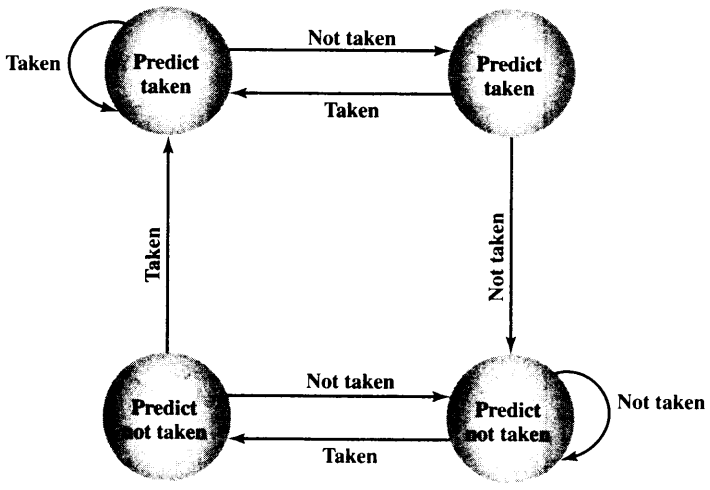


Figure 12.17 Branch Prediction State Diagram

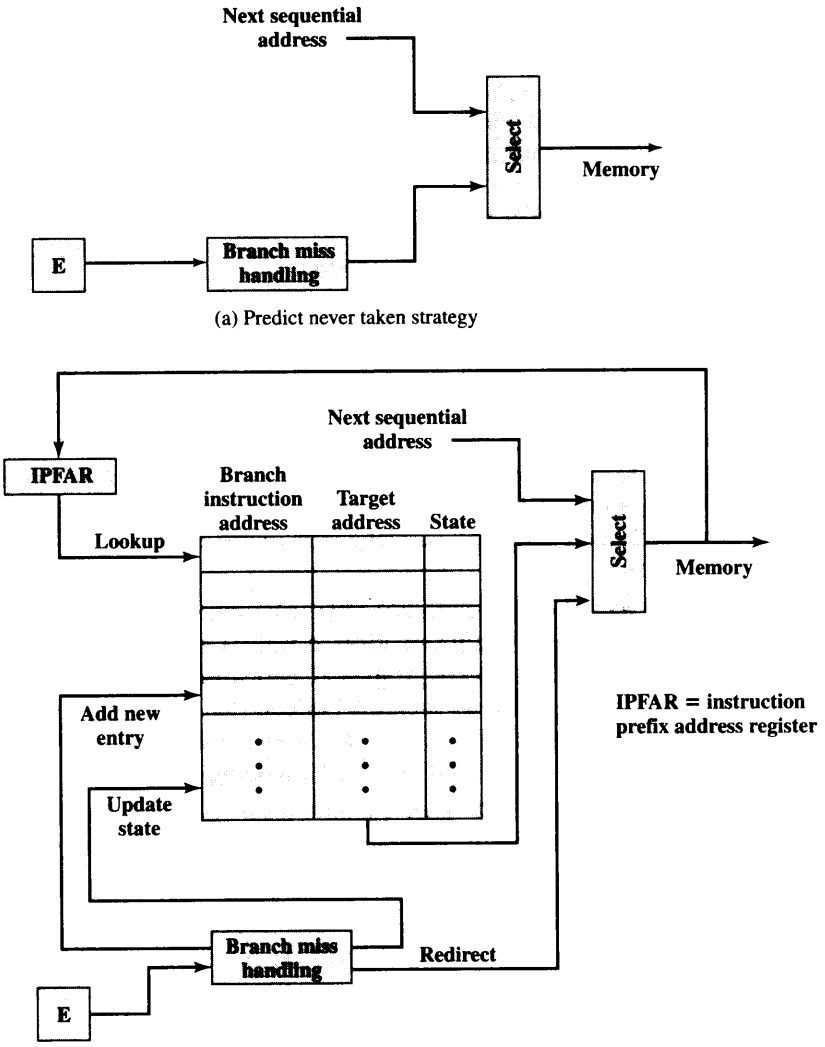


Figure 12.18 Dealing with Branches

branch history table is treated as a cache. Each prefetch triggers a lookup in the branch history table. If no match is found, the next sequential address is used for the fetch. If a match is found, a prediction is made based on the state of the instruction: Either the next sequential address or the branch target address is fed to the select logic.

