The following parameters are cited for the 68000 design: $N = 70$, $H_m = 650$, and $r = 0.4$ so that $H_n = 260$. Substituting into (5.20) and (5.21), we obtain $S_1 = 52{,}450$ and $S_2 = 30{,}550$. Consequently, the use of nanoprogramming saves a total of $52{,}450 - 30{,}550 = 21{,}850$ bits of control storage (42 percent of S_1). In general, two levels of control memory require less memory space if $S_2 < S_1$. Hence from (5.19) and (5.21), we conclude that the inequality

$$N \geq \lceil \log_2 H_m \rceil + \lceil \log_2 r \rceil + rN$$

must be satisfied.

5.3 PIPELINE CONTROL

Pipelining provides a basic way to speed up arithmetic operations, as we saw in Chapter 4. It is also used to implement the entire instruction-processing behavior of high-performance CPUs, a topic we examine in this section.

5.3.1 Instruction Pipelines

During program execution, instructions pass through a sequence of processing steps that lend themselves naturally to pipelining. Consequently, a CPU can be organized as one or more pipelines, whose various stages fetch opcodes and operands, execute instructions, and store results in local registers or external memory. In general, an *instruction pipeline* is a multifunction, reconfigurable pipeline designed to speed up a computer's performance by efficiently overlapping the processing of instructions. Such pipelines are contrasted with *arithmetic pipelines* of the type covered in section 4.3.2, which can, however, be built into instruction pipelines to implement the execution stages. An instruction pipeline is normally invisible to programmers and managed automatically by program compilers and by the CPU's internal program-control unit. Instruction pipelines were first used in the IBM 7030 (also known as Stretch) and a few other computers of the 1960s. They reemerged in the 1980s as key contributors to the high performance achieved by RISCs. Instruction pipelining has also been successfully incorporated into CISCs such as the 80X86/Pentium series, beginning with the 80486 microprocessor in 1989.

Pipeline structure. The general structure of a pipeline of m stages $S_1, S_2, \ldots, S_m$ appears in Figure 5.53 (which repeats Figure 4.47). When S_i has computed its results, it passes them, along with any unprocessed input operands, to S_{i+1} for further processing, and S_i receives a new set of operands from S_{i-1}. Thus the pipeline can contain up to m independent data sets, all in different stages of computation. Buffer registers and other synchronization logic are placed between stages so that the stages do not interfere with one another. The performance speedup of an instruction pipeline derives from the fact that up to m independent instructions can be in progress simultaneously in the m stages.

The simplest instruction pipeline breaks instruction processing into two parts: a fetch stage S_1 and an execute stage S_2. Thus a two-stage pipeline increases

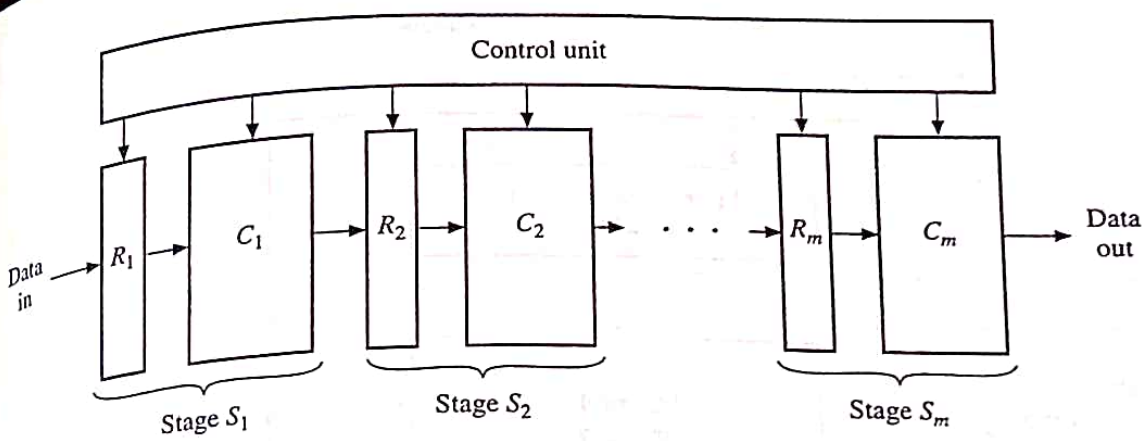


Figure 5.53
Structure of an m-stage pipeline.

throughput by overlapping instruction fetching and instruction execution. While instruction I_i with address A_i is being executed by stage S_2, the instruction I_{i+1} with the next consecutive address A_{i+1} is fetched from memory by stage S_1. If on executing I_i in S_2 it is determined that a branch must be made to a nonconsecutive address $A_j \neq A_{i+1}$, then the prefetched instruction I_{i+1} in S_1 has to be discarded. As we will see later, techniques exist to minimize the negative effect of branch instructions on pipeline performance.

Figure 5.54 shows an implementation of a two-stage instruction pipeline that is common in microprogrammed CPUs. It is the generic microprogrammed control unit of Figure 5.47 repackaged into two sequential stages. The fetch stage S_1 consists of the microprogram counter μPC, which is the source for microinstruction addresses, and the control memory CM, which stores the microinstructions. (CM is sometimes considered to lie outside the pipeline proper, with the task of feeding microinstructions "into" the pipeline.) Observe how μPC is appropriately positioned to be the buffer register for S_1. It is only necessary to increment μPC to obtain the next consecutive microinstruction address, which is then fetched while the current microinstruction is being executed in stage S_2. The execution stage S_2 contains the microinstruction register μIR, the decoders that extract control signals from the microinstructions in μIR, and the logic for choosing branch addresses. Another preexisting register, this time μIR, acts as the buffer register for stage S_2. Microinstruction execution is much simpler than the corresponding task at the instruction level. It involves decoding the control and condition-select fields of the current microinstruction μI stored in μIR, as well as distributing the resulting control signals. If μI specifies branching, the branch address is obtained directly from μI itself and fed back to S_1. There the branch address is loaded into μPC, replacing μPC's previous contents and causing any ongoing fetch operation in S_1 to be aborted.

Multistage pipelines. An m-stage instruction pipeline can overlap the processing of up to m instructions, so it is desirable to use more than two stages to maximize instruction throughput. The value of m depends on the maximum number of stages into which instruction processing can be efficiently broken. This number in turn depends on the complexity of the instruction set, the organization

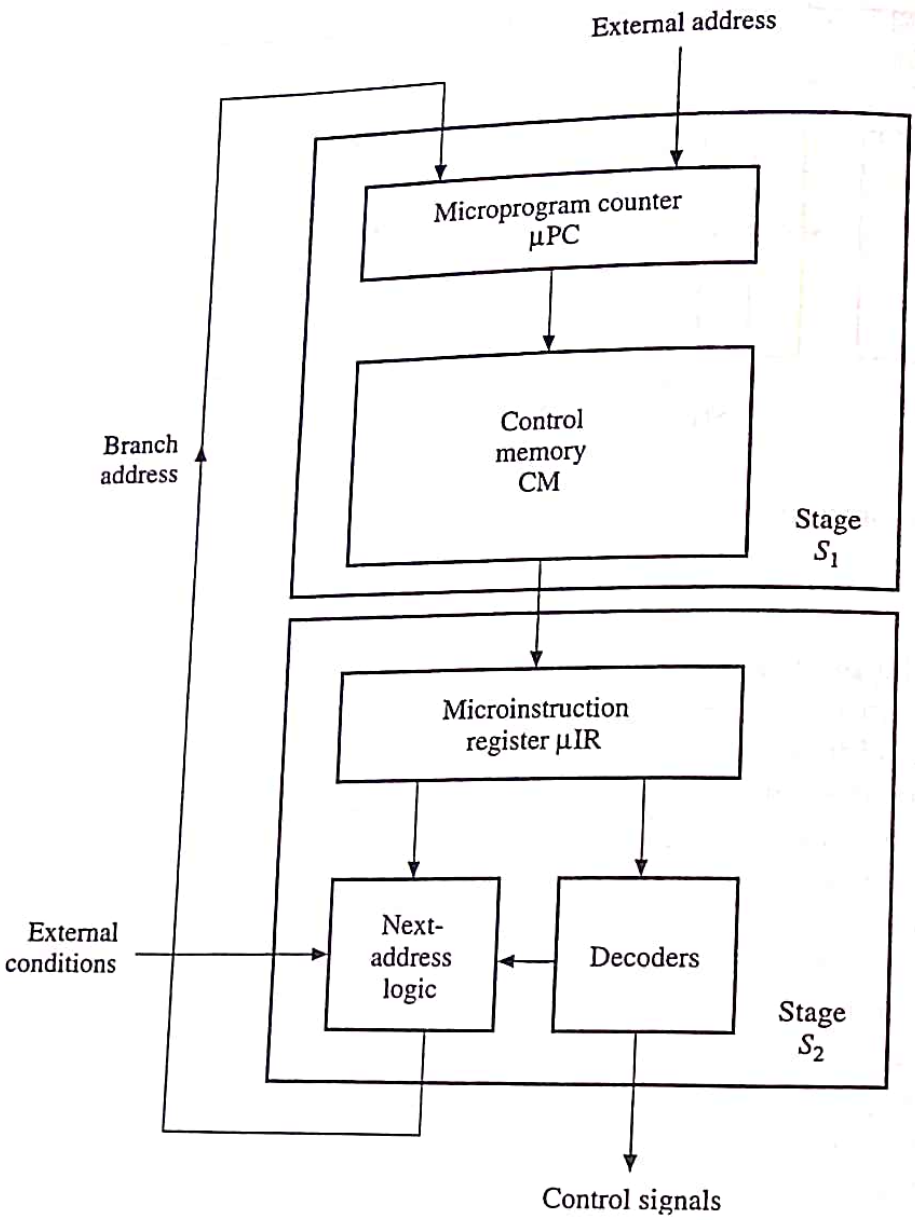


Figure 5.54
Two-stage pipelined microprogram control unit.

of the external memory M, and the way in which the CPU's datapath is implemented. In practice, the number of pipeline stages ranges from three (in the case of the ARM6) to a dozen or more. Pipeline structure is complicated by the provision of alternative (parallel) stages, feedback paths, and feedforward (bypass) features.

