Part Five The Central Processing Unit

*Chapter **16** Processor Structure and Function*

Learning Objectives

After studying this chapter, you should be able to:

- Distinguish between **user-visible** and **control/status registers**, and discuss the purposes of registers in each category.
- Summarize the **instruction cycle**.
- Discuss the principle behind **instruction pipelining** and how it works in practice.

- Compare and contrast the various forms of pipeline hazards.
- Present an overview of the x86 processor structure.
- Present an overview of the ARM processor structure.

*This chapter discusses aspects of the processor not covered in Part Four and sets the stage for the discussion of RISC and superscalar architecture in **Chapters 17** and **18**.*

*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 aspects of the x86 and ARM organizations.*

16.1 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 16.1 is a simplified view of the internal structure of a processor. 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*. 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.

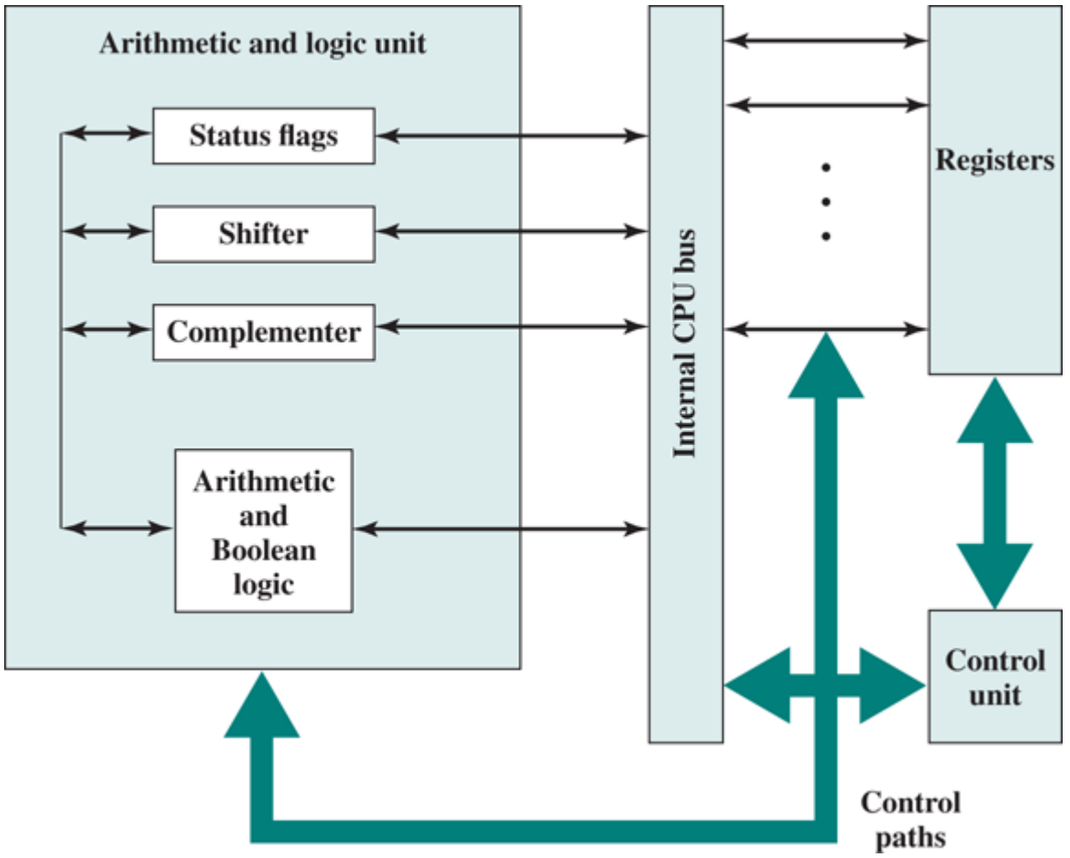


Figure 16.1 Internal Structure of the CPU

16.2 Register Organization

As we discussed in **Chapter 4**, aA 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., x86), 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 9.3**), 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 17**.

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 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 16.1**, based on [DERO87], lists key advantages and disadvantages of condition codes.

Table 16.1 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. 2. Conditional instructions such as BRANCH are simplified relative to composite instructions, such as TEST and BRANCH. 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. 4. Condition codes can be saved on the stack during subroutine calls	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. Condition codes are irregular; they are typically not part of the main data path, so they require extra hardware connections. 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.

along with other register information.	

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 16.2a** and **b** depict the register organization of each; purely internal registers, such as a memory address register, are not shown.

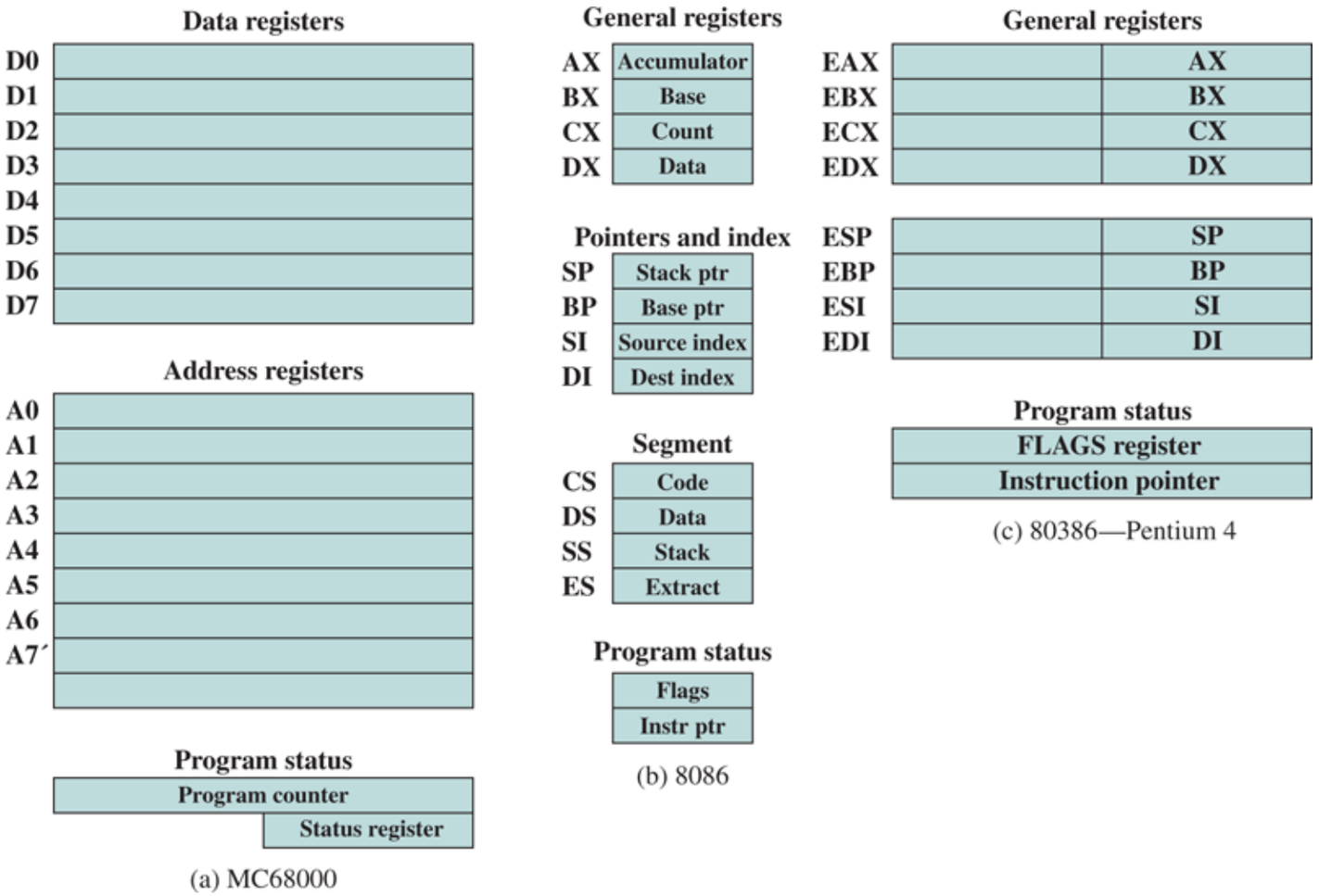


Figure 16.2 Example Microprocessor Register Organizations

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 16.2c**. 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.

[1] Because the MC68000 already uses 32-bit registers, the MC68020 [MACD84], which is a full 32-bit architecture, uses the same register organization.

16.3 Instruction Cycle

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

- **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 stage, known as the indirect cycle.

The Indirect Cycle

We have seen in **Chapter 14** 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 or more instruction stages. The result is shown in **Figure 16.3**. 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.

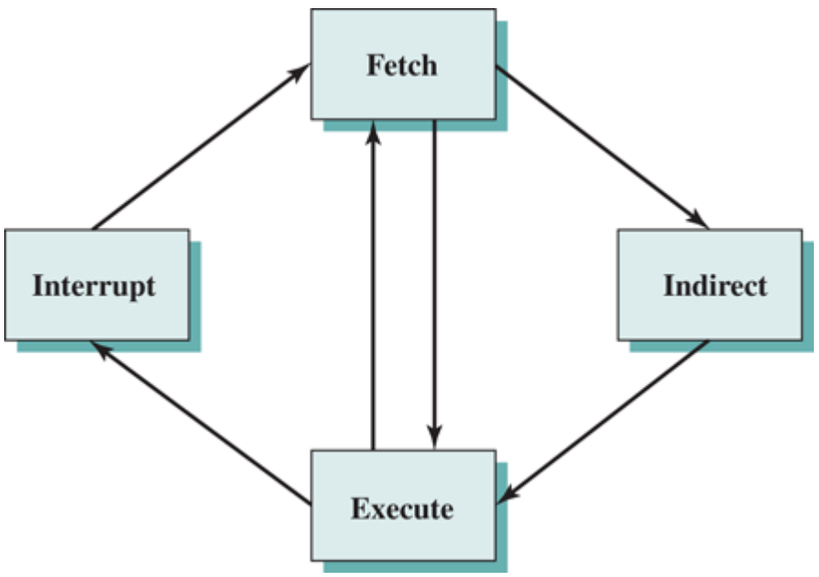


Figure 16.3 The Instruction Cycle

Another way to view this process is shown in **Figure 16.4**, 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.

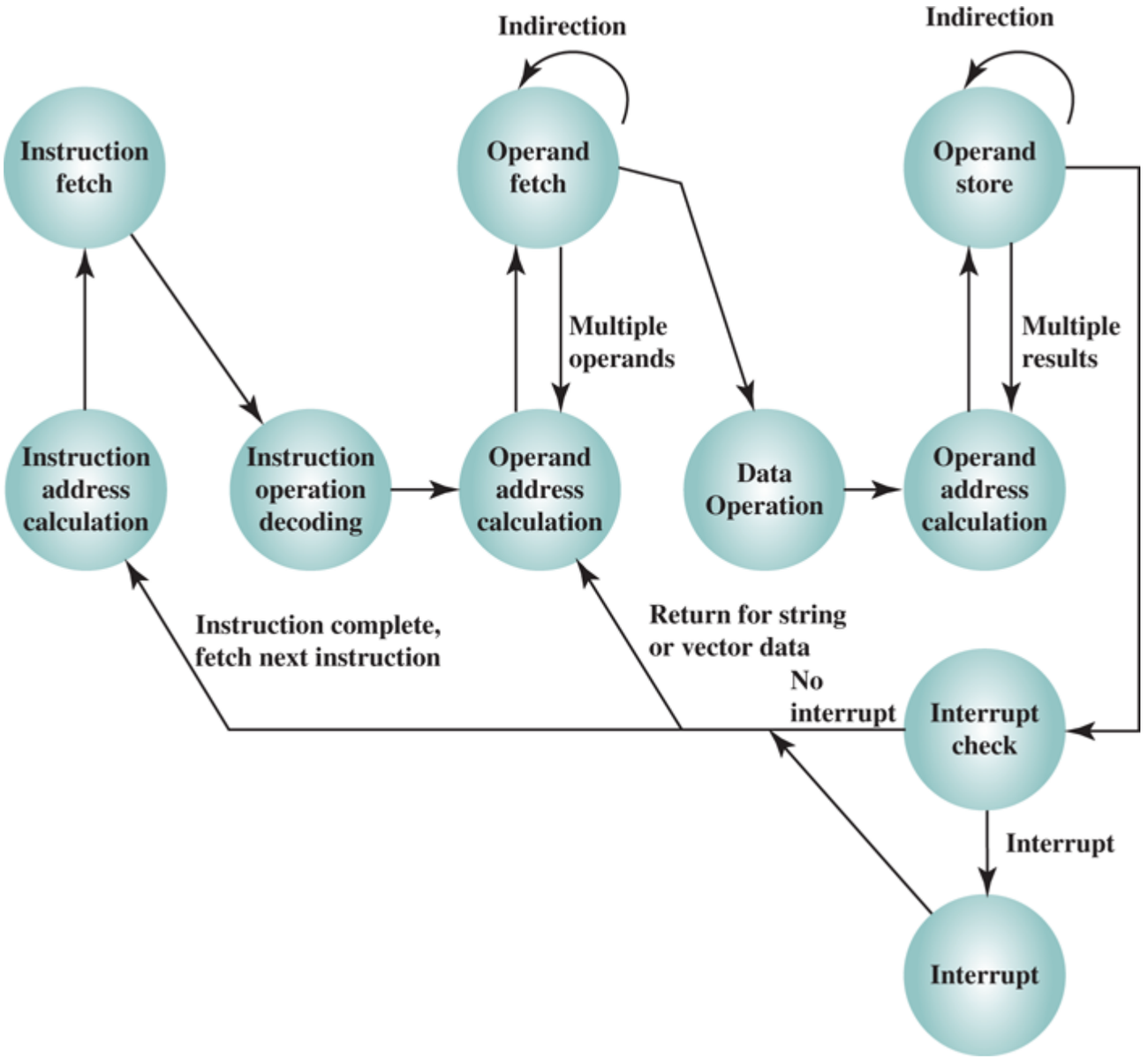


Figure 16.4 Instruction Cycle State Diagram

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 a processor that employs a memory address register (MAR), a memory buffer register (MBR), a program counter (PC), and an instruction register (IR).

During the *fetch cycle,* an instruction is read from memory. **Figure 16.5** 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.