When the branch instruction is executed, the execute stage signals the branch history table logic with the result. The state of the instruction is updated to reflect a correct or incorrect prediction. If the prediction is incorrect, the select logic is redirected to the correct address for the next fetch. When a conditional branch instruction is encountered that is not in the table, it is added to the table and one of the existing entries is discarded, using one of the cache replacement algorithms discussed in Chapter 4.

One example of an implementation of a branch history table is the Advanced Micro Device AMD29000 microprocessor.

Delayed Branch It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired. This intriguing approach is examined in Chapter 13.

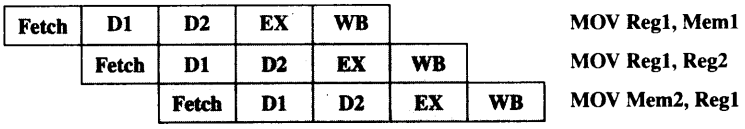
Intel 80486 Pipelining

The 80486 implements a five-stage pipeline:

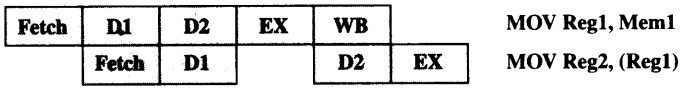
- **Fetch:** Instructions are fetched from the cache or from external memory and placed into one of the two 16-byte prefetch buffers. The objective of the fetch stage is to fill the prefetch buffers with new data as soon as the old data have been consumed by the instruction decoder. Because instructions are of variable length (from 1 to 11 bytes not counting prefixes), the status of the prefetcher relative to the other pipeline stages varies from instruction to instruction. On average, about five instructions are fetched with each 16-byte load [CRAW90]. The fetch stage operates independently of the other stages to keep the prefetch buffers full.
- **Decode stage 1:** All opcode and addressing-mode information is decoded in the D1 stage. The required information, as well as instruction-length information, is included in at most the first 3 bytes of the instruction. Hence, 3 bytes are passed to the D1 stage from the prefetch buffers. The D1 decoder can then direct the D2 stage to capture the rest of the instruction (displacement and immediate data), which is not involved in the D1 decoding.
- **Decode stage 2:** The D2 stage expands each opcode into control signals for the ALU. It also controls the computation of the more complex addressing modes.
- **Execute:** This stage includes ALU operations, cache access, and register update.
- **Write back:** This stage, if needed, updates registers and status flags modified during the preceding execute stage. If the current instruction updates memory, the computed value is sent to the cache and to the bus-interface write buffers at the same time.

With the use of two decode stages, the pipeline can sustain a throughput of close to one instruction per clock cycle. Complex instructions and conditional branches can slow down this rate.

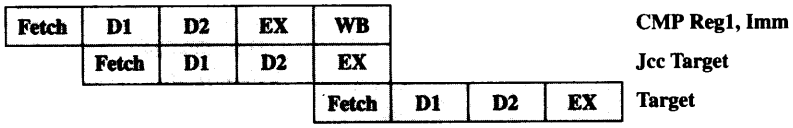
Figure 12.19 shows examples of the operation of the pipeline. Part a shows that there is no delay introduced into the pipeline when a memory access is required. However, as part b shows, there can be a delay for values used to compute memory addresses. That is, if a value is loaded from memory into a register and that register is then used as a base register in the next instruction, the processor will stall for one cycle. In this example, the processor accesses the cache in the EX stage of the first instruction and stores the value retrieved in the register during the WB stage. However, the next instruction needs this register in its D2 stage. When the D2 stage lines up with the WB stage of the previous instruction, bypass signal paths allow the D2 stage to have access to the same data being used by the WB stage for writing, saving one pipeline stage.