Figure 5.55 shows a CPU organization that implements a four-stage instruction pipeline. We assume that the CPU is directly connected to a cache memory, which is split into instruction and data parts, called the I-cache and D-cache, respectively. This splitting of the cache permits both an instruction word and a memory data word to be accessed in the same clock cycle. Each stage makes use of certain common resources such as the cache and the register file RF, which can be regarded as external to the pipeline proper. The four stages S_1:S_4 of Figure 5.55 perform the following functions:

1. IF: instruction fetching and decoding using the I-cache.
2. OL: operand loading from the D-cache to RF.

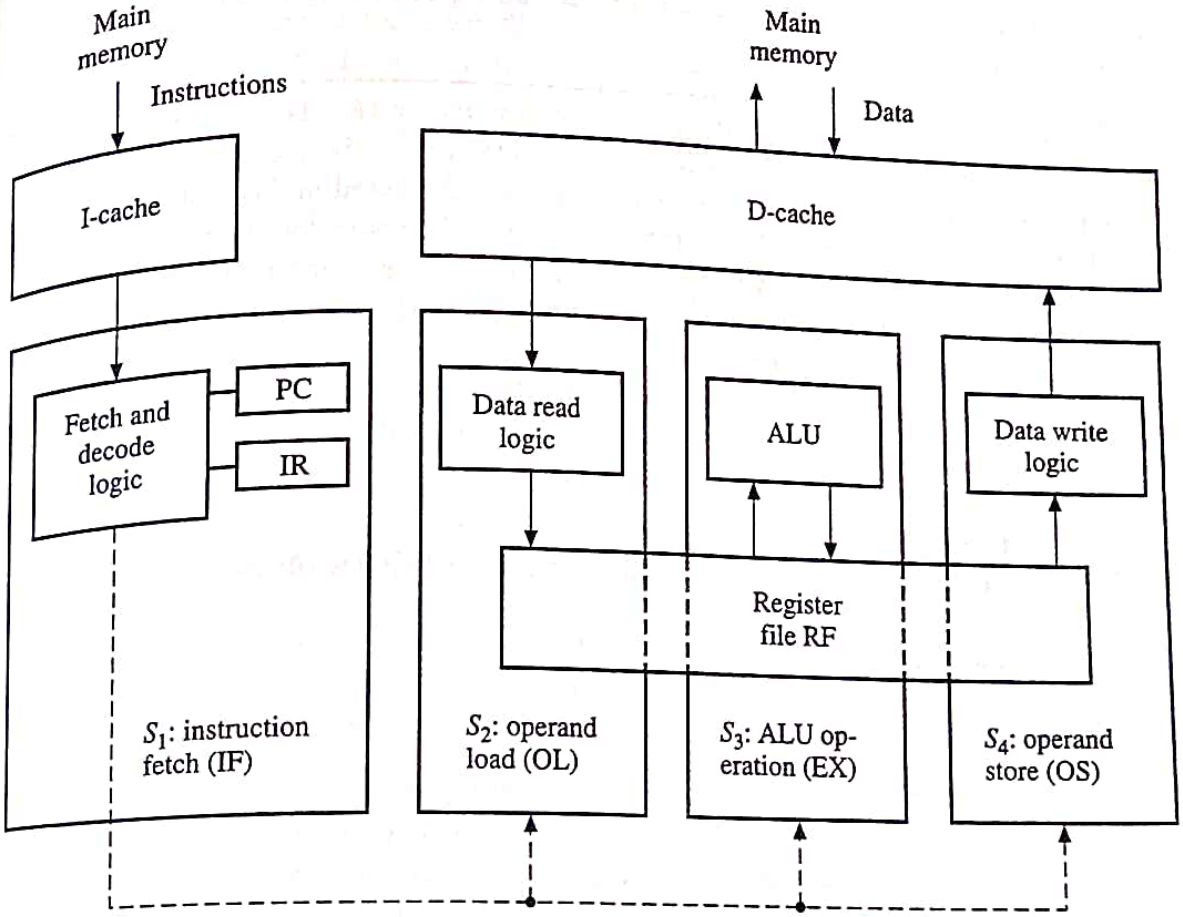


Figure 5.55
Organization of a CPU incorporating a four-stage instruction pipeline.

3. EX: data processing using the ALU and RF.
4. OS: operand storing to the D-cache from RF.

Stages S_2 and S_4 implement memory load and store operations, respectively, and are tailored to a load-store architecture. Stages S_2, S_3, and S_4 share the CPU's local registers in RF; these registers act as interstage buffer registers. The CPU's ALU is in stage S_3 and implements data-transfer and data-processing operations of the register-to-register type. If each stage completes its operation in a single CPU clock cycle of period T_C, the pipeline and the CPU as a whole can be clocked at a frequency of $f = 1/T_C$. At its maximum execution rate, which implies that no delays occur due to instruction branching, cache misses, or other causes, an ideal performance level of 1 clock cycle per instruction, or a *CPI* of 1, can be achieved.

We can vary the organization shown in Figure 5.55 in many ways to trade hardware cost for performance. For example, a less expensive D-cache cannot perform loads and stores simultaneously, in which case we can implement D-cache accesses in a single stage, thus merging S_2 and S_4 into a single load-store stage. Memory or register-file accesses are complicated by addressing modes such as indexing, which require an ALU to calculate a memory address before the access operation proper can be initiated. In such cases it may be desirable to add a stage, that is, a separate clock cycle, for operand address calculation. Instructions such as the more complicated arithmetic operations require multiple clock cycles for their execution; hence they require multiple cycles through the execution stage of a pipelined CPU. Such considerations, and the hardware/performance trade-offs they

ise to the many different instruction pipeline organizations in contem-
)uters. ·

'LE 5.7 PIPELINE ORGANIZATION OF THE MIPS R2/3000

[KANE EINRICH 1992]. The R2000 and R3000 are early members of the MIPS) series of RISC microprocessors, which we discussed in Examples 3.5 and 3.7. mplement the same MIPS-I instruction-set architecture and have nearly identical organizations, so we will treat them as a single machine denoted by R2/3000. members of the same series have numerous architectural extensions and far more)lex instruction pipelines.

The R2/3000 employs a five-stage instruction pipeline whose stages have the fol- ng functions designed to meet the goal of completing one instruction per clock .e:

IF: instruction fetching using the I-cache.
RD: operand loading (reading) from the register file RF while decoding the fetched instruction.
EX: data processing using the ALU and RF as needed.
. MA: operand accessing (load or store) using the D-cache.
. WB: operand storing (writing back) to RF.

Comparing this pipeline organization with that of Figure 5.55, we see that the first and third stages are roughly the same. The R2/3000's instruction-fetch (IF) stage is compli- cated by the use of virtual memory, which requires that the (virtual) addresses appear- ing in the input instruction stream be translated on the fly into physical addresses corresponding to the available main memory. Consequently, instruction decoding is deferred to the second stage of the pipeline. This operand-read (RD) stage also trans- fers any needed input operands from the CPU's 32-word register file RF in preparation for execution in stage 3 (EX). All memory data accesses (D-cache loads and stores) use stage 4 (MA), which transfers a data word between the CPU and the D-cache. The fifth or "write back" (WB) stage is used by load instructions to write a word fetched from the data cache into RF. The result of an ALU operation is also stored in RF during the WB stage.

Like other RISCs, the R2/3000 aims at single-cycle execution of its instructions. Figure 5.56 shows the ideal situation when, after a start-up phase during which it fills up, the instruction pipeline is fully utilized and outputs a new result every clock cycle.