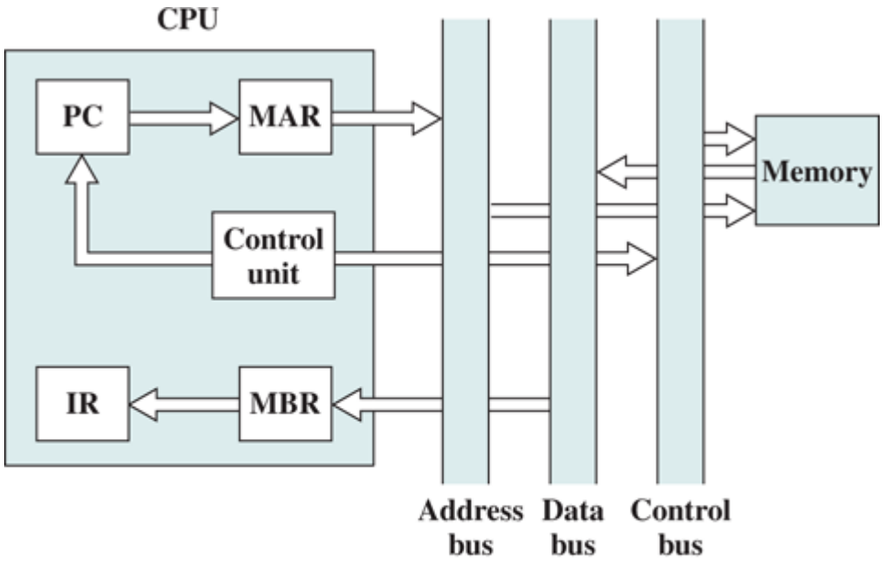


Figure 16.5 Data Flow, Fetch Cycle

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 16.6**, this is a simple cycle. The right-most *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.

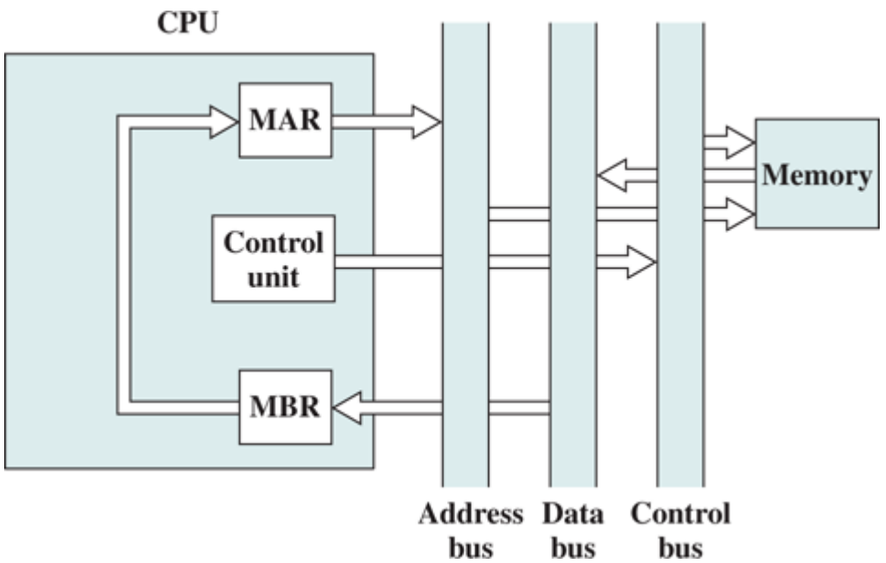


Figure 16.6 Data Flow, Indirect Cycle

The fetch and indirect cycles are simple and predictable. The *execute cycle* takes many forms, depending 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 16.7**). 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.

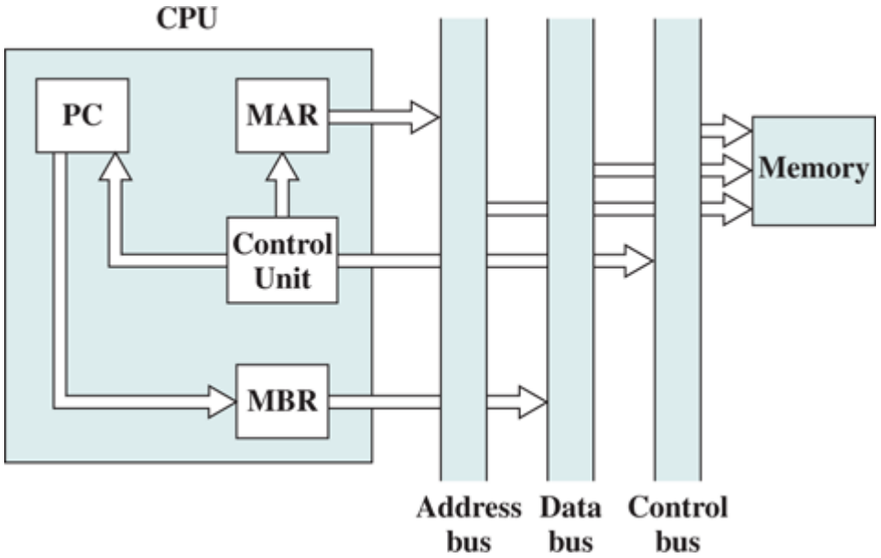


Figure 16.7 Data Flow, Interrupt Cycle

16.4 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 16.4** for example breaks the 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 16.8a** 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.* Note that this approach, which involves instruction buffering, requires more registers.

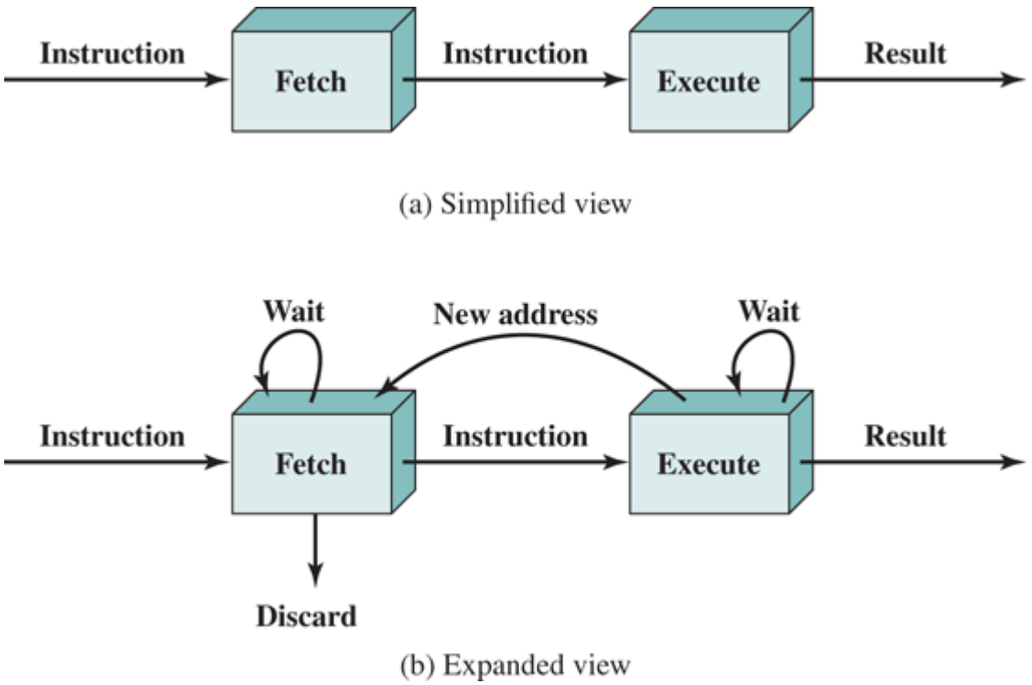


Figure 16.8 Two-Stage Instruction Pipeline

In general, pipelining requires fast registers, called **latches**, that store intermediate values between stages. **Figure 16.9** illustrates this in simplified form for a three-stage pipeline. The latches serve to decouple the stages from each other. Information flows between adjacent stages under the control of a common clock applied to all the latches simultaneously. As each clock cycle ends, the latches gate in their inputs and forward them into the next stage, where the required operation is performed. This simplified picture omits several details. Each stage may consist of multiple execution units that cooperate in performing the required operations. In addition, the latches may be extended with multiplexers that allow the input to a latch to come from a subsequent stage (feed back) or from a stage prior to the just preceding stage (feed forward).

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 16.8b**), we will see that this doubling of execution rate is unlikely for two reasons:

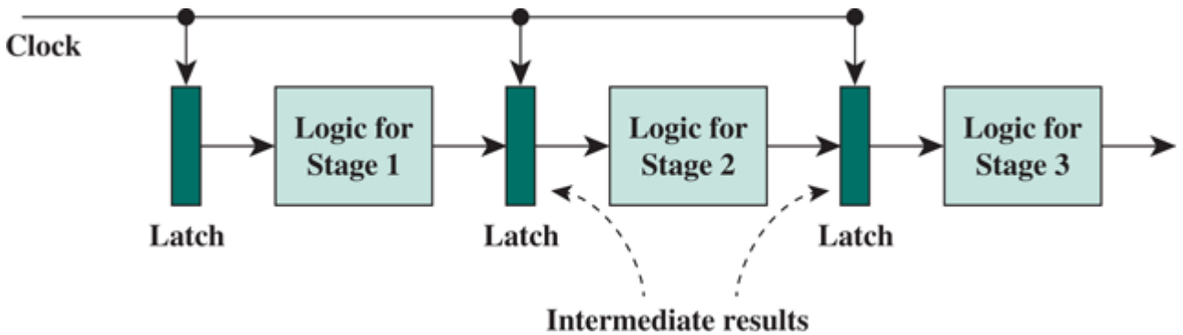


Figure 16.9 Simplified Pipeline Architecture

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 16.10** shows that a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.

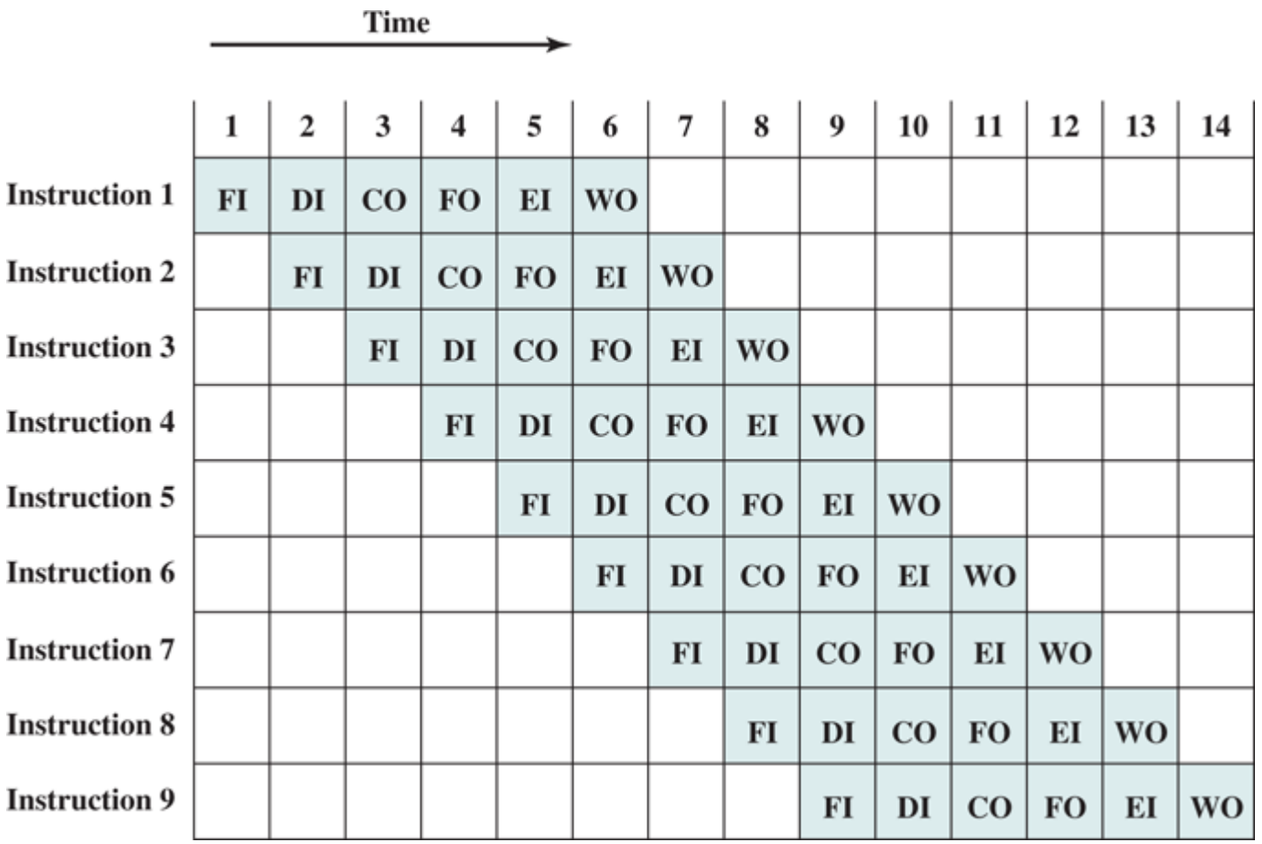


Figure 16.10 Timing Diagram for Instruction Pipeline Operation

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 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 16.11** illustrates the effects of the conditional branch, using the same program as **Figure 16.10**. 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 16.10**, the branch is not taken, and we get the full performance benefit of the enhancement. In **Figure 16.11**, 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 16.12** indicates the logic needed for pipelining to account for branches and interrupts.