(a) No data load delay in the pipeline



(b) Pointer load delay



(c) Branch instruction timing

Figure 12.19 80486 Instruction Pipeline Examples

Figure 12.19c illustrates the timing of a branch instruction, assuming that the branch is taken. The compare instruction updates condition codes in the WB stage, and bypass paths make this available to the EX stage of the jump instruction at the same time. In parallel, the processor runs a speculative fetch cycle to the target of the jump during the EX stage of the jump instruction. If the processor determines a false branch condition, it discards this prefetch and continues execution with the next sequential instruction (already fetched and decoded).

12.5 THE PENTIUM PROCESSOR

An overview of the Pentium 4 processor organization is depicted in Figure 4.13. In this section, we examine some of the details.

Register Organization

The register organization includes the following types of registers (Table 12.2):

- **General:** There are eight 32-bit general-purpose registers (see Figure 12.3c). These may be used for all types of Pentium instructions; they can also hold operands for address calculations. In addition, some of these registers also serve special purposes. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands without having to reference these registers explicitly in the instruction. As a result, a number of instructions can be encoded more compactly.
- **Segment:** The six 16-bit segment registers contain segment selectors, which index into segment tables, as discussed in Chapter 8. The code segment (CS) register references the segment containing the instruction being executed. The

Table 12.2 Pentium Processor Registers

(a) Integer Unit

Type	Number	Length (bits)	Purpose
General	8	32	General-purpose user registers
Segment	6	16	Contain segment selectors
Flags	1	32	Status and control bits
Instruction Pointer	1	32	Instruction pointer

(b) Floating-Point Unit

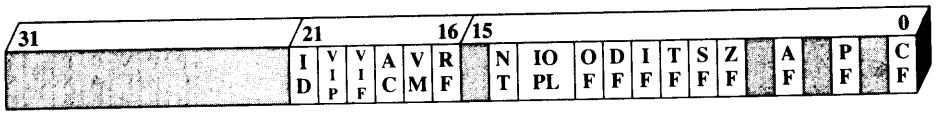
Type	Number	Length (bits)	Purpose
Numeric	8	80	Hold floating-point numbers
Control	1	16	Control bits
Status	1	16	Status bits
Tag Word	1	16	Specifies contents of numeric registers
Instruction Pointer	1	48	Points to instruction interrupted by exception
Data Pointer	1	48	Points to operand interrupted by exception

stack segment (SS) register references the segment containing a user-visible stack. The remaining segment registers (DS, ES, FS, GS) enable the user to reference up to four separate data segments at a time.

- **Flags:** The EFLAGS register contains condition codes and various mode bits.
- **Instruction pointer:** Contains the address of the current instruction.

There are also registers specifically devoted to the floating-point unit:

- **Numeric:** Each register holds an extended-precision 80-bit floating-point number. There are eight registers that function as a stack, with push and pop operations available in the instruction set.
- **Control:** The 16-bit control register contains bits that control the operation of the floating-point unit, including the type of rounding control; single, double, or extended precision; and bits to enable or disable various exception conditions.
- **Status:** The 16-bit status register contains bits that reflect the current state of the floating-point unit, including a 3-bit pointer to the top of the stack; condition codes reporting the outcome of the last operation; and exception flags.
- **Tag word:** This 16-bit register contains a 2-bit tag for each floating-point numeric register, which indicates the nature of the contents of the corresponding register. The four possible values are valid, zero, special (NaN, infinity, denormalized), and empty. These tags enable programs to check the contents of a numeric register without performing complex decoding of the actual data in the register. For example, when a context switch is made, the processor need not save any floating-point registers that are empty.