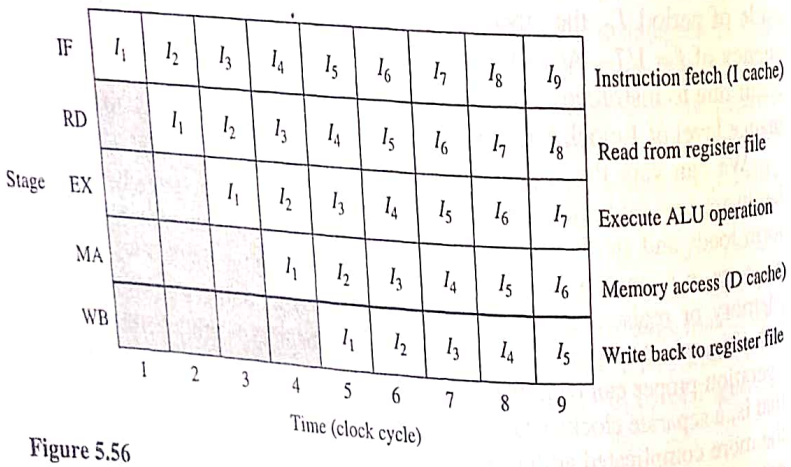


Figure 5.56
Maximum-rate instruction execution in the R2/3000 instruction pipeline.

If, in this "streaming" mode of operation, an instruction computes a new result in clock cycle i during the instruction's EX phase, that result can be used by another instruction in cycle $i + 1$. In some cases, notably load and branch instructions, this is not true, and delays occur due to the effects of an instruction on a subsequent one. For example, suppose that an instruction that loads a data word X into a register is immediately followed by an instruction that uses X in its EX stage. Then, as illustrated in Figure 5.57*a*, a one-cycle gap or *delay slot* occurs in the instruction stream because an LD (load) instruction's data is not available until after its MA cycle.

Several actions can be taken to deal with this situation:

- The pipeline can be temporarily halted or *stalled* whenever a load or branch instruction is executed. This action, however, complicates control of the pipeline and the synchronization of CPU operation and causes a loss in performance.
- A NOP (no operation) instruction can be inserted as shown in Figure 5.57*b*, which has the effect of synchronizing the issuing and execution of all instructions, none of which now needs to be delayed. This action does not improve the pipeline's performance, however.
- A nearby instruction that does not depend on X can be taken and repositioned in the instruction stream—which requires a smart compiler—immediately after the LD instruction. This approach is illustrated in Figure 5.57*c*, where the SUB instruction has been moved to fill LD's delay slot. Restructuring of this type is valid only if it does not alter the program's final results; for example, it requires that SUB not use the data fetched by LD as an input operand. The net effect is to make the pipeline operate at its maximum rate and to complete the four indicated instructions using one cycle fewer than before.

A similar delay problem arises in the case of branch instructions. The branch address computed by an R2/3000 branch instruction I does not become available for use until I's third (EX) stage, which creates a delay slot in I's second (RD) stage. Another instruction falling into this delay slot is executed, regardless of whether the branch is taken or not. Consequently, a compiler inserts a NOP into this slot unless, as in Figure 5.57*c*, the delay slot can be filled in some useful way that does not change the program's overall behavior.

Figure 5.58 summarizes the structure of another multistage instruction pipeline, that of the Amdahl 470V/7, a 1978-vintage machine designed to be compatible with the IBM System/370 series of mainframe computers [Amdahl 1978]. The 470V/7's memory system comprises a main memory and a single or *unified* cache (termed the high-speed buffer in Amdahl literature) intended for both instruction and data storage. The CPU is partitioned into a 12-stage pipeline, whose stages have the roles listed in Figure 5.58. These perform the same four functions as the generic pipeline of Figure 5.55, namely, instruction fetching and decoding (IF), operand loading (OL), instruction execution (EX), and, finally, operand storage (OS). Because of the many addressing modes and instruction types needed to support the 470V/7's CISC architecture, each of the preceding functions is subdivided into several pipeline stages. The first two stages S_1 and S_2 communicate with a memory control unit that is responsible for all accesses to main memory and the cache. These stages transfer instructions or data operands between the pipeline and the cache. All results are checked for errors in stage S_{11} using parity-check codes in most cases. If an error is detected, the instruction in question is automatically re-executed, an error-recovery technique called *instruction retry*.

entail, give rise to the many different instruction pipeline organizations in contemporary computers.

EXAMPLE 5.7 PIPELINE ORGANIZATION OF THE MIPS R2/3000 [KANE AND HEINRICH 1992]. The R2000 and R3000 are early members of the MIPS RX000 series of RISC microprocessors, which we discussed in Examples 3.5 and 3.7. They implement the same MIPS-I instruction-set architecture and have nearly identical CPU organizations, so we will treat them as a single machine denoted by R2/3000. Later members of the same series have numerous architectural extensions and far more complex instruction pipelines.

The R2/3000 employs a five-stage instruction pipeline whose stages have the following functions designed to meet the goal of completing one instruction per clock cycle:

1. IF: instruction fetching using the I-cache.
2. RD: operand loading (reading) from the register file RF while decoding the fetched instruction.
3. EX: data processing using the ALU and RF as needed.
4. MA: operand accessing (load or store) using the D-cache.
5. WB: operand storing (writing back) to RF.

Comparing this pipeline organization with that of Figure 5.55, we see that the first and third stages are roughly the same. The R2/3000's instruction-fetch (IF) stage is complicated by the use of virtual memory, which requires that the (virtual) addresses appearing in the input instruction stream be translated on the fly into physical addresses corresponding to the available main memory. Consequently, instruction decoding is deferred to the second stage of the pipeline. This operand-read (RD) stage also transfers any needed input operands from the CPU's 32-word register file RF in preparation for execution in stage 3 (EX). All memory data accesses (D-cache loads and stores) use stage 4 (MA), which transfers a data word between the CPU and the D-cache. The fifth or "write back" (WB) stage is used by load instructions to write a word fetched from the data cache into RF. The result of an ALU operation is also stored in RF during the WB stage.

Like other RISCs, the R2/3000 aims at single-cycle execution of its instructions. Figure 5.56 shows the ideal situation when, after a start-up phase during which it fills up, the instruction pipeline is fully utilized and outputs a new result every clock cycle.

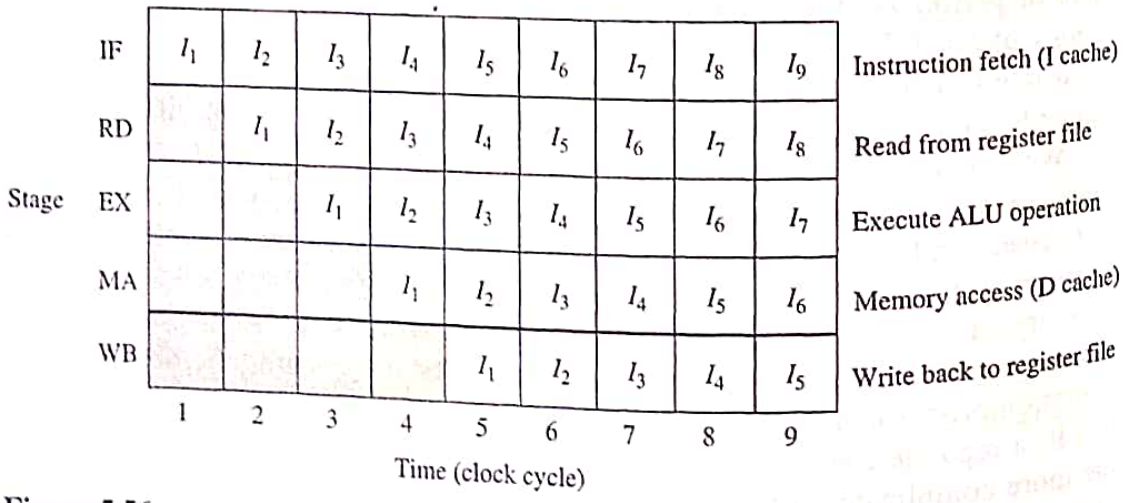


Figure 5.56
Maximum-rate instruction execution in the R2/3000 instruction pipeline.