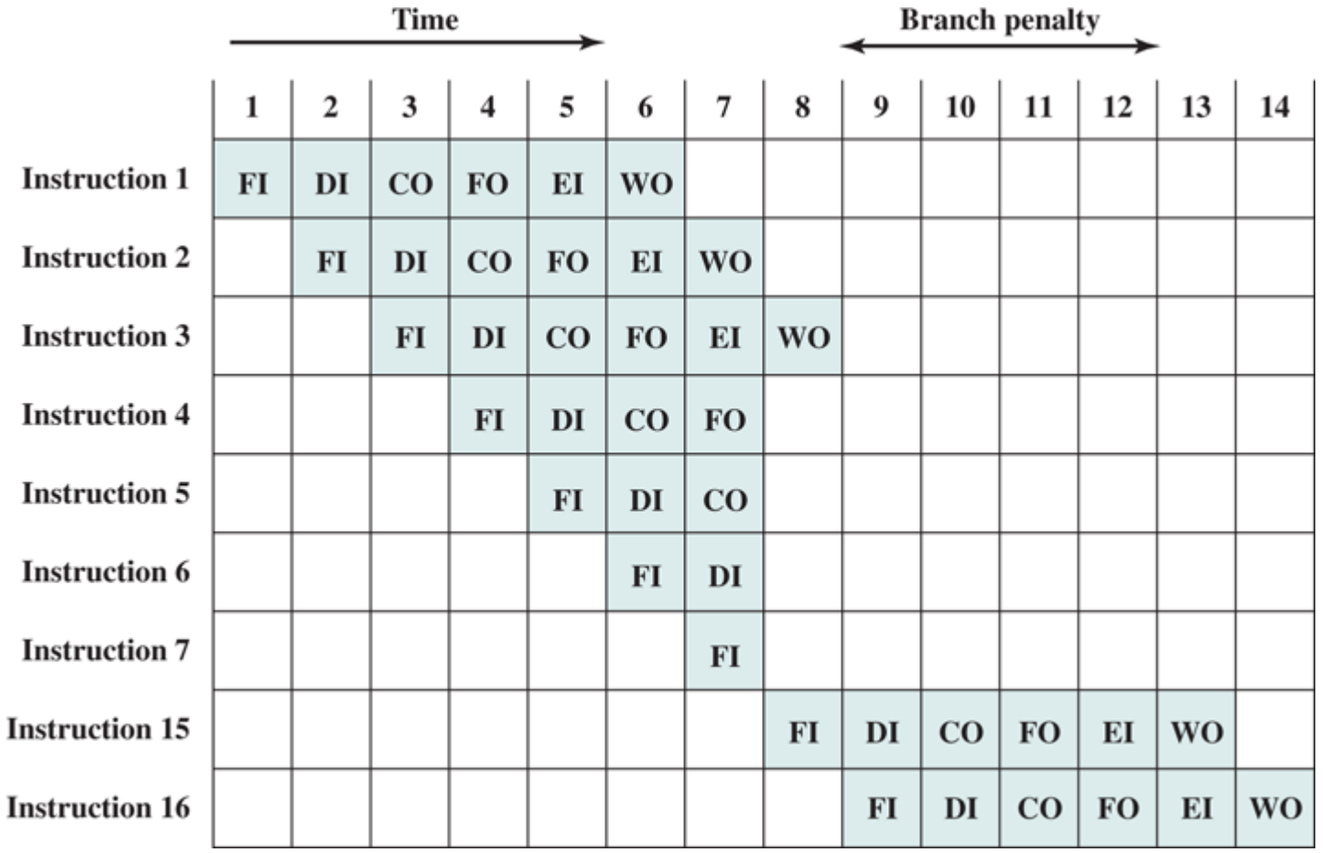


Figure 16.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

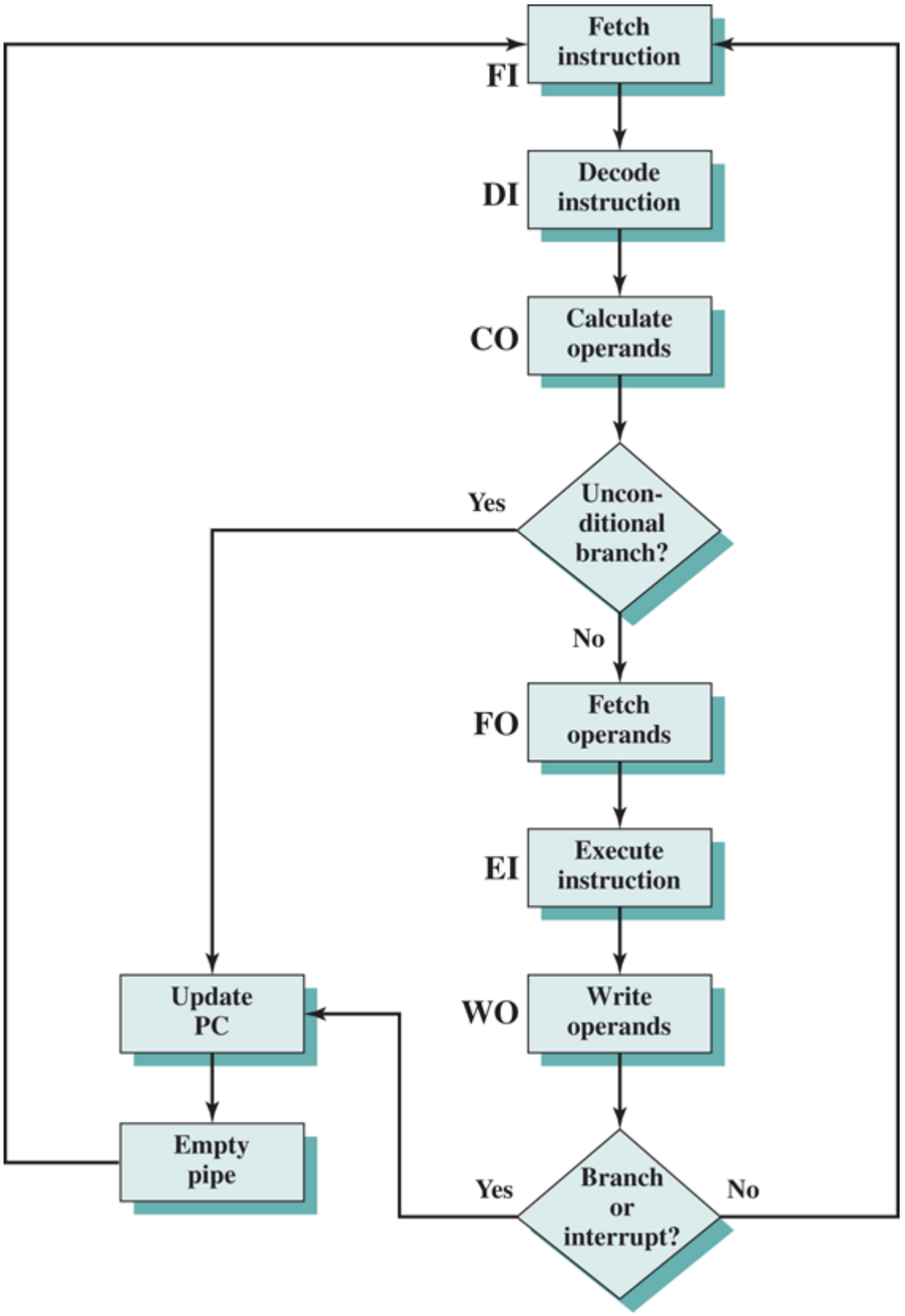
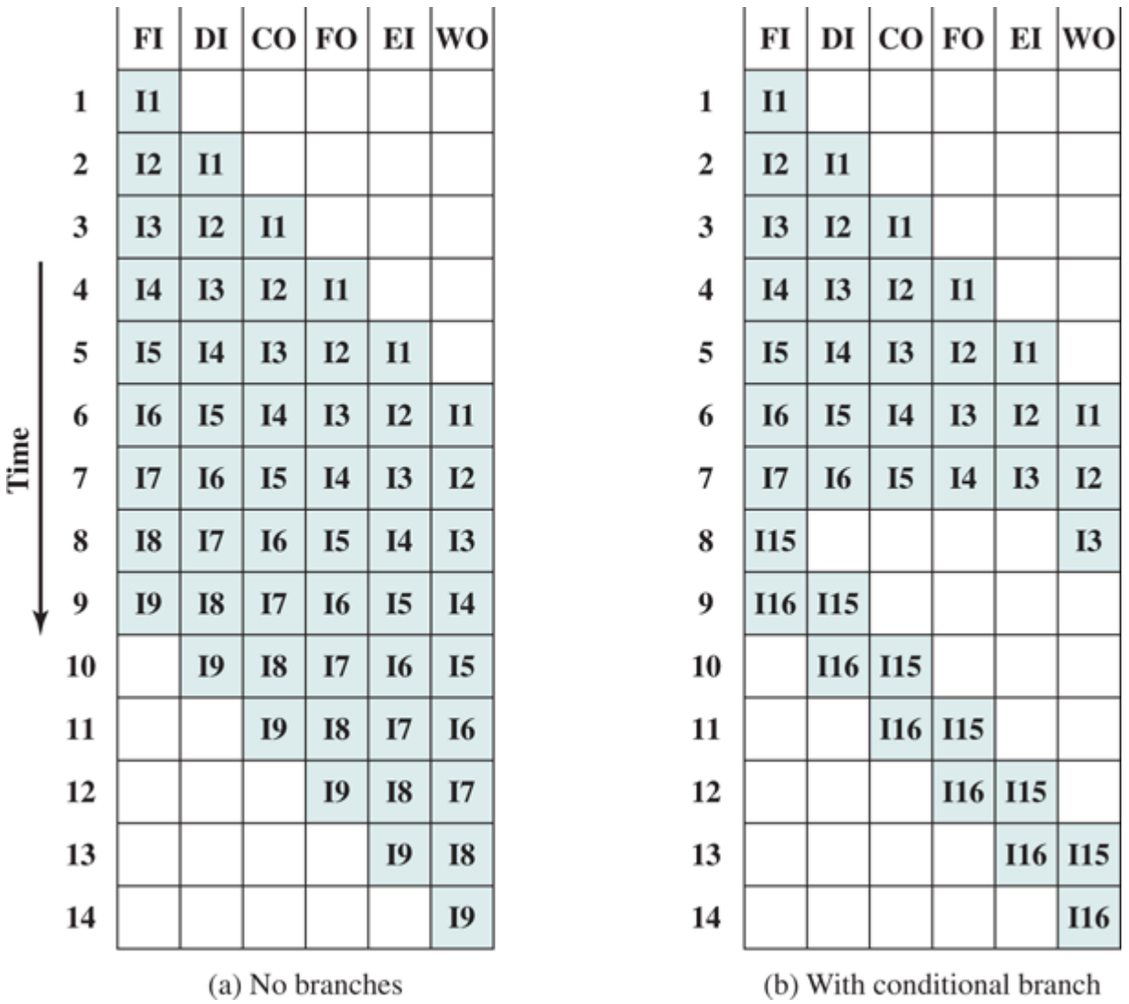


Figure 16.12 Six-Stage CPU Instruction Pipeline

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.

To clarify pipeline operation, it might be useful to look at an alternative depiction. **Figures 16.10** and **16.11** show the progression of time horizontally across the figures, with each row showing the

progress of an individual instruction. **Figure 16.13** shows the 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 16.13a** (which corresponds to **Figure 16.10**), 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 16.13b**, (which corresponds to **Figure 16.11**), 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 are flushed from the pipeline, so that at time 8, only two instructions are in the pipeline, I3 and I15.



(a) No branches (b) With conditional branch

Figure 16.13 An Alternative Pipeline Depiction

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.

Another consideration is latching delay: It takes time for pipeline buffers to operate, and this adds to instruction cycle time.

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 16.10** and **16.11** represents one cycle time. The cycle time can be determined as

$$\tau = \max_i [\tau_i] + d = \tau_m + d \quad 1 \le i \le k$$

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

$$T_{k,n} = [k + (n - 1)]\tau \tag{16.1}$$

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 16.10**. The ninth instruction completes at time cycle 14:

[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.

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

Now consider a 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

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)} \tag{16.2}$$

Figure 16.14a 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 $(n \to \infty)$, we have a *k*-fold speedup. **Figure**

16.14b 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 stages are countered by increases in cost, delays between stages, and the fact that branches will be encountered requiring the flushing of the pipeline.

[3] Note that the *x*-axis is logarithmic in Figure 16.14a and linear in Figure 16.14b.

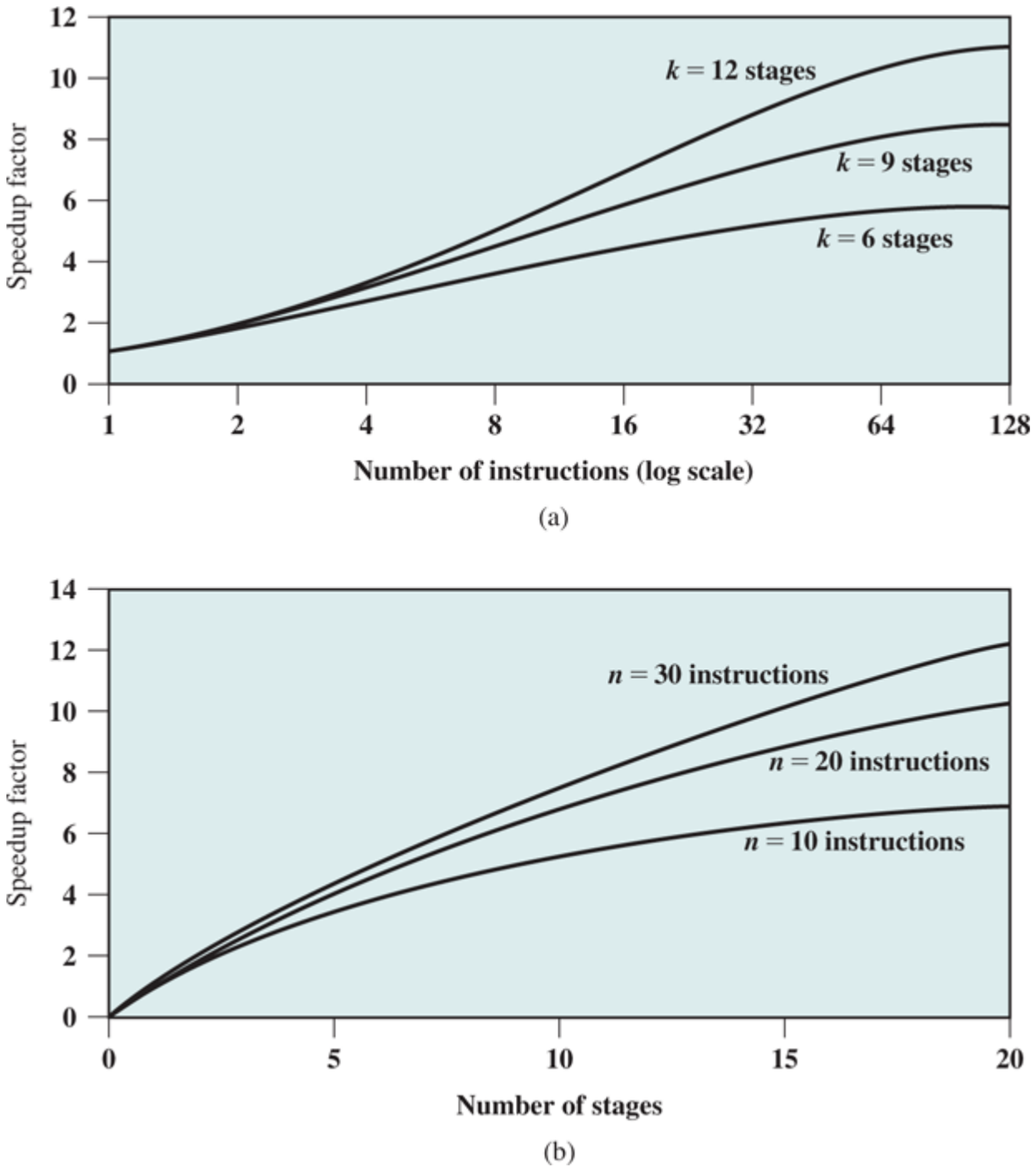


Figure 16.14 Speedup Factors with Instruction Pipelining

Pipeline Hazards

In the previous subsection, we mentioned some of the situations that can result in less than optimal

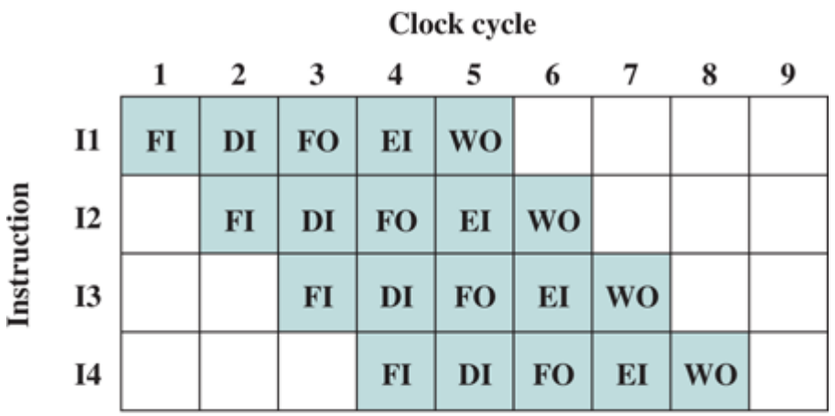
pipeline performance. In this subsection, we examine this issue in a more systematic way. **Chapter 18** revisits this issue in more detail, after we have introduced the complexities found in superscalar pipeline organizations.