ID = Identification flag
VIP = Virtual interrupt pending
VIF = Virtual interrupt flag
AC = Alignment check
VM = Virtual 8086 mode
RF = Resume flag
NT = Nested task flag
IOPL = I/O privilege level
OF = Overflow flag
DF = Direction flag
IF = Interrupt enable flag
TF = Trap flag
SF = Sign flag
ZF = Zero flag
AF = Auxiliary carry flag
PF = Parity flag
CF = Carry flag

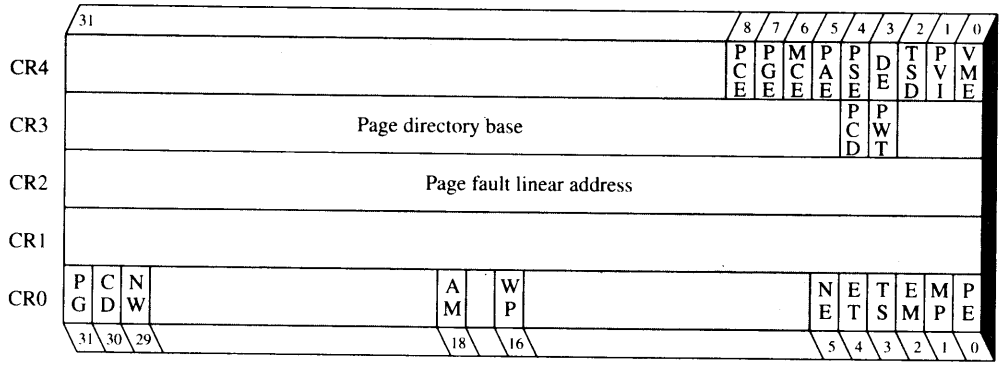
Figure 12.20 Pentium II EFLAGS Register

The use of most of the aforementioned registers is easily understood. Let us elaborate briefly on several of the registers.

EFLAGS Register The EFLAGS register (Figure 12.20) indicates the condition of the processor and helps to control its operation. It includes the six condition codes defined in Table 10.9 (carry, parity, auxiliary, zero, sign, overflow), which report the results of an integer operation. In addition, there are bits in the register that may be referred to as control bits:

- **Trap flag (TF):** When set, causes an interrupt after the execution of each instruction. This is used for debugging.
- **Interrupt enable flag (IF):** When set, the processor will recognize external interrupts.
- **Direction flag (DF):** Determines whether string processing instructions increment or decrement the 16-bit half-registers SI and DI (for 16-bit operations) or the 32-bit registers ESI and EDI (for 32-bit operations).
- **I/O privilege flag (IOPL):** When set, causes the processor to generate an exception on all accesses to I/O devices during protected-mode operation.
- **Resume flag (RF):** Allows the programmer to disable debug exceptions so that the instruction can be restarted after a debug exception without immediately causing another debug exception.
- **Alignment check (AC):** Activates if a word or doubleword is addressed on a nonword or nondoubleword boundary.
- **Identification flag (ID):** If this bit can be set and cleared, then this processor supports the processorID instruction. This instruction provides information about the vendor, family, and model.

In addition, there are 4 bits that relate to operating mode. The nested task (NT) flag indicates that the current task is nested within another task in protected-mode operation. The virtual mode (VM) bit allows the programmer to enable or disable virtual 8086 mode, which determines whether the processor runs as an 8086 machine. The virtual interrupt flag (VIF) and virtual interrupt pending (VIP) flag are used in a multitasking environment.



PCE = Performance counter enable
PGE = Page global enable
MCE = Machine check enable
PAE = Physical address extension
PSE = Page size extensions
DE = Debug extensions
TSD = Time stamp disable
PVI = Protected mode virtual interrupt
VME = Virtual 8086 mode extensions
PCD = Page-level cache disable
PWT = Page-level writes transparent

PG = Paging
CD = Cache disable
NW = Not write through
AM = Alignment mask
WP = Write protect
NE = Numeric error
ET = Extension type
TS = Task switched
EM = Emulation
MP = Monitor coprocessor
PE = Protection enable

Figure 12.21 Pentium II Control Registers

Control Registers The Pentium employs four 32-bit control registers (register CR1 is unused) to control various aspects of processor operation (Figure 12.21). The CR0 register contains system control flags, which control modes or indicate states that apply generally to the processor rather than to the execution of an individual task. The flags are as follows:

- **Protection enable (PE):** Enable/disable protected mode of operation.
- **Monitor coprocessor (MP):** Only of interest when running programs from earlier machines on the Pentium; it relates to the presence of an arithmetic coprocessor.
- **Emulation (EM):** Set when the processor does not have a floating-point unit, and causes an interrupt when an attempt is made to execute floating-point instructions.
- **Task switched (TS):** Indicates that the processor has switched tasks.
- **Extension type (ET):** Not used on the Pentium; used to indicate support of math coprocessor instructions on earlier machines.
- **Numeric error (NE):** Enables the standard mechanism for reporting floating-point errors on external bus lines.
- **Write protect (WP):** When this bit is clear, read-only user-level pages can be written by a supervisor process. This feature is useful for supporting process creation in some operating systems.
- **Alignment mask (AM):** Enables/disables alignment checking.

- **Not Write through (NW):** Selects mode of operation of the data cache. When this bit is set, the data cache is inhibited from cache write-through operations.
- **Cache disable (CD):** Enables/disables the internal cache fill mechanism.
- **Paging (PG):** Enables/disables paging.

When paging is enabled, the CR2 and CR3 registers are valid. The CR2 register holds the 32-bit linear address of the last page accessed before a page fault interrupt. The leftmost 20 bits of CR3 hold the 20 most significant bits of the base address of the page directory; the remainder of the address contains zeros. Two bits of CR3 are used to drive pins that control the operation of an external cache. The page-level cache disable (PCD) enables or disables the external cache, and the page-level writes transparent (PWT) bit controls write through in the external cache.

Nine additional control bits are defined in CR4:

- **Virtual-8086 mode extension (VME):** Enables support for the virtual interrupt flag in virtual-8086 mode.
- **Protected-mode virtual Interrupts (PVI):** Enables support for the virtual interrupt flag in protected mode.
- **Time stamp disable (TSD):** Disables the read from time stamp counter (RDTSC) instruction, which is used for debugging purposes.
- **Debugging extensions (DE):** Enables I/O breakpoints; this allows the processor to interrupt on I/O reads and writes.
- **Page size extensions (PSE):** Enables the use of 4-Mbyte pages when set in the Pentium or 2M-byte pages when set in the Pentium Pro and Pentium.
- **Physical address extension (PAE):** Enables address lines A35 through A32 whenever a special new addressing mode, controlled by the PSE, is enabled for the Pentium Pro and subsequent Pentium architectures (Pentium II through Pentium 4).
- **Machine check enable (MCE):** Enables the machine check interrupt, which occurs when a data parity error occurs during a read bus cycle or when a bus cycle is not successfully completed.
- **Page global enable (PGE):** Enables the use of global pages. When PGE = 1 and a task switch is performed, all of the TLB entries are flushed with the exception of those marked global.
- **Performance counter enable (PCE):** Enables the execution of the RDPMC (read performance counter) instruction at any privilege level. Two performance counters are used to measure the duration of a specific event type and the number of occurrences of a specific event type.

MMX Registers Recall from Section 10.3 that the Pentium MMX capability makes use of several 64-bit data types. The MMX instructions make use of 3-bit register address fields, so that eight MMX registers are supported. In fact, the processor does not include specific MMX registers. Rather, the processor uses an aliasing technique (Figure 12.22). The existing floating-point registers are used to store MMX values. Specifically, the low-order 64 bits (mantissa) of each floating-point register are used to form the eight MMX registers. Thus, the existing Pentium