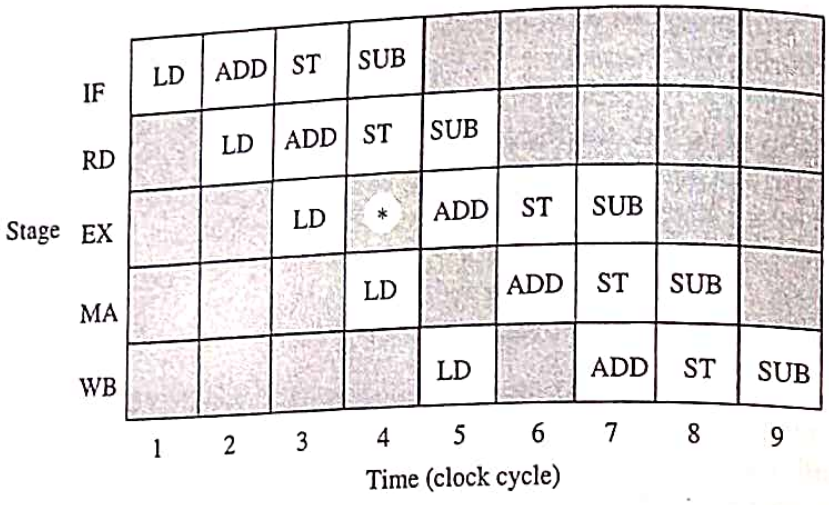
If, in this "streaming" mode of operation, an instruction computes a new result in clock cycle i during the instruction's EX phase, that result can be used by another instruction in cycle $i + 1$. In some cases, notably load and branch instructions, this is not true, and delays occur due to the effects of an instruction on a subsequent one. For example, suppose that an instruction that loads a data word X into a register is immediately followed by an instruction that uses X in its EX stage. Then, as illustrated in Figure 5.57*a*, a one-cycle gap or *delay slot* occurs in the instruction stream because an LD (load) instruction's data is not available until after its MA cycle.

Several actions can be taken to deal with this situation:

- The pipeline can be temporarily halted or *stalled* whenever a load or branch instruction is executed. This action, however, complicates control of the pipeline and the synchronization of CPU operation and causes a loss in performance.
- A NOP (no operation) instruction can be inserted as shown in Figure 5.57*b*, which has the effect of synchronizing the issuing and execution of all instructions, none of which now needs to be delayed. This action does not improve the pipeline's performance, however.
- A nearby instruction that does not depend on X can be taken and repositioned in the instruction stream—which requires a smart compiler—immediately after the LD instruction. This approach is illustrated in Figure 5.57*c*, where the SUB instruction has been moved to fill LD's delay slot. Restructuring of this type is valid only if it does not alter the program's final results; for example, it requires that SUB not use the data fetched by LD as an input operand. The net effect is to make the pipeline operate at its maximum rate and to complete the four indicated instructions using one cycle fewer than before.

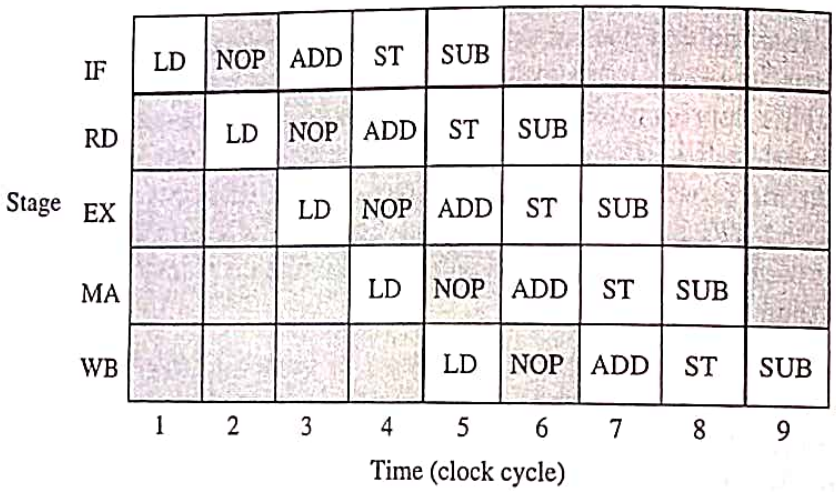
A similar delay problem arises in the case of branch instructions. The branch address computed by an R2/3000 branch instruction I does not become available for use until I's third (EX) stage, which creates a delay slot in I's second (RD) stage. Another instruction falling into this delay slot is executed, regardless of whether the branch is taken or not. Consequently, a compiler inserts a NOP into this slot unless, as in Figure 5.57*c*, the delay slot can be filled in some useful way that does not change the program's overall behavior.

Figure 5.58 summarizes the structure of another multistage instruction pipeline, that of the Amdahl 470V/7, a 1978-vintage machine designed to be compatible with the IBM System/370 series of mainframe computers [Amdahl 1978]. The 470V/7's memory system comprises a main memory and a single or *unified* cache (termed the high-speed buffer in Amdahl literature) intended for both instruction and data storage. The CPU is partitioned into a 12-stage pipeline, whose stages have the roles listed in Figure 5.58. These perform the same four functions as the generic pipeline of Figure 5.55, namely, instruction fetching and decoding (IF), operand loading (OL), instruction execution (EX), and, finally, operand storage (OS). Because of the many addressing modes and instruction types needed to support the 470V/7's CISC architecture, each of the preceding functions is subdivided into several pipeline stages. The first two stages S_1 and S_2 communicate with a memory control unit that is responsible for all accesses to main memory and the cache. These stages transfer instructions or data operands between the pipeline and the cache. All results are checked for errors in stage S_{11} using parity-check codes in most cases. If an error is detected, the instruction in question is automatically re-executed, an error-recovery technique called *instruction retry*

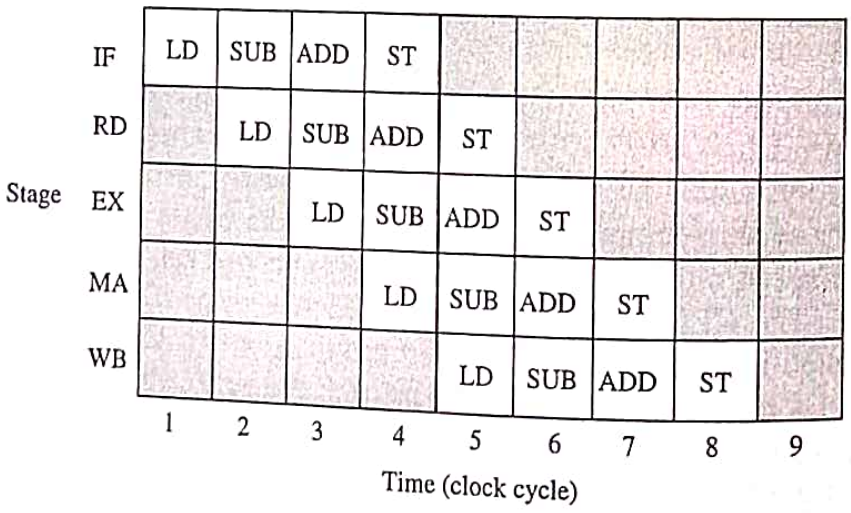


*Input operand of ADD unavailable

(*a*)



(*b*)



(*c*)

Figure 5.57
(*a*) R2/3000 pipeline delay slot caused by load instruction LD; (*b*) use of NOP instruction to fill the delay slot; (c) use of SUB instruction to eliminate the delay slot.

Function	Stage	Name	Action performed
Instruction fetch IF	S_1	Instruction address	Request next instruction from memory control unit
	S_2	Start buffer	Initiate cache to read instruction
	S_3	Read buffer	Read instruction from cache into I-unit
	S_4	Decode instruction	Decode opcode of instruction
Operand load OL	S_5	Read register	Read address (base and index) registers
	S_6	Compute address	Compute address of current memory operand
	S_7	Start buffer	Initiate cache to read memory operand
	S_8	Read buffer	Read operands from cache and register file
Execute instruction EX	S_9	Execute 1	Pass data to E-unit and begin instruction execution
	S_{10}	Execute 2	Complete instruction execution
Operand store OS	S_{11}	Check result	Perform code-based error check on result
	S_{12}	Write result	Store result

Figure 5.58
The stages of the Amdahl 470V/7 instruction pipeline.

In recent years the large number of pipeline stages illustrated by the 470V/7 have become common, because such fine-grained stages enable a pipeline to operate at higher clock frequencies. Multiple-instruction pipelines are also common, especially in *superscalar* processors, which can issue (dispatch) two or more instructions simultaneously. For example, each of the three functional units (E-units) of the PowerPC 601 microprocessor (Example 1.7) is implemented as a distinct pipeline; the structure and relationship of these pipelines are outlined in Figure 5.59 [Becker et al. 1993]. The 601 has an instruction buffer or queue that stores up to eight instructions which are prefetched from the single (unified) cache memory. In each clock cycle this buffer can send a separate instruction to each of the pipelined E-units. The two-stage branch-processing unit fetches and processes branch instructions. The five-stage fixed-point unit processes fixed-point ALU operations and also handles cache data accesses both for itself and for the floating-point unit. Some operations, such as multiply and divide, circulate repeatedly through the execute stage. The floating-point unit supports a full range of floating-point instructions, including a compound multiply-and-add instruction.

5.3.2 Pipeline Performance

The goal in controlling a pipelined CPU is to maximize its performance with respect to target workloads. After reviewing the performance measures applicable to instruction pipelines, we consider the factors that reduce performance and how they can be overcome.