A **pipeline hazard** occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution. Such a pipeline stall is also referred to as a *pipeline bubble*. There are three types of hazards: resource, data, and control.

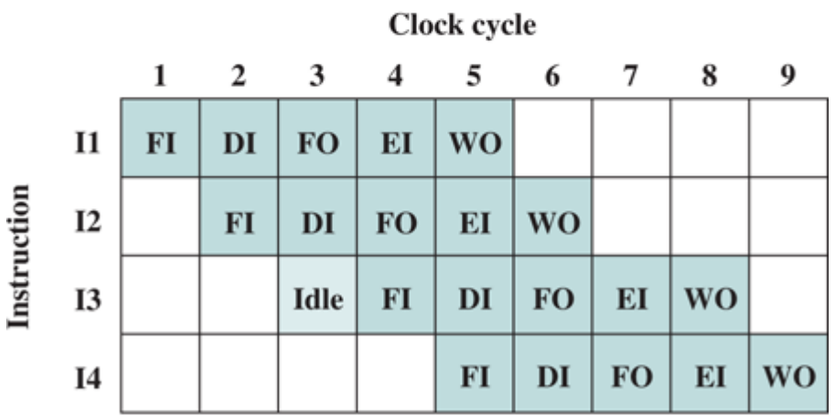
RESOURCE HAZARDS

A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource. The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline. A resource hazard is sometime referred to as a *structural hazard*.

Let us consider a simple example of a resource hazard. Assume a simplified five-stage pipeline, in which each stage takes one clock cycle. **Figure 16.15a** shows the ideal case, in which a new instruction enters the pipeline each clock cycle. Now assume that main memory has a single port and that all instruction fetches and data reads and writes must be performed one at a time. Further, ignore the cache. In this case, an operand read to or write from memory cannot be performed in parallel with an instruction fetch. This is illustrated in **Figure 16.15b**, which assumes that the source operand for instruction I1 is in memory, rather than a register. Therefore, the fetch instruction stage of the pipeline must idle for one cycle before beginning the instruction fetch for instruction I3. The figure assumes that all other operands are in registers.



(a) Five-stage pipeline, ideal case



(b) I1 source operand in memory

Figure 16.15 Example of Resource Hazard

Another example of a resource conflict is a situation in which multiple instructions are ready to enter the execute instruction phase and there is a single ALU. One solutions to such resource hazards is to increase available resources, such as having multiple ports into main memory and multiple ALU units.

Aleksandr Lukin/123RF

Reservation Table Analyzer

One approach to analyzing resource conflicts and aiding in the design of pipelines is the reservation table. We examine reservation tables in Appendix I.

DATA HAZARDS

A data hazard occurs when there is a conflict in the access of an operand location. In general terms, we can state the hazard in this form: Two instructions in a program are to be executed in sequence and both access a particular memory or register operand. If the two instructions are executed in strict sequence, no problem occurs. However, if the instructions are executed in a pipeline, then it is possible for the operand value to be updated in such a way as to produce a different result than would occur with strict sequential execution. In other words, the program produces an incorrect result because of the use of pipelining.

As an example, consider the following x86 machine instruction sequence:

```
ADD EAX, EBX /* EAX = EAX + EBX
SUB ECX, EAX /* ECX = ECX - EAX
```

The first instruction adds the contents of the 32-bit registers EAX and EBX, and stores the result in EAX. The second instruction subtracts the contents of EAX from ECX and stores the result in ECX. **Figure 16.16** shows the pipeline behavior. The ADD instruction does not update register EAX until the end of stage 5, which occurs at clock cycle 5. But the SUB instruction needs that value at the beginning of its stage 2, which occurs at clock cycle 4. To maintain correct operation, the pipeline must stall for two clocks cycles. Thus, in the absence of special hardware and specific avoidance algorithms, such a data hazard results in inefficient pipeline usage.

	Clock cycle									
	1	2	3	4	5	6	7	8	9	10
ADD EAX, EBX	FI	DI	FO	EI	WO					
SUB ECX, EAX		FI	DI	Idle		FO	EI	WO		
I3			FI			DI	FO	EI	WO	
I4						FI	DI	FO	EI	WO

Figure 16.16 Example of Data Hazard

There are three types of data hazards:

- **Read after write (RAW), or true dependency:** An instruction modifies a register or memory location, and a succeeding instruction reads the data in that memory or register location. A hazard occurs if the read takes place before the write operation is complete. This type of hazard is referred to as a true dependency because it is a real data dependency that is not just due to a shortage of registers. It can occur whether or not the first instruction stalls, and cannot be avoided by reassigning or renaming registers.
- **Write after read (WAR), or antidependency:** An instruction reads a register or memory location and a succeeding instruction writes to the location. A hazard occurs if the write operation completes before the read operation takes place.
- **Write after write (WAW), or output dependency:** Two instructions both write to the same location. A hazard occurs if the write operations take place in the reverse order of the intended sequence.

The example of **Figure 16.16** is a RAW hazard. The other two hazards are best discussed in the context of superscalar organization, discussed in **Chapter 18**.

CONTROL HAZARDS

A control hazard, also known as a *branch hazard,* occurs when the pipeline makes the wrong decision on a branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded. We discuss approaches to dealing with control hazards next.

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 16.17 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.

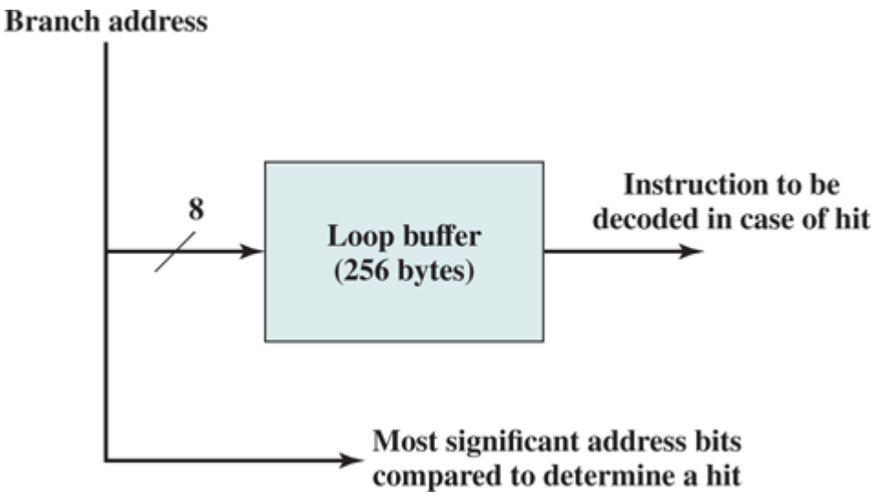


Figure 16.17 Loop Buffer

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 DBcc (decrement and branch on condition) instruction (see Problem 16.14). A three-word buffer is maintained, and the processor executes these instructions repeatedly until the loop condition is satisfied.

Aleksandr Lukin/123RF

Branch Prediction Simulator

Branch Target Buffer

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 predict-never-taken approach is the most popular of all the branch prediction methods.

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 16.18** shows a typical approach (see Problem 16.13 for other 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.

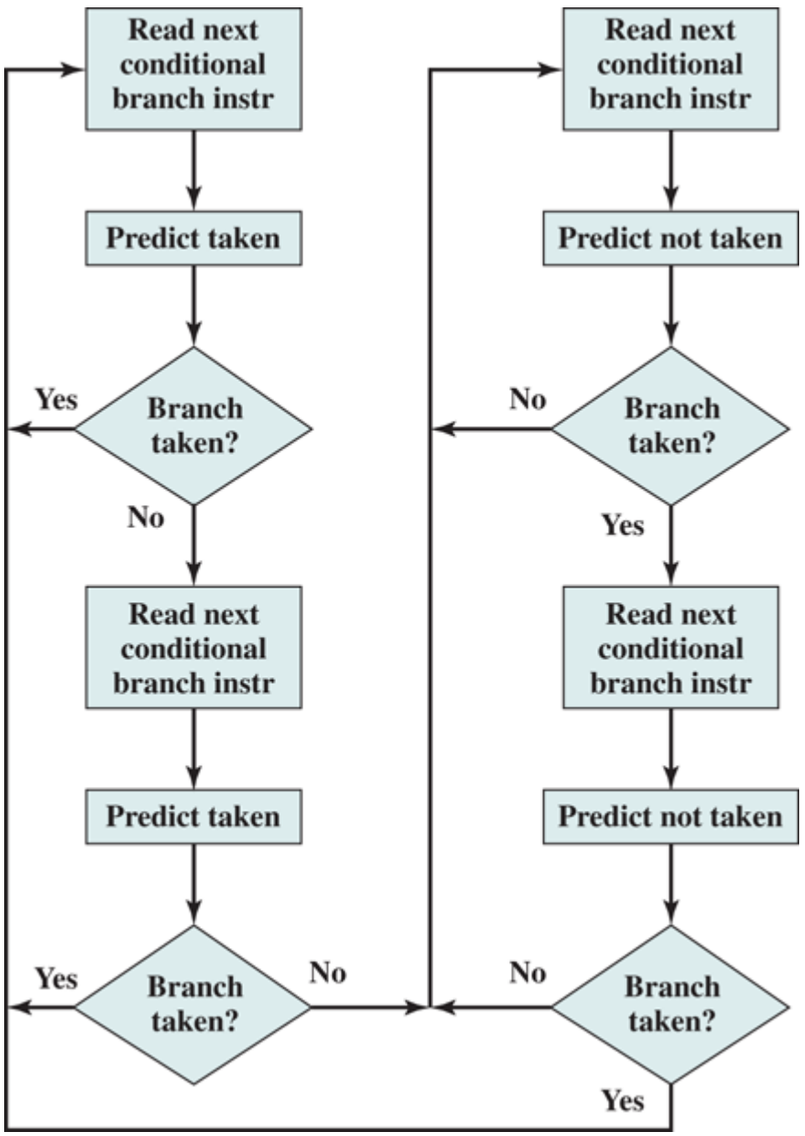


Figure 16.18 Branch Prediction Flowchart

The decision process can be represented more compactly by a finite-state machine, shown in **Figure 16.19**. The finite-state machine representation is commonly used in the literature.

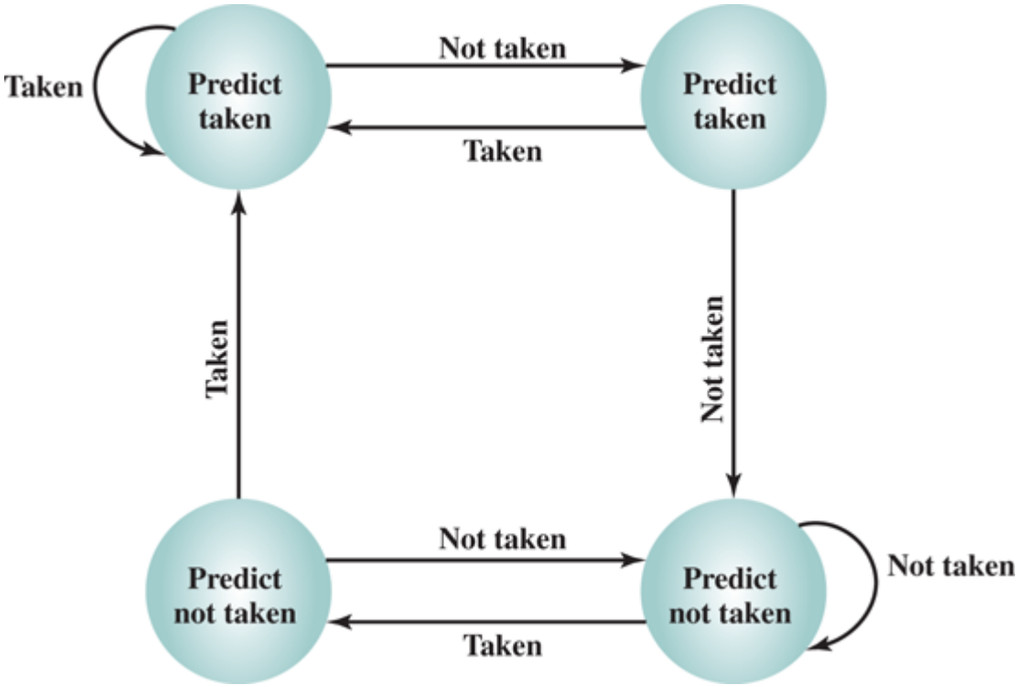


Figure 16.19 Branch Prediction State Diagram

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 16.20 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 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.