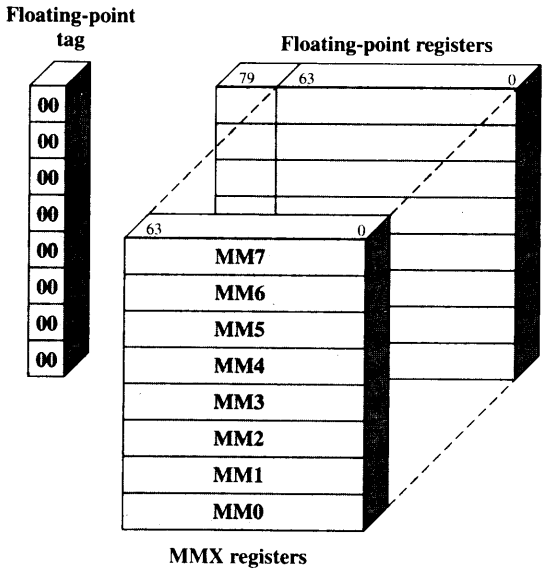


Figure 12.22 Mapping of MMX Registers to Floating-Point Registers

architecture is easily extended to support the MMX capability. Some key characteristics of the MMX use of these registers:

- Recall that the floating-point registers are treated as a stack for floating-point operations. For MMX operations, these same registers are accessed directly.
- The first time that an MMX instruction is executed after any floating-point operations, the FP tag word is marked valid. This reflects the change from stack operation to direct register addressing.
- The EMMS (Empty MMX State) instruction sets bits of the FP tag word to indicate that all registers are empty. It is important that the programmer insert this instruction at the end of an MMX code block so that subsequent floating-point operations function properly.
- When a value is written to an MMX register, bits [79:64] of the corresponding FP register (sign and exponent bits) are set to all ones. This sets the value in the FP register to NaN (not a number) or infinity when viewed as a floating-point value. This ensures that an MMX data value will not look like a valid floating-point value.

Interrupt Processing

Interrupt processing within a processor is a facility provided to support the operating system. It allows an application program to be suspended, in order that a variety of interrupt conditions can be serviced and later resumed.

Interrupts and Exceptions Two classes of events cause the Pentium to suspend execution of the current instruction stream and respond to the event: interrupts and exceptions. In both cases, the processor saves the context of the current process and

transfers to a predefined routine to service the condition. An *interrupt* is generated by a signal from hardware, and it may occur at random times during the execution of a program. An *exception* is generated from software, and it is provoked by the execution of an instruction. There are two sources of interrupts and two sources of exceptions:

1. Interrupts
   - **Maskable interrupts:** Received on the processor's INTR pin. The processor does not recognize a maskable interrupt unless the interrupt enable flag (IF) is set.
   - **Nonmaskable interrupts:** Received on the processor's NMI pin. Recognition of such interrupts cannot be prevented.
2. Exceptions
   - **Processor-detected exceptions:** Results when the processor encounters an error while attempting to execute an instruction.
   - **Programmed exceptions:** These are instructions that generate an exception (e.g., INTO, INT3, INT, and BOUND).

Interrupt Vector Table Interrupt processing on the Pentium uses the interrupt vector table. Every type of interrupt is assigned a number, and this number is used to index into the interrupt vector table. This table contains 256 32-bit interrupt vectors, which is the address (segment and offset) of the interrupt service routine for that interrupt number.

Table 12.3 shows the assignment of numbers in the interrupt vector table; shaded entries represent interrupts, while nonshaded entries are exceptions. The NMI hardware interrupt is type 2. INTR hardware interrupts are assigned numbers in the range of 32 to 255; when an INTR interrupt is generated, it must be accompanied on the bus with the interrupt vector number for this interrupt. The remaining vector numbers are used for exceptions.

If more than one exception or interrupt is pending, the processor services them in a predictable order. The location of vector numbers within the table does not reflect priority. Instead, priority among exceptions and interrupts is organized into five classes. In descending order of priority, these are

- **Class 1:** Traps on the previous instruction (vector number 1)
- **Class 2:** External interrupts (2, 32–255)
- **Class 3:** Faults from fetching next instruction (3, 14)
- **Class 4:** Faults from decoding the next instruction (6, 7)
- **Class 5:** Faults on executing an instruction (0, 4, 5, 8, 10–14, 16, 17)

Interrupt Handling Just as with a transfer of execution using a CALL instruction, a transfer to an interrupt-handling routine uses the system stack to store the processor state. When an interrupt occurs and is recognized by the processor, a sequence of events takes place:

1. If the transfer involves a change of privilege level, then the current stack segment register and the current extended stack pointer (ESP) register are pushed onto the stack.