Performance measures. A pipeline's performance can be measured by its *throughput* in terms of millions of instructions executed per second or *MIPS*. Another popular measure of performance is the number of clock cycles per

instruction or *CPI*. These quantities are related by the equation

$$CPI = f/MIPS \tag{5.22}$$

where f is the pipeline's clock frequency in MHz, and the values of *CPI* and *MIPS* are average figures that can be determined experimentally by processing suites of representative programs (benchmarks). The maximum value of *CPI* for a single pipeline is one, making the pipeline's maximum possible throughput equal to f. This throughput is attained only when the pipeline is supplied with a continuous stream of instructions that keep all its stages busy. Superscalar machines reduce *CPI* below one by executing several instruction streams simultaneously using multiple pipelines.

Figures 5.56 and 5.57 illustrate a useful way to visualize pipeline behavior called a *space-time diagram*, which shows the utilization of each pipeline stage as a function of time. In general, a space-time diagram for an m-stage pipeline has the form of an $m \times n$ grid, where n is the number of clock cycles to complete the processing of some sequence of N instructions of interest. Figure 5.60 shows a space-time diagram for the four-stage arithmetic pipeline of Example 4.8, which is executing a complex vector summation instruction denoted I. An unshaded box in these figures marks a busy stage S_i, and the box's entry denotes the particular instruction being processed by S_i. As the shading shows, some stages are not utilized at the beginning and end of the instruction sequence, when the pipeline must be filled and emptied (flushed), respectively. The stages are also underutilized if operands are not available when needed. The ratio of the unshaded (busy) area to

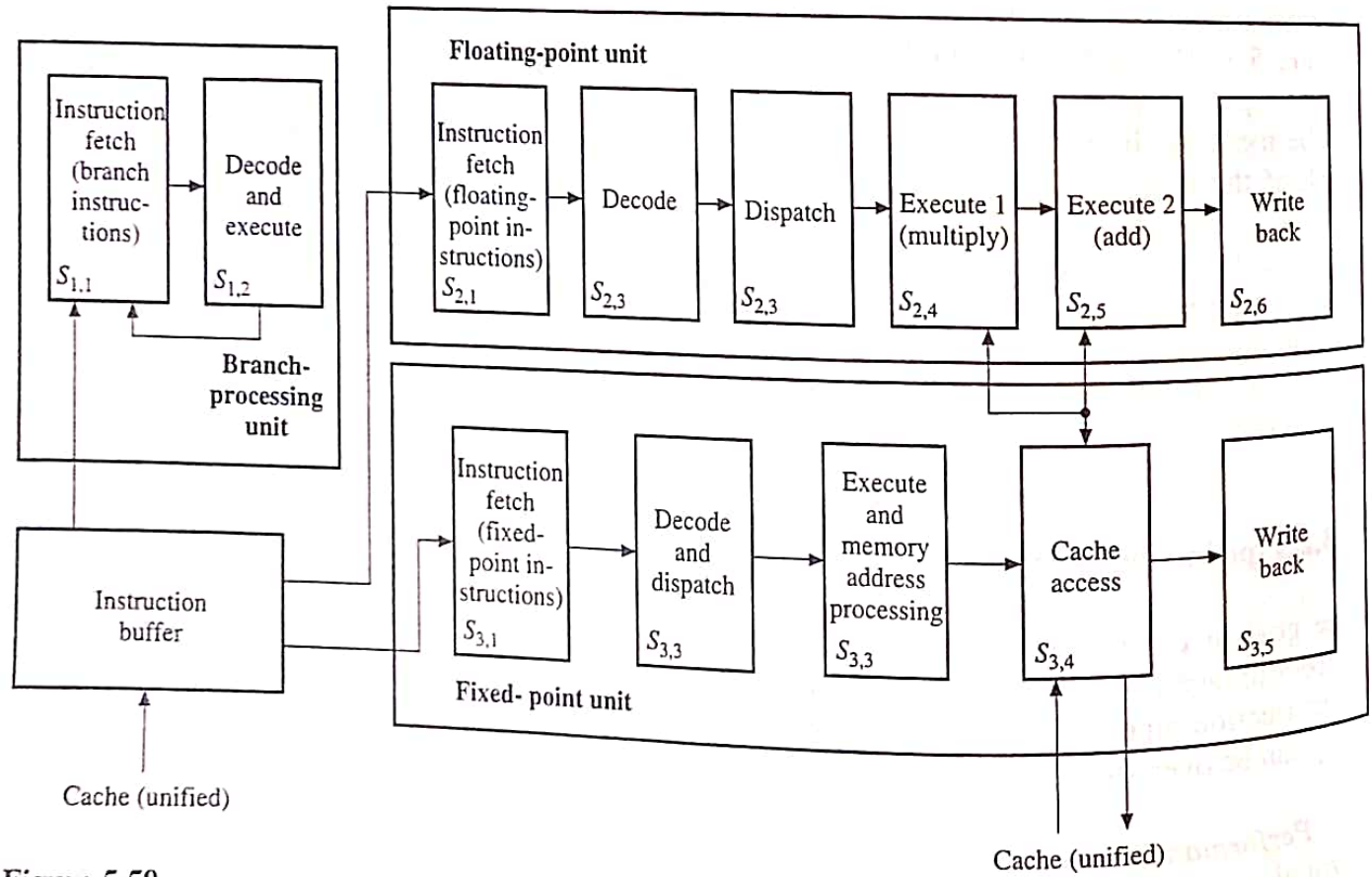


Figure 5.59
Instruction pipelining in the PowerPC 601.

the total (shaded and unshaded) area of a space-time diagram for an m-stage pipeline is defined as the *efficiency* or *utilization* $E(m)$ of the pipeline. In other words, $E(m)$ is the fraction of time the pipeline is busy. In the case of Figure 5.60, the efficiency is $E(4) = 44/76 = 0.58$. Note how the instruction reordering shown in Figure 5.57*c* improves the pipeline's efficiency by eliminating the delay slot.

Another general measure of pipeline performance is the *speedup* $S(m)$ defined by

$$S(m) = \frac{T(1)}{T(m)} \tag{5.23}$$

where $T(m)$ is the execution time for some target workload on an m-stage pipeline and $T(1)$ is the execution time for the same workload on a similar, nonpipelined processor. It is reasonable to assume that $T(1) \leq mT(m)$, in which case $S(m) \leq m$. A pipeline's efficiency and speedup are related as follows:

$$S(m) = m \times E(m) \tag{5.24}$$

Hence for the example in Figure 5.60 where $m = 4$ and $E(4) = 0.58$, the speedup $S(4) = 4 \times 0.58 = 2.32$ and cannot exceed 4. In general, speedup and efficiency provide rough performance estimates which should be used with caution, since they depend on the programs being run. Their values can change drastically from program to program, or from one part of a program to another.

Optimizing m. Equation (5.24) suggests that an easy way to improve a pipeline's performance is to increase the number of stages m. This assumes that the pipeline's processing tasks can be subdivided in a useful way and that the cost of doing so is acceptable. Each new stage S_i introduces some new hardware cost and delay due to its buffer register R_i and associated control logic. We now analyze the trade-offs involved in doing this [Kogge 1981; Hwang 1993]. In particular, we will determine the pipeline's *performance/cost ratio PCR* defined as

$$PCR = \frac{f}{K} \tag{5.25}$$

where f is the pipeline's clock frequency and K is its hardware cost.

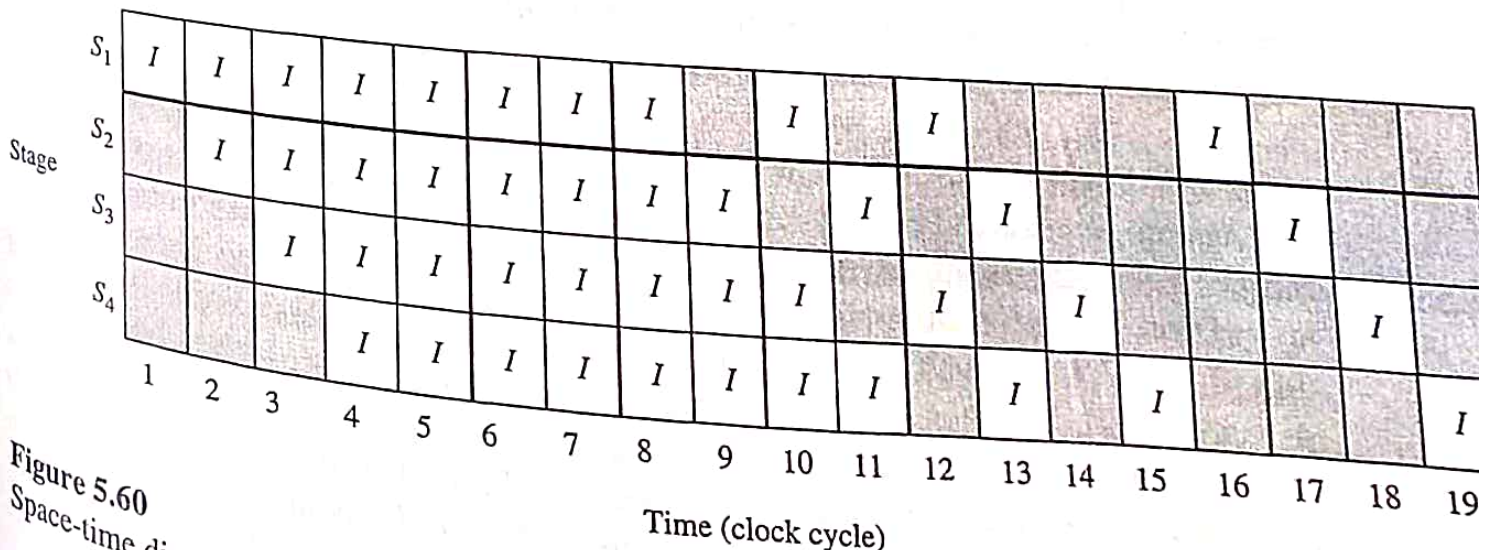


Figure 5.60
Space-time diagram for a four-stage pipeline.