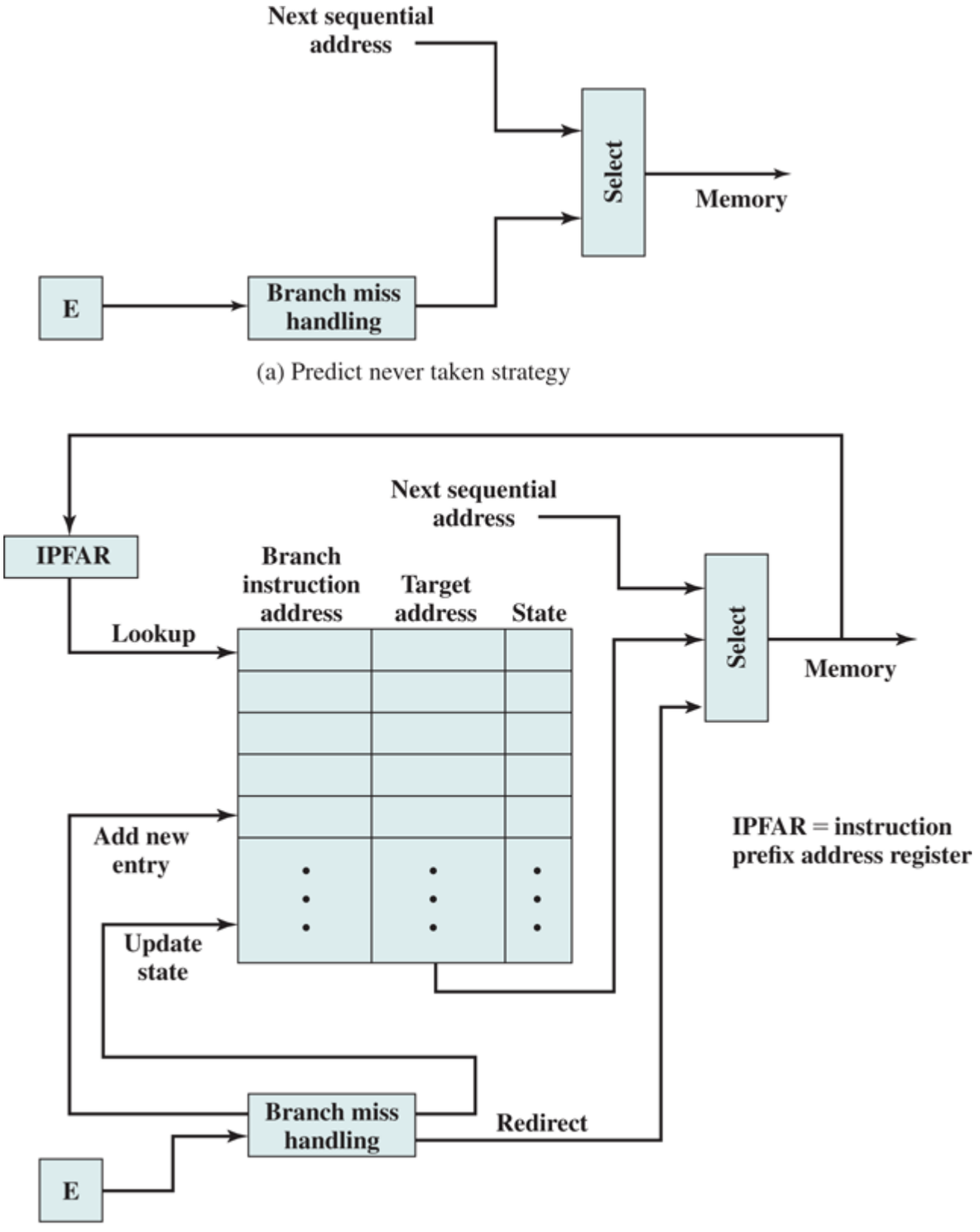


Figure 16.20 Dealing with Branches

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 5**.

A refinement of the branch history approach is referred to as two-level or correlation-based branch history [YEH91]. This approach is based on the assumption that whereas in loop-closing branches, the past history of a particular branch instruction is a good predictor of future behavior, with more complex control-flow structures the direction of a branch is frequently correlated with the direction of related branches. An example is an if-then-else or case structure. There are a number of possible strategies. Typically, recent global branch history (i.e., the history of the most recent branches, not just of this branch instruction) is used in addition to the history of the current branch instruction. The general structure is defined as an (*m, n*) correlator, which uses the behavior of the last *m* branches to choose from 2^m *n*-bit branch predictors for the current branch instruction. In other words, an *n*-bit history is kept for a given branch for each possible combination of branches taken by the most recent *m* branches.

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 17**.

Intel 80486 Pipelining

An instructive example of an instruction pipeline is that of the Intel 80486. 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 16.21 shows examples of the operation of the pipeline. **Figure 16.21a** shows that there is no delay introduced into the pipeline when a memory access is required. However, as **Figure 16.21b** 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.

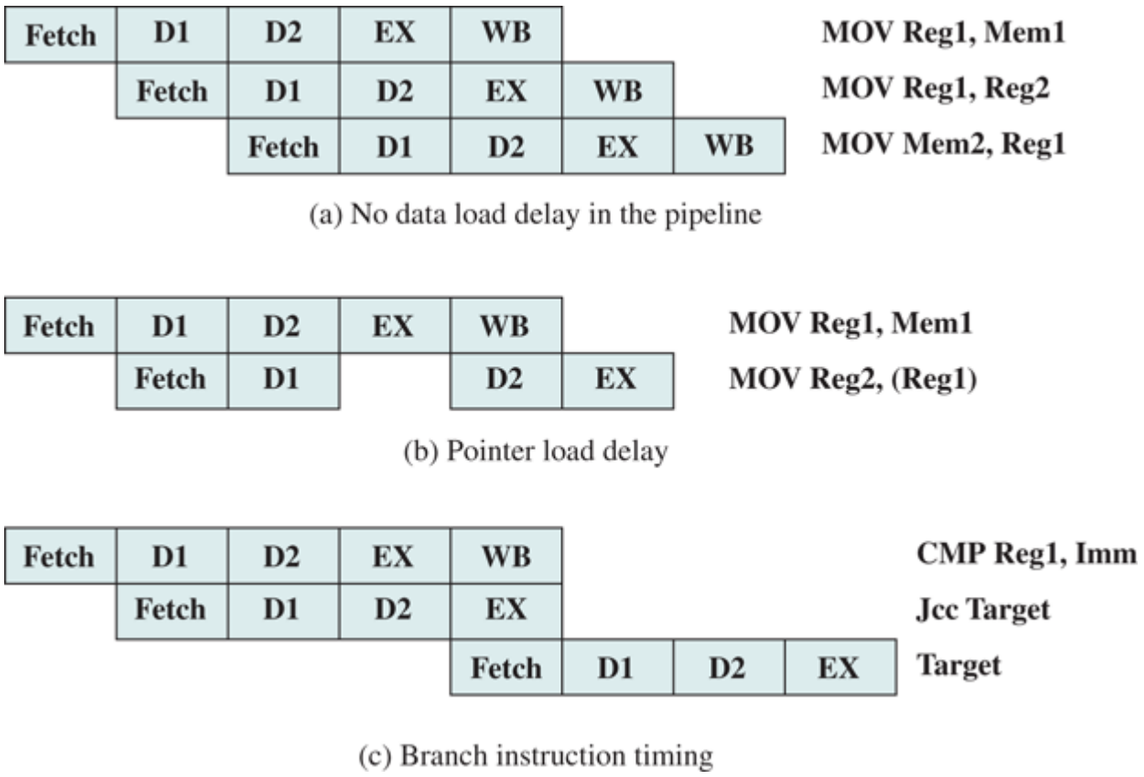


Figure 16.21 80486 Instruction Pipeline Examples

Figure 16.21c 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).

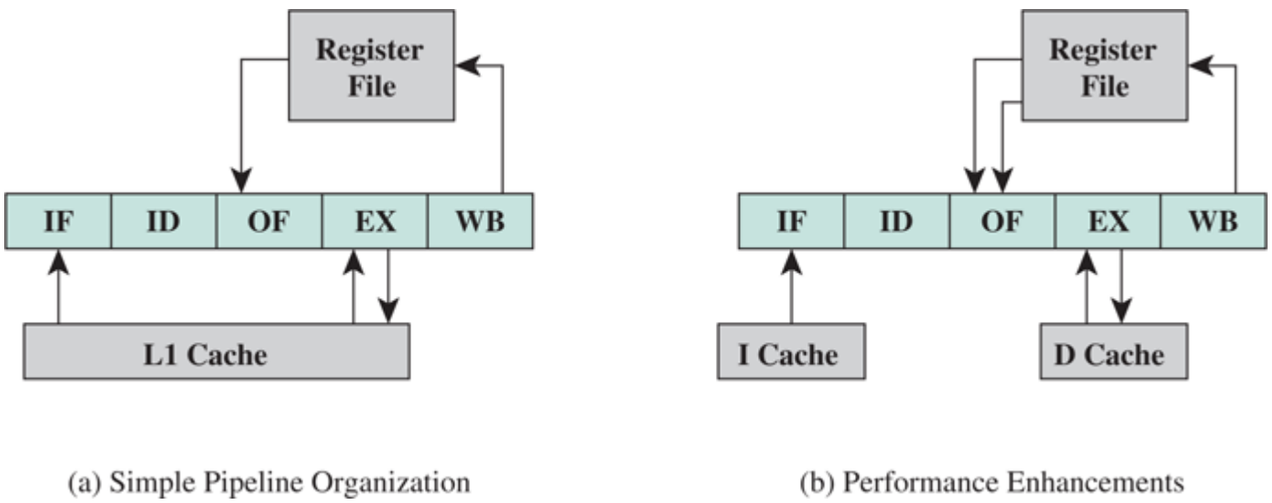
16.5 Processor Organization for Pipelining

This section looks at some of the enhancements to a simple pipeline that can be used to improve performance. Consider a five-stage pipeline:

- **Instruction fetch (IF):** Load instruction from cache.
- **Instruction decode (ID):** Determine the opcode and the operand specifiers.
- **Operand fetch (OF):** Read and buffer any register operands.
- **Execute (EX):** Perform the indicated operation. For a memory load or store, this involves memory access through the cache.
- **Write back (WB):** Write back instruction result to its destination register.

Figure 16.22a indicates a simple organization for this pipeline. Two contention problems are apparent. The WB and OF stages both need access to the register file, and the IF and EX stages both need access to the cache. **Figure 16.22b** shows modifications to the organization to alleviate these conflicts. Increasing the number of register ports and busses enables simultaneous read and write. Separating the L1 cache into an I-cache and a D-cache removes the conflict between the IF and EX stages.

Figure 16.23 shows a more complex organization that further enhances performance. The following are the changes:



(a) Simple Pipeline Organization

(b) Performance Enhancements

Figure 16.22 Approaches to Pipeline Organization

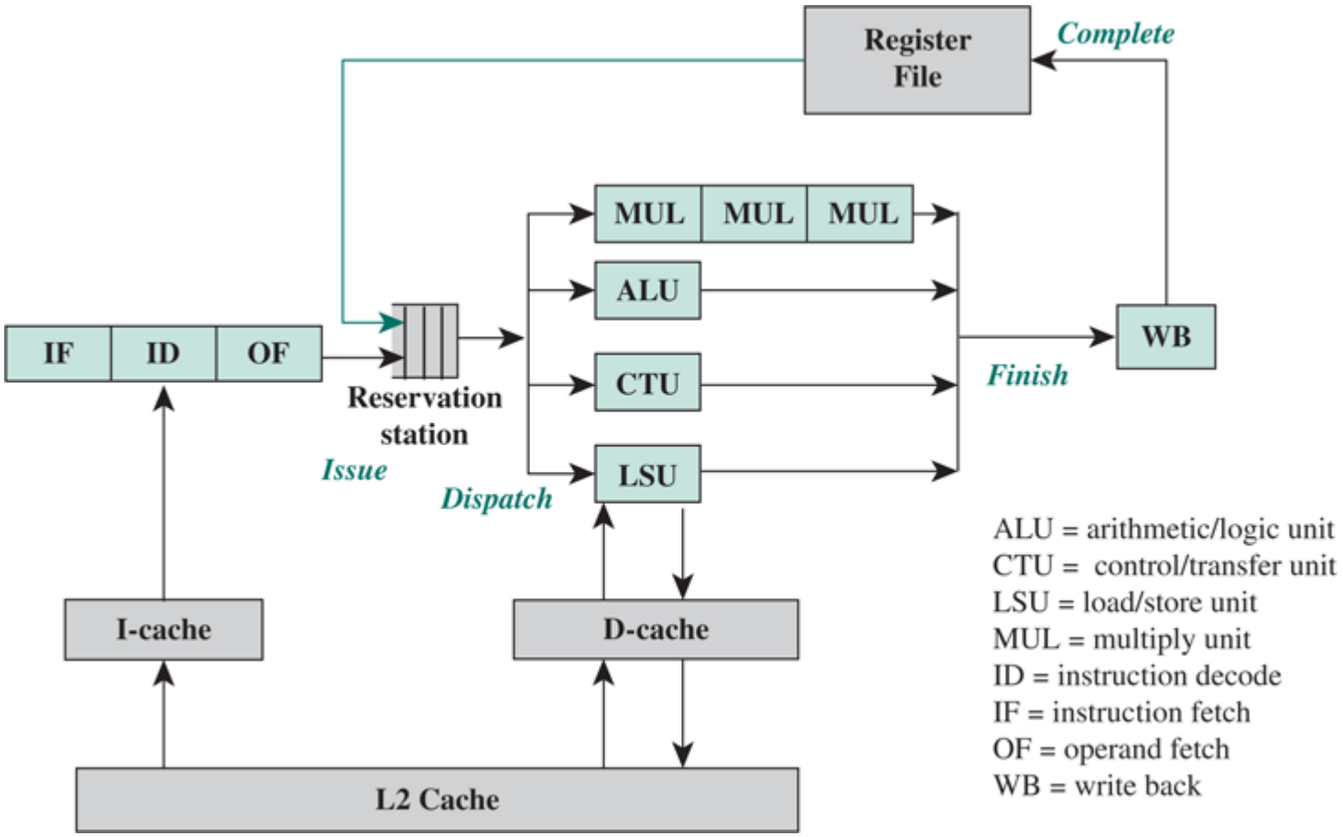


Figure 16.23 Improved Pipeline Organization

- **Dedicated execution units for each function:** Different units can have different time delays, allowing for more flexible pipelining.
- **Pipelining a functional unit:** Because different functional (EX) units have different time delays, it is possible to pipeline executions in a longer unit. For example, an integer multiply unit may take multiple clock cycles, compared to only one clock cycle for add/subtract and control transfer. Instead of having to stall the pipeline until the entire multiply operation is complete, a new EX stage can be started as soon as the first EX stage of the multiply is complete.
- **Reservation station:** A buffer used to hold operations and operands for an EX unit until the operands are available.