Table 12.3 Pentium Exception and Interrupt Vector Table

Vector Number	Description
0	Divide error; division overflow or division by zero
1	Debug exception; includes various faults and traps related to debugging
2	**NMI pin interrupt; signal on NMI pin**
3	Breakpoint; caused by INT 3 instruction, which is a 1-byte instruction useful for debugging
4	INTO-detected overflow; occurs when the processor executes INTO with the OF flag set
5	BOUND range exceeded; the BOUND instruction compares a register with boundaries stored in memory and generates an interrupt if the contents of the register is out of bounds.
6	Undefined opcode
7	Device not available; attempt to use ESC or WAIT instruction fails due to lack of external device
8	Double fault; two interrupts occur during the same instruction and cannot be handled serially
9	Reserved
10	Invalid task state segment; segment describing a requested task is not initialized or not valid
11	Segment not present; required segment not present
12	Stack fault; limit of stack segment exceeded or stack segment not present
13	General protection; protection violation that does not cause another exception (e.g., writing to a read-only segment)
14	Page fault
15	Reserved
16	Floating-point-error; generated by a floating-point arithmetic instruction
17	Alignment check; access to a word stored at an odd byte address or a doubleword stored at an address not a multiple of 4
18	Machine check; model specific
19–31	Reserved
32–255	**User interrupt vectors; provided when INTR signal is activated**

Unshaded: exceptions
Shaded: interrupts

2. The current value of the EFLAGS register is pushed onto the stack.
3. Both the interrupt (IF) and trap (TF) flags are cleared. This disables INTR interrupts and the trap or single-step feature.
4. The current code segment (CS) pointer and the current instruction pointer (IP or EIP) are pushed onto the stack.
5. If the interrupt is accompanied by an error code, then the error code is pushed onto the stack.
6. The interrupt vector contents are fetched and loaded into the CS and IP or EIP registers. Execution continues from the interrupt service routine.

To return from an interrupt, the interrupt service routine executes an IRET instruction. This causes all of the values saved on the stack to be restored; execution resumes from the point of the interrupt.

12.6 THE POWERPC PROCESSOR

An overview of the PowerPC processor organization is depicted in Figure 4.14. In this section, we examine some of the details of the 64-bit implementation.

Register Organization

Figure 12.23 depicts the user-visible registers for the PowerPC. The fixed-point unit includes the following:

- **General:** There are thirty-two 64-bit general-purpose registers. These may be used to load, store, and manipulate data operands and may also be used for register indirect addressing. Register 0 is treated somewhat differently. For