Suppose the pipeline P has m stages and implements a particular set of operations (instructions) SI. Let a be the delay (latency) of an efficient, nonpipelined processor that also implements SI. It is reasonable to assume that each stage S_i of P has delay a/m—that is, m times less than the corresponding nonpipelined processor—plus some extra delay b due to S_i's buffer register R_i. Hence if $T_C = 1/f$ is P's clock period, we can write

$$T_C = a/m + b \tag{5.26}$$

The pipeline's hardware cost can be estimated by

$$K = cm + d \tag{5.27}$$

where c is the buffer-register cost per stage and d is the cost of the pipeline's (combinational) data-processing logic. Hence from (5.25), (5.26), and (5.27) we have $PCR^{-1} = T_C K = (a/m + b)(cm + d)$, so

$$PCR = m/[bcm^2 + (ac + bd)m + ad] \tag{5.28}$$

To maximize PCR with respect to the number of stages m, we differentiate (5.28) with respect to m and equate the result to zero. Using the standard differentiation-by-parts formula

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \frac{1}{v}\frac{du}{dx} - \frac{u}{v^2}\frac{dv}{dx}$$

we obtain

$$\frac{d}{dm}(PCR) = 1/v - m(2bcm + ac + bd)/v^2 \tag{5.29}$$

where $u = m$ and $v = bcm^2 + (ac + bd)m + ad$. On equating (5.29) to zero, we get $v = m(2acm + ad + bc)$. Substituting for v and solving for m yields the value m_{opt} of m that maximizes PCR, namely,

$$m_{opt} = \sqrt{\frac{ad}{bc}} \tag{5.30}$$

The optimum number of stages is the integer closest to m_{opt}. Figure 5.61 plots PCR against m according to (5.28) for $a = d = 5$ and $b = c = 1$. The optimum value of m is five, as predicted by (5.30). Hence in this instance, the maximum throughput per unit of hardware cost or, equivalently, the minimum cost per instruction processed, occurs when the pipeline has five stages.

Collisions. As discussed in section 4.3.2 pipelines can have feedback paths that enable a stage to be used repeatedly while processing a single instruction. In Figure 5.60, for example, each stage is used many times while processing the instruction I (vector summation). A new instruction of the same kind cannot be started until clock cycle 17, after I uses stage S_1 for the last time. If a second instruction is initiated at the wrong time—at $t = 9$, for instance—then both instructions will attempt to use stage S_1 at $t = 10$, a situation termed a *collision*. However, a simple add instruction of the kind in Figure 5.57 could be initiated at $t = 9$, 11, 13, 14, or 15 without colliding with the sum instruction I. Thus up to five add instructions, if available for execution at the right times, could be interleaved with the eight-element sum operation, thereby increasing the pipeline's overall efficiency.

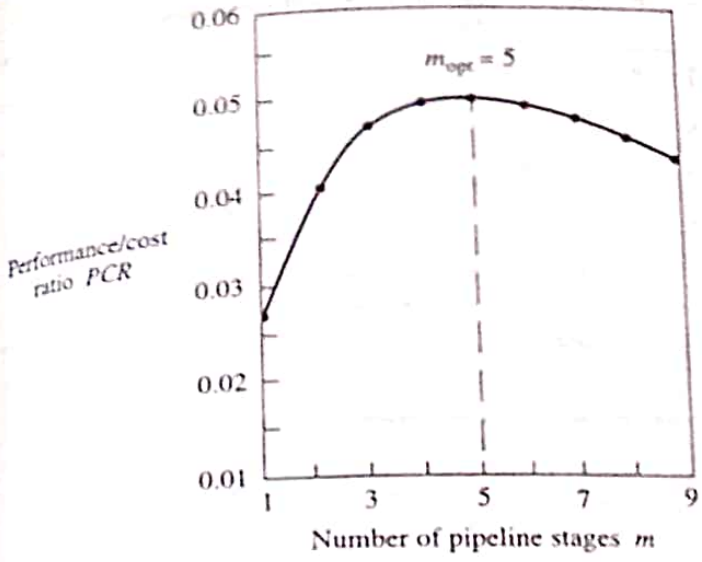


Figure 5.61
Performance/cost ratio *PCR* for an *m*-stage pipeline.

In general, pipeline collisions of the foregoing type are avoided by carefully scheduling the times at which new pipeline operations are initiated. We present a pipeline control strategy to avoid collisions and maximize performance in a pipeline with feedback or feedforward connections [Kogge 1981; Stone 1993]. Let P be such a pipeline that consists of m stages $S_1, S_2, \ldots, S_m$ and executes an instruction of type I. (Later we will consider the problem of scheduling different types of instructions in the same pipeline.) We can represent I's usage of the pipeline with a space-time diagram (refer to Figure 5.60), which indicates stage usage in every clock cycle while I is being executed. We will also represent the same information in a slightly different form R called a *reservation table*. The m rows of R represent the stages of P, while the columns represent the sequence of clock cycles required for one complete execution of I by P. An × is placed at the intersection of row S_i and column C_j if stage S_i is used by I in clock cycle $t = j$. Figure 5.62*a* shows the reservation table corresponding to Figures 4.52 and 5.60. If the method of Figure 4.52 is used to sum the pair of numbers b_1, b_2, then the small reservation table of Figure 5.62*b* results.

Two operations of type I that are initiated k clock cycles apart collide at stage S_i of P, if row i of the corresponding reservation table R contains two ×s that are separated by a horizontal distance of k. In the case of Figure 5.62*b*, a collision occurs at every stage if $k = 1$, 4, or 5, as is easily verified. For example, if the first instruction is initiated at $t = 1$ and the second at $t = 5$, in which case $k = 4$, then both instructions will attempt to use all four stages and collide at $t = 6$. Let F be the set of numbers, called the *forbidden list* of R, whose entries are the distances, that is, the numbers of clock cycles between all distinct pairs of ×s in every row of R. The collision conditions for R are characterized by the following easily proven result: Two pipeline instructions initiated k clock cycles apart collide if and only if k is in the forbidden list F of R. Thus we can easily meet the fundamental requirement of avoiding collisions by delaying new instructions by time periods *not* appearing in the forbidden list. Much less obvious is how to schedule initiation times that maximize the pipeline's performance.

The maximum number of collision-free operations that can be initiated per unit time under steady-state conditions corresponds to the pipeline's throughput defined

	Time t																		
Stage	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
S_1	×	×	×	×	×	×	×	×		×		×				×			
S_2		×	×	×	×	×	×	×	×		×		×				×		
S_3			×	×	×	×	×	×	×	×		×		×				×	
S_4				×	×	×	×	×	×	×	×		×		×				×

(*a*)

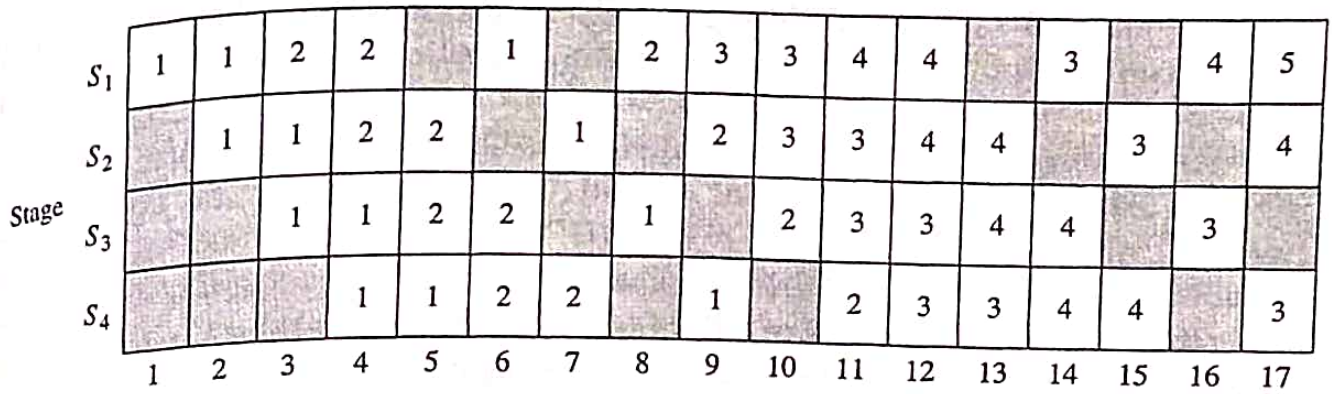
	Time t								
Stage	1	2	3	4	5	6	7	8	9
S_1	×	×				×			
S_2		×	×				×		
S_3			×	×				×	
S_4				×	×				×