The purpose of the reservation station is to relieve a bottleneck at the OF stage. The OF can issue an instruction as soon as a functional unit is available and hazards are resolved. The problem this creates is that the OF stage cannot receive a new instruction until the previous instruction has been issued. The reservation stations provide a buffer that enables the OF stage to issue instructions as soon as possible. Then, the reservation station will dispatch each instruction to its functional unit when the latter is available.

Figure 16.24 shows the typical contents of a reservation station for a machine that has up to two operands per instruction. Each slot (shown vertically in the figure) holds information for one instruction consisting of one or more tag/value pairs and an OP field. The OP field is the instruction operation command for a functional unit. The Tag field indicates “valid” if the corresponding Value field contains an operand. Otherwise the Tag field indicates the identity of the desired operand, such as by using a register number. If the desired operand is available, it is copied from a register to the Value field; otherwise, the slot is in a waiting state until the operand is available.

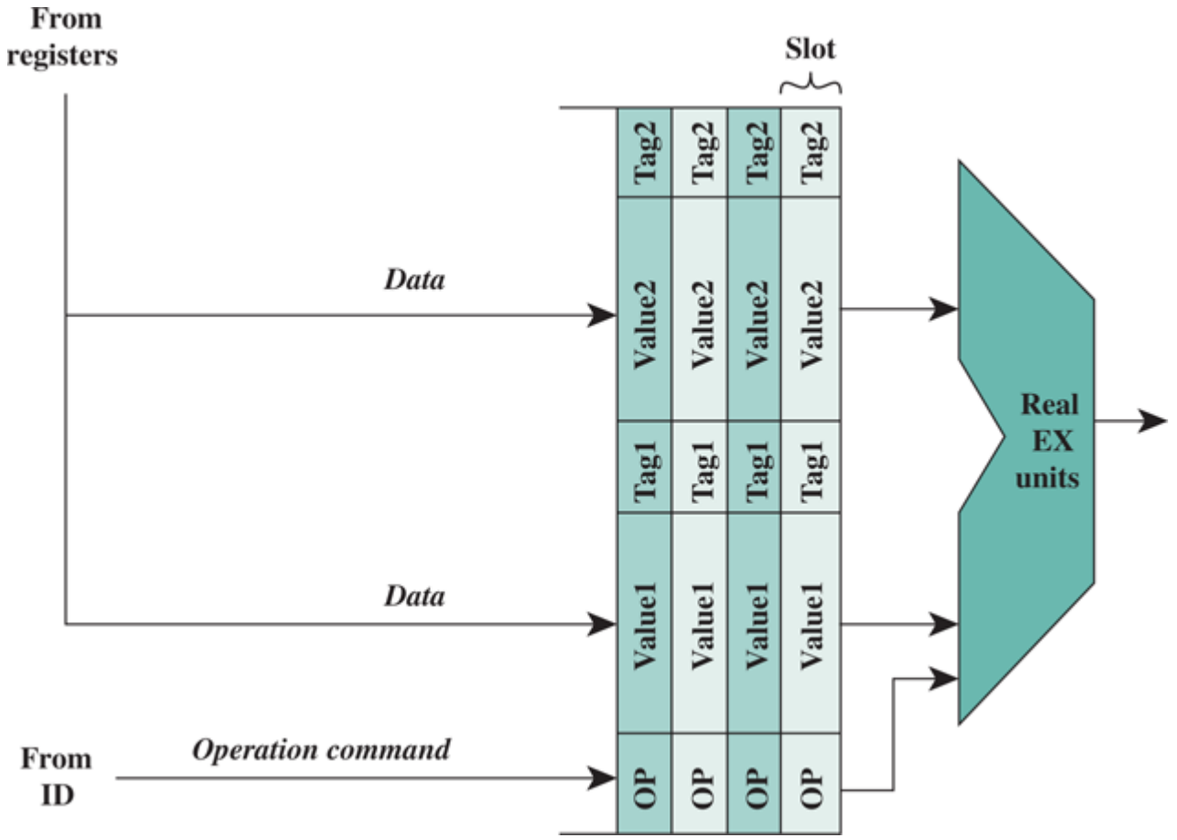


Figure 16.24 Reservation Station Contents

16.6 The x86 Processor Family

The x86 organization has evolved dramatically over the years. In this section we examine some of the details of the most recent processor organizations, concentrating on common elements in single processors. **Chapter 18** looks at superscalar aspects of the x86, and **Chapter 20** examines the multicore organization. An overview of the Pentium 4 processor organization is depicted in **Figure 5.17**.

Register Organization

The register organization includes the following types of registers (**Table 16.2**):

Table 16.2 x86 Processor Registers

(a) Integer Unit in 32-bit Mode			
Type	Number	Length (bits)	Purpose
General	8	32	General-purpose user registers
Segment	6	16	Contain segment selectors
EFLAGS	1	32	Status and control bits
Instruction Pointer	1	32	Instruction pointer

(b) Integer Unit in 64-bit Mode			
Type	Number	Length (bits)	Purpose
General	16	32	General-purpose user registers
Segment	6	16	Contain segment selectors
RFLAGS	1	64	Status and control bits
Instruction Pointer	1	64	Instruction pointer

(c) 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

- **General:** There are eight 32-bit general-purpose registers (see **Figure 16.3c**). These may be used for all types of x86 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. In 64-bit mode, there are sixteen 64-bit general-purpose registers.
- **Segment:** The six 16-bit segment registers contain segment selectors, which index into segment tables, as discussed in **Chapter 9**. The code segment (CS) register references the segment containing the instruction being executed. The 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 32-bit EFLAGS register contains condition codes and various mode bits. In 64-bit mode, this register is extended to 64 bits and referred to as RFLAGS. In the current architecture definition, the upper 32 bits of RFLAGS are unused.
- **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.

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 16.25**) indicates the condition of the processor and helps to control its operation. It includes the six condition codes defined in**Table 13.8** (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:

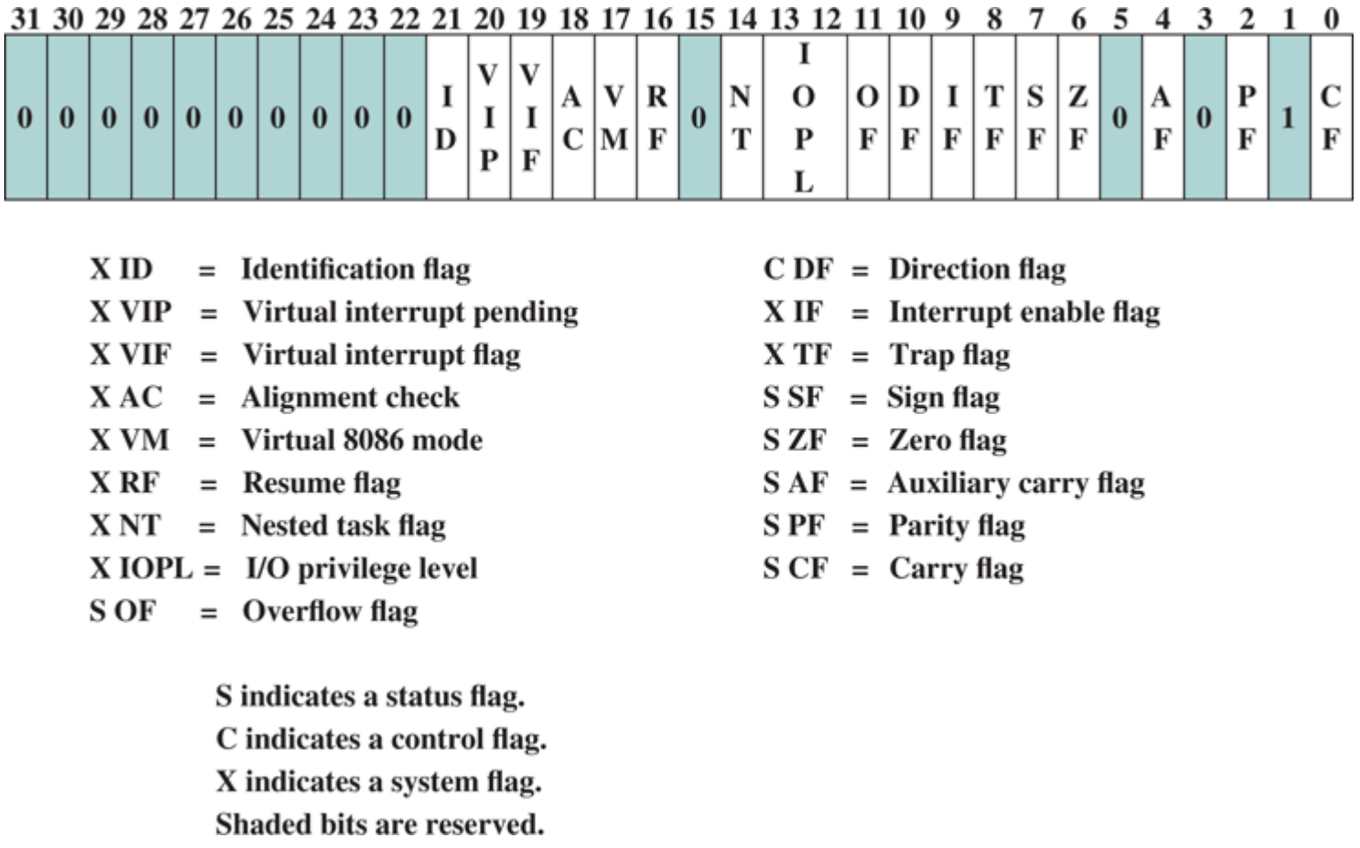


Figure 16.25 x86 EFLAGS Register

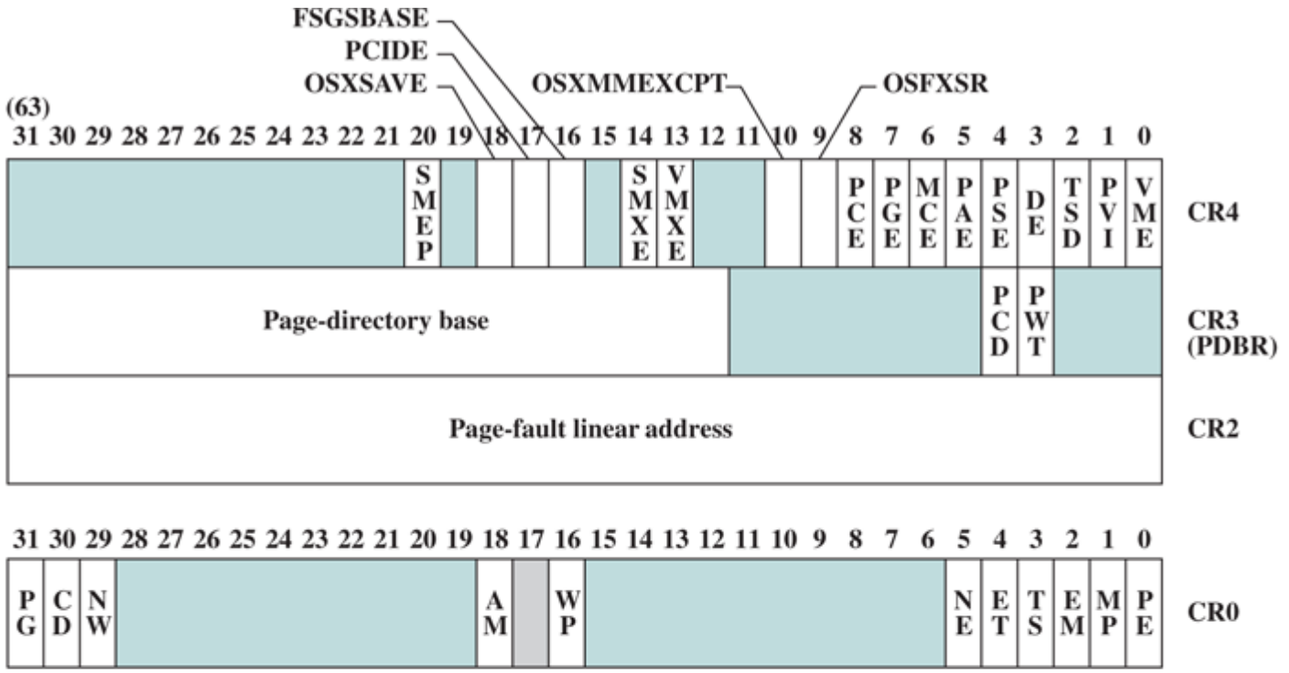
- **Trap flag (TF):** When set, causes an interrupt after the execution of each instruction. This is used for debugging.
- **Interrupt enable flag (IF):** When set, the processor will recognize external interrupts.
- **Direction flag (DF):** Determines whether string processing instructions increment 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.

CONTROL REGISTERS

The x86 employs four control registers (register CR1 is unused) to control various aspects of processor operation (**Figure 16.26**). All of the registers except CR0 are either 32 bits or 64 bits long, depending on whether the implementation supports the x86 64-bit architecture. 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:



Shaded area indicates reserved bits.

OSXSAVE	=	XSAVE enable bit	VME	=	Virtual 8086 mode extensions
PCIDE	=	Enables process-context identifiers	PCD	=	Page-level cache disable
FSGSBASE	=	Enables segment base instructions	PWT	=	Page-level writes transparent
SMXE	=	Enable safer mode extensions	PG	=	Paging
VMXE	=	Enable virtual machine extensions	CD	=	Cache disable
OSXMMEXCPT	=	Support unmasked SIMD FP exceptions	NW	=	Not write through
OSFXSR	=	Support FXSAVE, FXSTOR	AM	=	Alignment mask
PCE	=	Performance counter enable	WP	=	Write protect
PGE	=	Page global enable	NE	=	Numeric error
MCE	=	Machine check enable	ET	=	Extension type
PAE	=	Physical address extension	TS	=	Task switched
PSE	=	Page size extensions	EM	=	Emulation
DE	=	Debug extensions	MP	=	Monitor coprocessor
TSD	=	Time stamp disable	PE	=	Protection enable
PVI	=	Protected mode virtual interrupt			

Figure 16.26 x86 Control Registers

- **Protection Enable (PE):** Enable/disable protected mode of operation.
- **Monitor Coprocessor (MP):** Only of interest when running programs from earlier machines on the x86; 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 and later machines; 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. CR4 contains additional control bits.