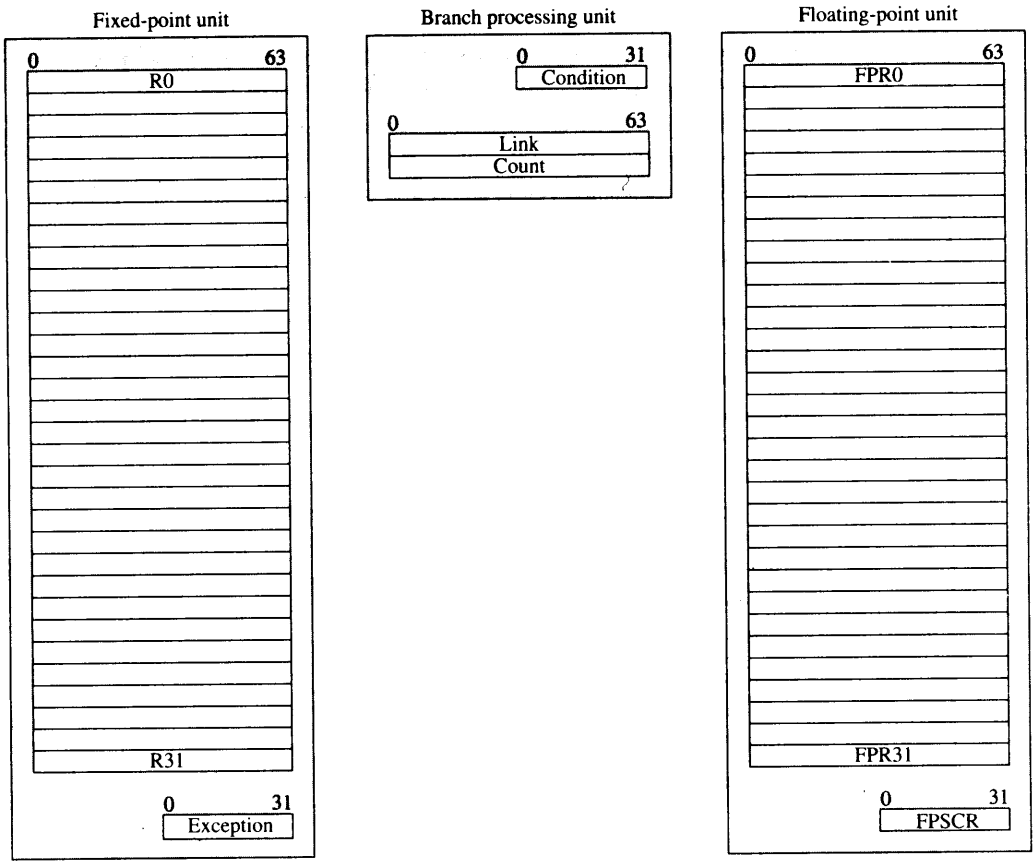


Figure 12.23 Power PC User-Visible Registers

load and store operations and several of the add instructions, register 0 is treated as having a constant value zero regardless of its actual contents.

- **Exception register (XER):** Includes 3 bits that report exceptions in integer arithmetic operations. This register also includes a byte count field that is used as an operand for some string instructions (Figure 12.23a).

The floating-point unit contains additional user-visible registers:

- **General:** There are thirty-two 64-bit general-purpose registers, used for all floating-point operations.
- **Floating-point status and control register (FPSCR):** This 32-bit register contains bits that control the operation of the floating-point unit and bits that record the status resulting from floating-point operations (Table 12.4).

Table 12.4 PowerPC Floating-Point Status and Control Register

Bit	Definition
0	Exception summary. Set if any exception occurs; remains set until reset by software.
1	Enabled exception summary. Set if any enabled exception has occurred.
2	Invalid operation exception summary. Set if an invalid operation exception has occurred.
3	Overflow exception. Magnitude of result exceeds what can be represented.
4	Underflow exception. Result is to small to be normalized.
5	Zero divide exception. Divisor is zero and dividend is finite nonzero.
6	Inexact exception. Rounded result differs from intermediate result or an overflow occurs with overflow exception disabled.
7:12	Invalid operation exception. 7: signaling NaN; 8: $(\infty - \infty)$; 9: $(\infty \div \infty)$; 10: $(0 \div 0)$; 11: $(\infty \times 0)$; 12: comparison involving NaN.
13	Fraction rounded. Rounding of the result incremented the fraction.
14	Fraction inexact. Rounded result changes fraction or an overflow occurs with overflow exception disabled.
15:19	Result flags. Five-bit code specifies less than, greater than, equal, unordered, quiet NaN, $\pm\infty$, $\pm$normalized, $\pm$denormalized, ± 0.
20	Reserved
21:23	Invalid operation exception. 21: software request; 22: square root of a negative number; 23: Integer conversion involving a large number, an infinity, or a NaN.
24	Invalid operation exception enable
25	Overflow exception enable
26	Underflow exception enable
27	Zero divide exception enable
28	Inexact exception enable
29	Non-IEEE mode
30:31	Rounding control. Two-bit code specifies to nearest, toward 0, toward $+\infty$, toward $-\infty$.

Unshaded: status bits
Shaded: control bits