(*b*)

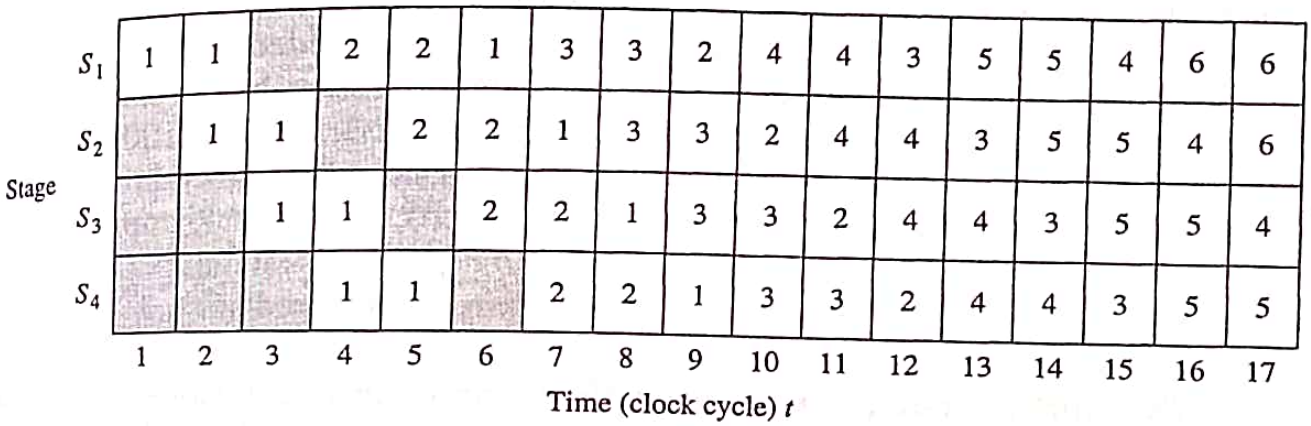
Figure 5.62
Pipeline reservation tables for N-element vector summation: (*a*) $N = 8$ corresponding to Figure 5.60 and (*b*) $N = 2$.

by Equation (5.22). The delay occurring between the start of two successive, collision-free pipeline instructions is called the *initiation latency,* or simply the *latency* L in this context. Under steady-state operating conditions, L corresponds to the pipeline's *CPI* and is measured in clock cycles. We now turn to the problem of devising control strategies that maximize the performance of a basic, single-function pipeline by determining the best values of L to use for collision-free operation.

We denote the minimum value of the initiation latency L, that is, the *minimum average latency* by L_{min}. A simpler goal is to achieve the *minimum constant latency,* defined as the smallest fixed value L_{cmin} of L such that any number of instructions can be initiated L clock cycles apart without causing collisions. Clearly, $L_{min} \leq L_{cmin}$. The number L_{cmin} can be calculated from the forbidden list F using the fact that L_{cmin} is the smallest integer L such that hL is not in F for any integer $h \geq 1$. The forbidden lists for the reservation tables of Figures 5.62*a* and 5.62*b* are {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15} and {1, 4, 5}, respectively. Thus, as observed earlier, successive sum instructions with the reservation table of Figure 5.62*a* must be initiated at least 16 clock cycles apart, since the latencies L_{min} and L_{cmin} are both 16. In the case of Figure 5.62*b*, new instructions can be initiated as few as two cycles apart. However, the minimum constant latency $L_{cmin} \neq 2$, because $2 \times 2 = 4$ is in F; in this case $L_{cmin} = 3$. If instructions are initiated at $t = 1$ and 3, a third instruction cannot be initiated until $t = 9$, as demonstrated by the space-time diagram of Figure 5.63*a*. The average initiation latency for the pipeline scheduling scheme defined by Figure 5.63*b* is four, because two new instructions



(*a*)



(*b*)

Figure 5.63
Pipeline scheduling strategies for the reservation table of Figure 5.62*b*: (*a*) nonoptimal and (*b*) optimal.

are initiated every eight clock cycles. The minimum average latency is achieved for this example when new operations are initiated $L_{\min} = 3$ clock cycles apart, as in Figure 5.63*b*. Observe that in the latter case the steady-state efficiency of the pipeline is 100 percent.

Control scheme. An elegant way to control a pipeline for collision-free operation is by computing collision vectors. A *collision vector* CV for a reservation table R at time t is a binary vector $c_1c_2\cdots c_{M-1}c_M$, where the *ith* bit c_i is 1 if initiating a pipeline instruction at $t + i$ results in a collision; c_i is 0 otherwise. An *initial collision vector* CV_0 is obtained from the forbidden list F of R as follows. Element c_i of CV_0 is set to 1 if i is in F, and c_i is set to 0 otherwise, for $i = 1,2,\ldots,M$, where M is the maximum element in F. A convenient way to store CV is in a shift register $CR = CR_1{:}CR_M$ called a *collision register*. By inspecting CR_1 at time t, we can determine whether issuing a new instruction in the next clock cycle $t + 1$ will result in a collision. A simple left shift of CR, with the right-most bit CR_M set to 0 prepares CR_1 for inspection in the next clock cycle. If we decide to initiate a new instruction at $t + 1$, then CR is left shifted and its contents are replaced by CR *or* CV_0, where CV_0 is the initial collision vector obtained from F as specified above

and *or* denotes the bitwise OR operation. These actions ensure that CR defines all the collision possibilities due either to ongoing pipeline operations or to the newly initiated one.

To illustrate the foregoing concepts, consider again the reservation table in Figure 5.62*b*. Since $F = \{1, 4, 5\}$, $M = 5$ and the corresponding initial collision vector CV_0 is 10011. The collision register CR is initialized to 00000. When it is decided, say, at $t = 0$, to start the first pipeline instruction at $t = 1$, CR is left shifted and ORed with CV_0, resulting in CR = 00000 *or* 10011 = 10011 = CV_0. At $t = 1$ the new pipeline instruction is initiated, and CR is again inspected. Since $CR_1 = 1$, we conclude that a new instruction must be delayed; CR is merely shifted during this cycle, changing its contents to 00110. At $t = 2$, CR contains 00110 with $CR_1 = 0$, allowing a new instruction to start at $t = 3$. If a second pipeline operation is initiated in the next cycle, CR is shifted and ORed with CV_0, therefore becoming 01100 *or* 10011 = 11111 at $t = 3$. Five subsequent shifts are needed before CR_1 again becomes zero at $t = 8$. A third instruction cannot therefore begin until $t = 9$, as shown by Figure 5.63*a*. If no new task is started at $t = 3$, CR is 01100, indicating that a new instruction can begin at $t = 4$. If a new instruction is then initiated at $t = 4$, CR becomes successively 11000 and 11000 *or* 10011 = 11011, implying that a third instruction can begin at $t = 7$; Figure 5.63*b* depicts this situation. Observe that at $t = 6$, CR becomes 01100, repeating the pattern encountered at $t = 3$.

Task-initiation diagram. We can derive an optimal collision-free schedule for initiating pipeline operations from the state behavior of the collision register CR. For this purpose we construct a condensed state-transition graph for CR called a *task-initiation diagram* (TID). The states of the TID are all the collision vectors $\{CV_i\}$ formed by the operation CR *or* CV_0, when new pipeline operations can be initiated. (The other states of CR are formed by shifting these vectors and are excluded from the TID.) An arrow from CV_i to CV_j indicates that there is a sequence of state transitions that changes CR's state from CV_i to CV_j; the arrow is labeled with the minimum number of state transitions n_{ij} required. Thus n_{ij} denotes the minimum latency between the initiations represented by the TID states CV_i and

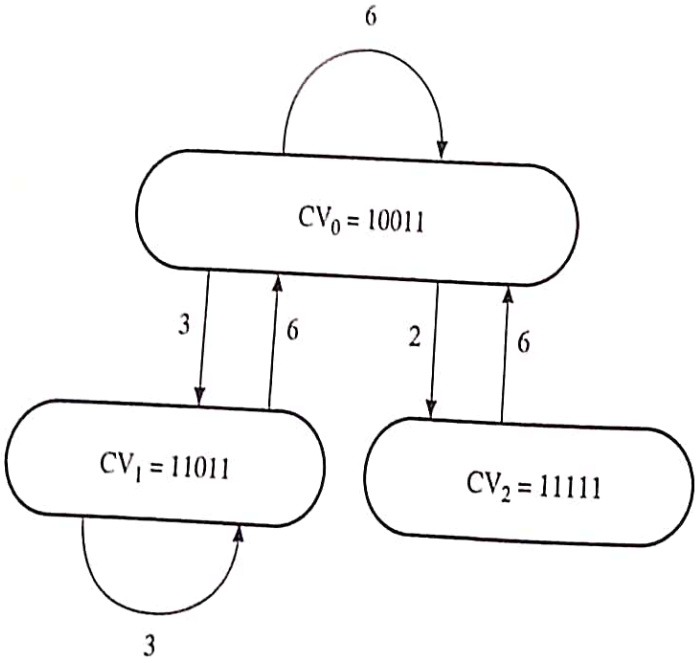


Figure 5.64
Task-initiation diagram (TID) for Figure 5.63*b*.