MMX REGISTERS

Recall from **Section 13.3** that theThe x86 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 16.27**). 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 older 32-bit x86 architecture is easily extended to support the MMX capability. Some key characteristics of the MMX use of these registers are as follows:

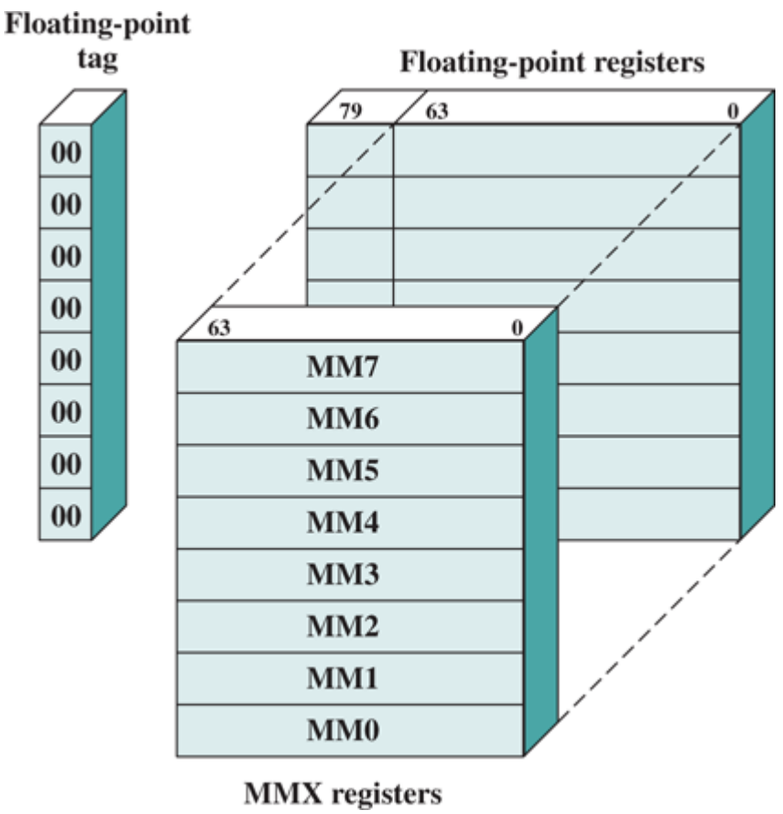


Figure 16.27 Mapping of MMX Registers to Floating-Point Registers

- 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 x86 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 x86 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 16.3 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.

Table 16.3 x86 Exception and Interrupt Vector Table
Unshaded: exceptions

Shaded: interrupts

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

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.
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.

16.7 The ARM Processor

In this section, we look at some of the key elements of the ARM architecture and organization. We defer a discussion of more complex aspects of organization and pipelining until **Chapter 18**. For the discussion in this section and in **Chapter 18**, it is useful to keep in mind key characteristics of the ARM architecture. ARM is primarily a RISC system with the following notable attributes:

- A moderate array of uniform registers, more than are found on some CISC systems but fewer than are found on many RISC systems.
- A load/store model of data processing, in which operations only perform on operands in registers and not directly in memory. All data must be loaded into registers before an operation can be performed; the result can then be used for further processing or stored into memory.
- A uniform fixed-length instruction of 32 bits for the standard set and 16 bits for the Thumb instruction set.
- To make each data processing instruction more flexible, either a shift or rotation can preprocess one of the source registers. To efficiently support this feature, there are separate arithmetic logic unit (ALU) and shifter units.
- A small number of addressing modes with all load/store addressees determined from registers and instruction fields. Indirect or indexed addressing involving values in memory are not used.
- Auto-increment and auto-decrement addressing modes are used to improve the operation of program loops.
- Conditional execution of instructions minimizes the need for conditional branch instructions, thereby improving pipeline efficiency, because pipeline flushing is reduced.

Processor Organization

The ARM processor organization varies substantially from one implementation to the next, particularly when based on different versions of the ARM architecture. However, it is useful for the discussion in this section to present a simplified, generic ARM organization, which is illustrated in **Figure 16.28**. In this figure, the arrows indicate the flow of data. Each box represents a functional hardware unit or a storage unit.

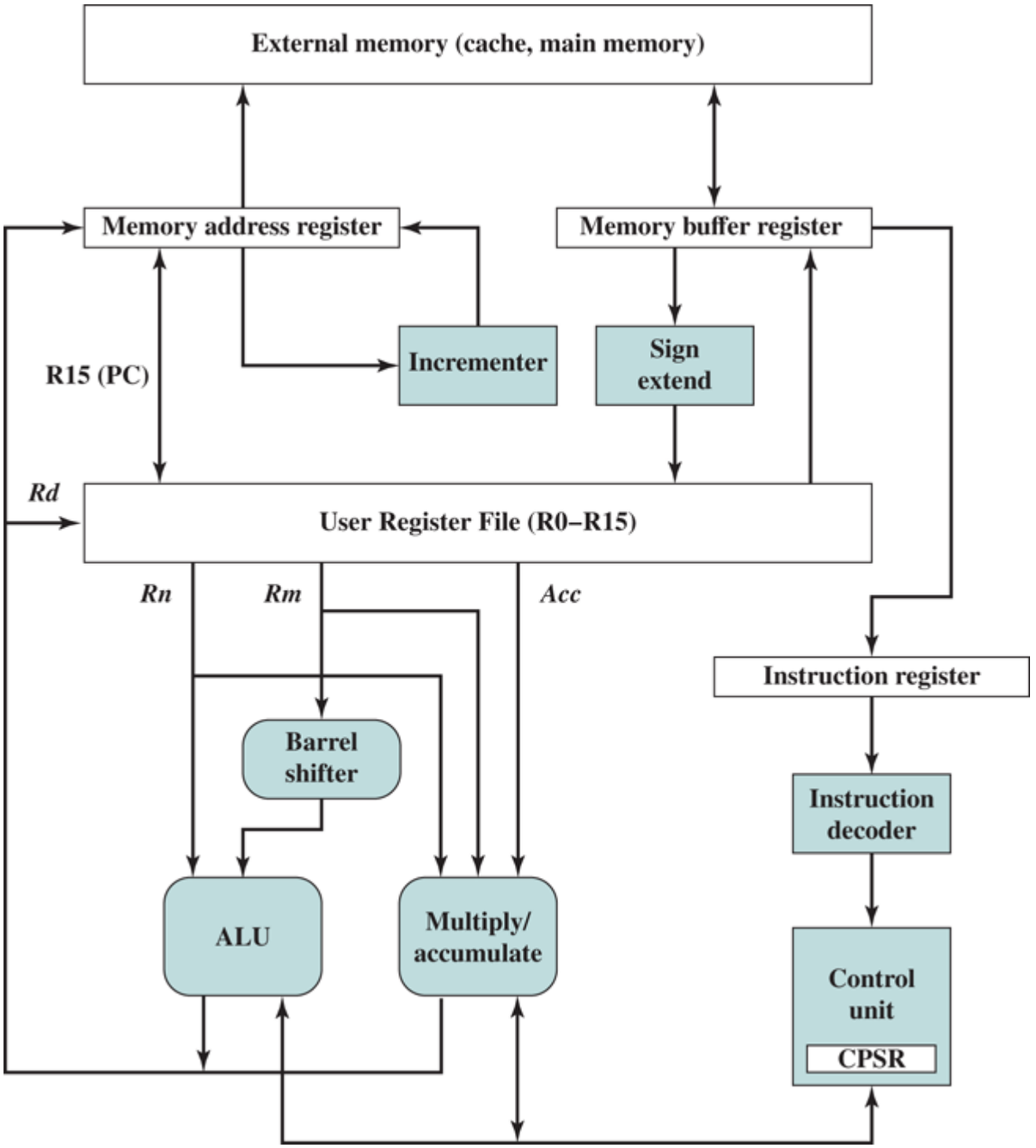


Figure 16.28 Simplified ARM Organization

Data are exchanged with the processor from external memory through a data bus. The value transferred is either a data item, as a result of a load or store instruction, or an instruction fetch. Fetched instructions pass through an instruction decoder before execution, under control of a control unit. The latter includes pipeline logic and provides control signals (not shown) to all the hardware elements of the processor. Data items are placed in the register file, consisting of a set of 32-bit registers. Byte or halfword items treated as twos-complement numbers are sign-extended to 32 bits.

ARM data processing instructions typically have two source registers, *Rn* and *Rm*, and a single result or destination register, *Rd*. The source register values feed into the ALU or a separate multiply unit that makes use of an additional register to accumulate partial results. The ARM processor also includes a hardware unit that can shift or rotate the *Rm* value before it enters the ALU. This shift or rotate occurs within the cycle time of the instruction and increases the power and flexibility of many data processing operations.

The results of an operation are fed back to the destination register. Load/store instructions may also use the output of the arithmetic units to generate the memory address for a load or store.

Processor Modes

It is quite common for a processor to support only a small number of processor modes. For example, many operating systems make use of just two modes: a user mode and a kernel mode, with the latter mode used to execute privileged system software. In contrast, the ARM architecture provides a flexible foundation for operating systems to enforce a variety of protection policies.

The ARM architecture supports seven execution modes. Most application programs execute in **user mode**. While the processor is in user mode, the program being executed is unable to access protected system resources or to change mode, other than by causing an exception to occur.

The remaining six execution modes are referred to as privileged modes. These modes are used to run system software. There are two principal advantages to defining so many different privileged modes: (1) The OS can tailor the use of system software to a variety of circumstances, and (2) certain registers are dedicated for use for each of the privileged modes, allowing swifter changes in context.

The exception modes have full access to system resources and can change modes freely. Five of these modes are known as exception modes. These are entered when specific exceptions occur. Each of these modes has some dedicated registers that substitute for some of the user mode registers, and which are used to avoid corrupting user mode state information when the exception occurs. The exception modes are as follows:

- **Supervisor mode:** Usually what the OS runs in. It is entered when the processor encounters a software interrupt instruction. Software interrupts are a standard way to invoke operating system services on ARM.
- **Abort mode:** Entered in response to memory faults.
- **Undefined mode:** Entered when the processor attempts to execute an instruction that is supported neither by the main integer core nor by one of the coprocessors.
- **Fast interrupt mode:** Entered whenever the processor receives an interrupt signal from the designated fast interrupt source. A fast interrupt cannot be interrupted, but a fast interrupt may interrupt a normal interrupt.
- **Interrupt mode:** Entered whenever the processor receives an interrupt signal from any other interrupt source (other than fast interrupt). An interrupt may only be interrupted by a fast interrupt.

The remaining privileged mode is the **System mode**. This mode is not entered by any exception and uses the same registers available in User mode. The System mode is used for running certain privileged operating system tasks. System mode tasks may be interrupted by any of the five exception categories.

Register Organization

Figure 16.29 depicts the user-visible registers for the ARM. The ARM processor has a total of 37 32-bit registers, classified as follows:

- Thirty-one registers referred to in the ARM manual as general-purpose registers. In fact, some of these, such as the program counters, have special purposes.
- Six program status registers.

Registers are arranged in partially overlapping banks, with the current processor mode determining which bank is available. At any time, sixteen numbered registers and one or two program status

registers are visible, for a total of 17 or 18 software-visible registers. **Figure 16.29** is interpreted as follows:

- Registers R0 through R7, register R15 (the program counter) and the current program status register (CPSR) are visible in and shared by all modes.
- Registers R8 through R12 are shared by all modes except fast interrupt, which has its own dedicated registers R8_fiq through R12_fiq.
- All the exception modes have their own versions of registers R13 and R16.
- All the exception modes have a dedicated saved program status register (SPSR).

Modes						
	Privileged modes					
		Exception modes				
User	**System**	**Supervisor**	**Abort**	**Undefined**	**Interrupt**	**Fast interrupt**
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13(SP)	R13(SP)	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14(LR)	R14(LR)	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Shading indicates that the normal register used by User or System mode has been replaced by an alternative register specific to the exception mode.
SP = stack pointer **CPSR = current program status register**
LR = link register **SPSR = saved program status register**
PC = program counter

Figure 16.29 ARM Register Organization

GENERAL-PURPOSE REGISTERS

Register R13 is normally used as a stack pointer and is also known as the SP. Because each exception mode has a separate R13, each exception mode can have its own dedicated program stack. R14 is known as the link register (LR) and is used to hold subroutine return addresses and exception mode returns. Register R15 is the program counter (PC).

PROGRAM STATUS REGISTERS

The CPSR is accessible in all processor modes. Each exception mode also has a dedicated SPSR that is used to preserve the value of the CPSR when the associated exception occurs.

The 16 most significant bits of the CPSR contain user flags visible in user mode, and which can be used to affect the operation of a program (**Figure 16.30**). These are as follows:

- **Condition code flags:** The N, Z, C, and V flags,They are the N, Z, C, and V flags which are discussed in **Chapter 13**.
- **Q flag:** used to indicate whether overflow and/or saturation has occurred in some SIMD-oriented instructions.
- **J bit:** indicates the use of special 8-bit instructions, known as Jazelle instructions, which are beyond the scope of our discussion.
- **GE[3:0] bits:** SIMD instructions use bits [19:16] as Greater than or Equal (GE) flags for individual bytes or halfwords of the result.

The 16 least significant bits of the CPSR contain system control flags that can only be altered when the processor is in a privileged mode. The fields are as follows:

- **E bit:** Controls load and store endianness for data; ignored for instruction fetches.
- **Interrupt disable bits:** The A bit disables imprecise data aborts when set; the I bit disables IRQ interrupts when set; and the F bit disables FIQ interrupts when set.
- **T bit:** Indicates whether instructions should be interpreted as normal ARM instructions or Thumb instructions.
- **Mode bits:** Indicates the processor mode.

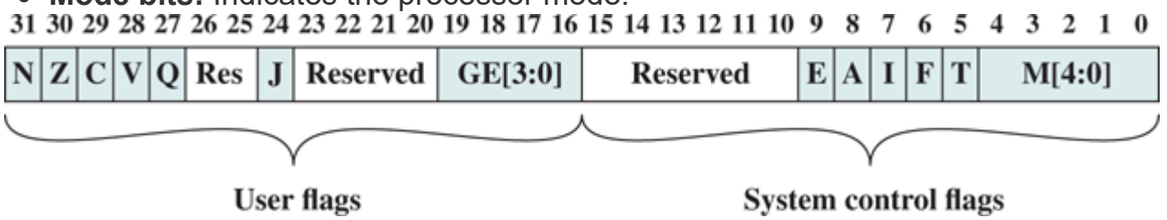


Figure 16.30 Format of ARM CPSR and SPSR

Interrupt Processing

As with any processor, the ARM includes a facility that enables the processor to interrupt the currently executing program to deal with exception conditions. Exceptions are generated by internal and external sources to cause the processor to handle an event. The processor state just before handling the exception is normally preserved so that the original program can be resumed when the exception routine has completed. More than one exception can arise at the same time. The ARM architecture supports seven types of exceptions. **Table 16.4** lists the types of exception and the processor mode that is used to process each type. When an exception occurs, execution is forced from a fixed memory address corresponding to the type of exception. These fixed addresses are called the exception vectors.

Table 16.4 ARM Interrupt Vector

Exception type	Mode	Normal entry address	Description
Reset	Supervisor	0x00000000	Occurs when the system is initialized.
Data abort	Abort	0x00000010	Occurs when an invalid memory address has been accessed, such as if there is no physical memory for an address or the correct access permission is lacking.
FIQ (fast interrupt)	FIQ	0x0000001C	Occurs when an external device asserts the FIQ pin on the processor. An interrupt cannot be interrupted except by an FIQ. FIQ is designed to support a data transfer or channel process, and has sufficient private registers to remove the need for register saving in such applications, therefore minimizing the overhead of context switching. A fast interrupt cannot be interrupted.
IRQ (interrupt)	IRQ	0x00000018	Occurs when an external device asserts the IRQ pin on the processor. An interrupt cannot be interrupted except by an FIQ.
Prefetch abort	Abort	0x0000000C	Occurs when an attempt to fetch an instruction results in a memory fault. The exception is raised when the instruction enters the execute stage of the pipeline.
Undefined instructions	Undefined	0x00000004	Occurs when an instruction not in the instruction set reaches the execute stage of the pipeline.
Software interrupt	Supervisor	0x00000008	Generally used to allow user mode programs to call the OS. The user program executes a SWI instruction with an argument that identifies the function the user wishes to perform.

If more than one interrupt is outstanding, they are handled in priority order. **Table 16.4** lists the exceptions in priority order from highest to lowest.

When an exception occurs, the processor halts execution after the current instruction. The state of the processor is preserved in the SPSR that corresponds to the type of exception, so that the original program can be resumed when the exception routine has completed. The address of the instruction the processor was just about to execute is placed in the link register of the appropriate processor mode. To return after handling the exception, the SPSR is moved into the CPSR and R14 is moved

into the PC.

16.8 Key Terms, Review Questions, and Problems

Key Terms

branch prediction

condition code

delayed branch

flag

functional unit

instruction cycle

instruction pipeline

instruction prefetch

program status word (PSW)

reservation station

Review Questions

16.1 What general roles are performed by processor registers?
16.2 What categories of data are commonly supported by user-visible registers?
16.3 What is the function of condition codes?
16.4 What is a program status word?
16.5 Why is a two-stage instruction pipeline unlikely to cut the instruction cycle time in half, compared with the use of no pipeline?
16.6 List and briefly explain various ways in which an instruction pipeline can deal with conditional branch instructions.
16.7 How are history bits used for branch prediction?

Problems

16.1

a. If the last operation performed on a computer with an 8-bit word was an addition in which the two operands were 00000010 and 00000011, what would be the value of the following flags?
   - Carry
   - Zero
   - Overflow
   - Sign
   - Even Parity
   - Half-Carry

b. Repeat for the addition of -1 (twos complement) and $+1$.

16.2 Repeat Problem 16.1 for the operation $A - B$, where A contains 11110000 and B contains 0010100.

16.3 A microprocessor is clocked at a rate of 5 GHz.

a. How long is a clock cycle?
b. What is the duration of a particular type of machine instruction consisting of three clock cycles?

16.4 A microprocessor provides an instruction capable of moving a string of bytes from one area of memory to another. The fetching and initial decoding of the instruction takes 10 clock cycles. Thereafter, it takes 15 clock cycles to transfer each byte. The microprocessor is clocked at a rate of 10 GHz.

a. Determine the length of the instruction cycle for the case of a string of 64 bytes.
b. What is the worst-case delay for acknowledging an interrupt if the instruction is noninterruptible?
c. Repeat part (b) assuming the instruction can be interrupted at the beginning of each byte transfer.

16.5 The Intel 8088 consists of a bus interface unit (BIU) and an execution unit (EU), which form a 2-stage pipeline. The BIU fetches instructions into a 4-byte instruction queue. The BIU also participates in address calculations, fetches operands, and writes results in memory as requested by the EU. If no such requests are outstanding and the bus is free, the BIU fills any vacancies in the instruction queue. When the EU completes execution of an instruction, it passes any results to the BIU (destined for memory or I/O) and proceeds to the next instruction.

a. Suppose the tasks performed by the BIU and EU take about equal time. By what factor does pipelining improve the performance of the 8088? Ignore the effect of branch instructions.
b. Repeat the calculation assuming that the EU takes twice as long as the BIU.

16.6 Assume an 8088 is executing a program in which the probability of a program jump is 0.1. For simplicity, assume that all instructions are 2 bytes long.

a. What fraction of instruction fetch bus cycles is wasted?
b. Repeat if the instruction queue is 8 bytes long.

16.7 Consider the timing diagram of **Figures 16.10** . Assume that there is only a two-stage pipeline (fetch, execute). Redraw the diagram to show how many time units are now needed for four instructions.

16.8 Assume a pipeline with four stages: fetch instruction (FI), decode instruction and calculate addresses (DA), fetch operand (FO), and execute (EX). Draw a diagram similar to **Figure 16.10** for a sequence of 7 instructions, in which the third instruction is a branch that is taken and in which there are no data dependencies.

16.9 A pipelined processor has a clock rate of 2.5 GHz and executes a program with 1.5 million instructions. The pipeline has five stages, and instructions are issued at a rate of one per clock cycle. Ignore penalties due to branch instructions and out-of-sequence executions.

a. What is the speedup of this processor for this program compared to a nonpipelined processor, making the same assumptions used in **Section 16.4** ?
b. What is throughput (in MIPS) of the pipelined processor?

16.10 A nonpipelined processor has a clock rate of 2.5 GHz and an average CPI (cycles per instruction) of 4. An upgrade to the processor introduces a five-stage pipeline. However, due to internal pipeline delays, such as latch delay, the clock rate of the new processor has to be reduced to 2 GHz.

a. What is the speedup achieved for a typical program?
b. What is the MIPS rate for each processor?

16.11 Consider an instruction sequence of length *n* that is streaming through the instruction pipeline. Let *p* be the probability of encountering a conditional or unconditional branch instruction, and let *q* be the probability that execution of a branch instruction I causes a jump to a nonconsecutive address. Assume that each such jump requires the pipeline to be cleared, destroying all ongoing instruction processing, when I emerges from the last stage. Revise **Equations (16.1)** and **(16.2)** to take these probabilities into account.

16.12 One limitation of the multiple-stream approach to dealing with branches in a pipeline is that additional branches will be encountered before the first branch is resolved. Suggest two additional limitations or drawbacks.

16.13 Consider the state diagrams of **Figure 16.31** .

a. Describe the behavior of each.
b. Compare these with the branch prediction state diagram in **Section 16.4** . Discuss the relative merits of each of the three approaches to branch prediction.

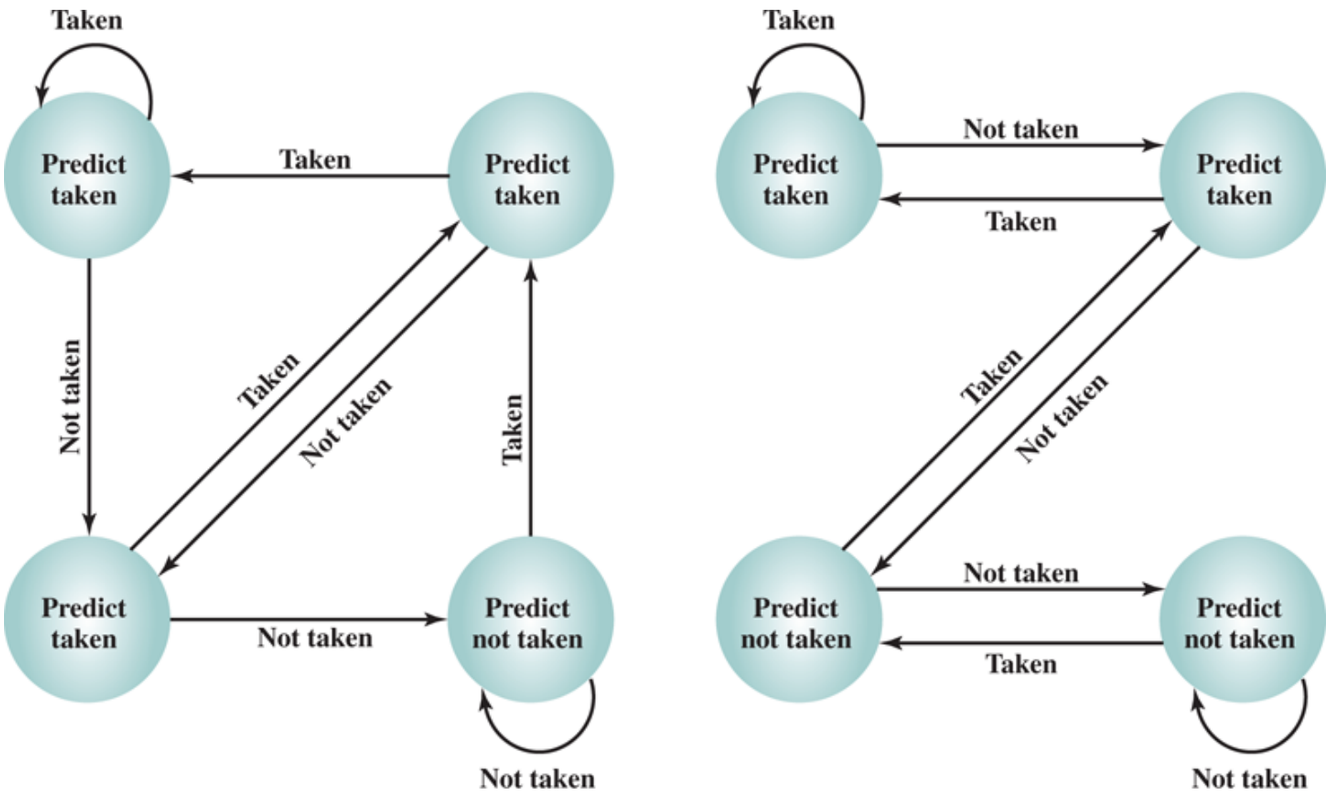


Figure 16.31 Two Branch Prediction State Diagrams

16.14 The Motorola 680x0 machines include the instruction Decrement and Branch According to Condition, which has the following form:

DBcc Dn, displacement

where cc is one of the testable conditions, Dn is a general-purpose register, and displacement specifies the target address relative to the current address. The instruction can be defined as follows:

```
if (cc = False)
then begin
     Dn: = (Dn) -1;
     if Dn Z -1 then PC: = (PC) + displacement end
else PC: = (PC) + 2;
```

When the instruction is executed, the condition is first tested to determine whether the termination condition for the loop is satisfied. If so, no operation is performed and execution continues at the next instruction in sequence. If the condition is false, the specified data register is decremented and checked to see if it is less than zero. If it is less than zero, the loop is terminated and execution continues at the next instruction in sequence. Otherwise, the program branches to the specified location. Now consider the following assembly-language program fragment:

AGAIN	CMPM.L	$(A0)+,(A1)+$
	DBNE	D1, AGAIN
	NOP	

Two strings addressed by A0 and A1 are compared for equality; the string pointers are incremented with each reference. D1 initially contains the number of longwords (4 bytes) to be compared.

a. The initial contents of the registers are $A0 = \$00004000, A1 = \$00005000$ and $D1 = \$000000FF$ (the $ indicates hexadecimal notation). Memory between $4000 and $6000 is loaded with words $AAAA. If the foregoing program is run, specify the number of times the DBNE loop is executed and the contents of the three registers when the NOP instruction is reached.
b. Repeat (a), but now assume that memory between $4000 and $4FEE is loaded with $0000 and between $5000 and $6000 is loaded with $AAA.

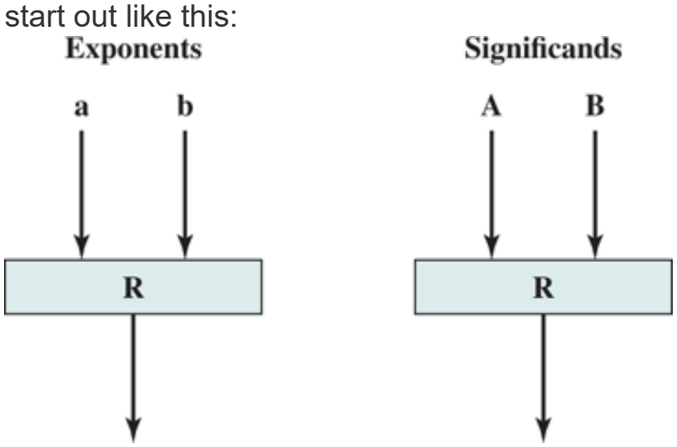
16.15 Redraw **Figure 16.19c** , assuming that the conditional branch is not taken.
16.16 **Table 16.5** summarizes statistics from [MACD84] concerning branch behavior for various classes of applications. With the exception of type 1 branch behavior, there is no noticeable difference among the application classes. Determine the fraction of all branches that go to the branch target address for the scientific environment. Repeat for commercial and systems environments.

Table 16.5 Branch Behavior in Sample Applications

Occurrence of branch classes:				
Type 1: Branch	72.5%			
Type 2: Loop control	9.8%			
Type 3: Procedure call, return	17.7%			
Type 1 branch: where it goes		**Scientific**	**Commercial**	**Systems**
Unconditional—100% go to target		20%	40%	35%
Conditional—went to target		43.2%	24.3%	32.5%

Conditional—did not go to target (inline)	36.8%	35.7%	32.5%
Type 2 branch (all environments)			
That go to target	91%		
That go inline	9%		
Type 3 branch			
100% go to target			

16.17 Pipelining can be applied within the ALU to speed up floating-point operations. Consider the case of floating-point addition and subtraction. In simplified terms, the pipeline could have four stages: (1) Compare the exponents; (2) Choose the exponent and align the significands; (3) Add or subtract significands; (4) Normalize the results. The pipeline can be considered to have two parallel threads, one handling exponents and one handling significands, and could start out like this:



In this figure, the boxes labeled R refer to a set of registers used to hold temporary results. Complete the block diagram that shows at a top level the structure of the pipeline.