CV_j. A closed path or loop in the TID corresponds to the task initiation schedule for the pipeline that can be sustained indefinitely without collisions. Let s be the sum of the n_{ij} labels along the arrows forming the loop divided by the number of arrows in the loop. Clearly, s is the average latency of the corresponding schedule of pipeline task initiations. Therefore, the average latency of the pipeline is minimized by choosing a task sequence corresponding to a loop of the TID with a minimum value of s, which is then the minimum average initiation latency L_{min} [Kogge 1981].

Figure 5.64 shows the TID derived from the reservation table of Figure 5.62*b*, where the initial collision vector $CV_0 = 10011$. This TID is obtained in straightforward fashion by examining all the possible states and state transitions of the corresponding CR as described earlier. The states included in the TID are all those loaded into SR when new operations can be initiated, namely, the three states $CV_0 = 10011$, $CV_1 = 11011$, and $CV_2 = 11111$ identified above. For example, the self-loop labeled 3 on state CV_1 is a consequence of the following sequence of state transitions involving CR:

Clock cycle	State of CR	Actions taken
t	$11011 = CV_1$	Initiate new instruction. Left shift CR.
$t+1$	10110	Left shift CR.
$t+2$	01100	Select new instruction to initiate. Left shift CR. CR := CR *or* $CV_0 = 11000$ *or* 10011.
$t+3$	$11011 = CV_1$	Initiate new instruction. Left shift CR.

The TID of Figure 5.64 contains several loops corresponding to pipeline control strategies with different average initiation latencies. For example, the loop formed by the two arrows linking CV_0 and CV_2 has an average initiation latency of $(2 + 6)/2 = 4$ cycles and corresponds to the space-time diagram of Figure 5.63*a*. The self-loop of state CV_1 has the minimum average latency $L_{min} = 3$ and therefore maximizes pipeline performance; using this loop for pipeline control yields the space-time diagram of Figure 5.63*b*. The analysis confirms our previous observation that the optimum scheduling strategy for this example is to initiate a new instruction three cycles after the previous instruction. Hence a simple logic circuit based on a modulo-3 counter suffices to control this particular pipeline.

The progress of an instruction stream through a pipeline can be delayed by various unfavorable dependency relationships among instructions and their data operands, which are collectively referred to as *hazards*. We now define the main types of pipeline hazards and discuss some general ways to detect them and reduce their impact on performance.

Control dependencies. Conditional and unconditional jumps, subroutine calls, and other program-control instructions that involve branching can adversely affect the performance of an instruction pipeline. In these cases the address of the next instruction is not known with certainty until after the program-control instruction I has been executed. Hence the question arises: Which instructions should be

entered into the pipeline immediately after I? If these happen to be the wrong instructions, that is, I causes a jump to a distant part of the program, then provision must be made to cancel the effects of the partially executed instructions. This process is sometimes termed *flushing* the pipeline and clearly reduces its throughput. In the case of a two-way branch, it is sometimes worthwhile for the compiler or the pipeline's control logic to "guess" the direction of the branch, that is, to anticipate the outcome of the branch condition test, and enter the instruction at I's more likely target address into the pipeline immediately after I. This process is known as *speculative* execution. Pipeline flushing is then needed only when the wrong guess has been made.

We can estimate the influence of branch instructions on the performance of an instruction pipeline as follows. Suppose that the pipeline has m stages and that each instruction requires m clock cycles, corresponding to one complete pass through the pipeline. If there are no branch instructions in the instruction streams being processed, then an ideal throughput of one instruction per clock cycle is achieved; that is, $CPI = 1$. Let p be the probability of encountering a 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 flushed, destroying all ongoing instruction processing, when I emerges from the last stage (a pessimistic assumption).

Now consider an instruction sequence of length r that is streaming through the pipeline. The number of instructions causing branches to take place is pqr, and these instructions are executed at a rate of $1/m$ instructions per cycle. The remaining $(1 - pq)r$ nonbranching instructions are processed at the maximum rate of one instruction per cycle. Hence the total number of cycles n_c needed to process all r instructions is

$$n_c = pqrm + (1 - pq)r$$

This implies that the average CPI of the pipeline, which by definition is n_c/r, is given by

$$CPI = 1 + pq(m - 1) \tag{5.31}$$

with the optimum value $CPI = 1$ occurring when $q = 0$, that is, when no branching occurs during program execution. Note that a comparable nonpipelined instruction processor has $CPI = m$. If $p = 0.2$, $q = 0.4$, and $m = 5$, which are typical values for instruction pipelines, then (5.31) implies that $CPI = 1.32$. Hence, in this case, pipelining reduces the number of cycles per instruction from 5 to 1.32, an improvement by a factor of about four. The improvement is less for longer pipelines, since each branch to a nonconsecutive instruction address causes more partially processed instructions to be discarded. A compiler or programmer can increase throughput by employing fewer branch instructions (to reduce p) and by constructing conditional branch instructions so that the more probable results of the condition tests cause no branching (to reduce q).

Pipelined computers employ various hardware techniques to minimize the performance degradation due to branching. The Amdahl 470V/7, for example, has special branch-resolution logic to send the result of a branch condition test from the E-unit to the I-unit before the conditional branch instruction has been completely processed. This logic allows the I-unit to initiate processing of the correct next

instruction with a loss of data in only 3 of its 12 pipeline stages. A different approach is taken by the IBM 3033. Its cache is divided into three separate instruction buffer areas: One holds a normal sequence of consecutive instructions prefetched under the assumption that no branches will occur; the other two buffers hold prefetched instruction sequences starting at up to two branch addresses specified by previously decoded branch instructions. Thus when the 3033's CPU decodes an unconditional branch instruction of the form **go to** A_j, and has an instruction buffer with available space, it proceeds to prefetch and process instructions starting at location A_j. In the case of a two-way conditional branch instruction with two target branch addresses A_j and A_k, the CPU selects one branch address for prefetching. If, when the conditional branch instruction is subsequently executed, it turns out that the wrong selection was made by the CPU, then time is lost while the correct instruction is fetched. If the CPU has anticipated the outcome of the condition test correctly, then the required next instruction is either already in the instruction pipeline or is stored in an instruction buffer.

RISC machines rely on instruction pipelines that overlap instruction fetch and execute to achieve single-cycle execution for most instructions. As we saw in the case of the MIPS R2/3000 (Example 5.7), special measures are taken to nullify the delay slots associated with load and branch instructions. A closely related technique called *delayed branching* is used in some RISCs to reduce the penalty due to pipeline flushes on program branching. A delayed branch instruction I_1 causes the instruction I_2 immediately following I_1 to be executed while the instruction I' at the target address specified by I_1 is still being fetched. The execution of I' then follows that of I_2 rather than following that of I_1, as would normally be the case. For example, the IBM 801, the prototype RISC processor, has an alternative *branch-with-execute* form of every normal branch instruction [Radin 1983]. Thus the instruction sequence

```
LOAD  R1, A        (5.32)
BNZ   L
```

for the 801 containing the normal conditional branch instruction BNZ (branch if nonzero) idles the CPU while the instruction at the branch address L is being fetched. Suppose that BNZ is replaced by the corresponding branch-and-execute instruction BNZX and the instruction order is reversed as follows:

```
BNZX  L            (5.33)
LOAD  R1, A
```

The modified code (5.33) has the same meaning as (5.32), but now the LOAD instruction is executed while the instruction specified by BNZX is being fetched. The compiler of the 801 is able to translate about 60 percent of program branches into the more efficient branch-with-execute form.

Data dependencies. An m-stage pipeline operates at its maximum performance level when it contains m different instructions, each in a different stage of computation. As we have seen, problems can occur if the decision to execute a particular instruction depends on the outcome of an earlier branch instruction. This problem is due to the program's flow of control and so is called a *control dependency*. Other, more subtle *data dependencies* can exist among the operands being