CHAPTER

8

PIPELINING

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Pipelining as a means for executing machine instructions concurrently
- Various hazards that cause performance degradation in pipelined processors and means for mitigating their effect
- Hardware and software implications of pipelining
- Influence of pipelining on instruction set design
- Superscalar processors

The basic building blocks of a computer are introduced in preceding chapters. In this chapter, we discuss in detail the concept of pipelining, which is used in modern computers to achieve high performance. We begin by explaining the basics of pipelining and how it can lead to improved performance. Then we examine machine instruction features that facilitate pipelined execution, and we show that the choice of instructions and instruction sequencing can have a significant effect on performance. Pipelined organization requires sophisticated compilation techniques, and *optimizing compilers* have been developed for this purpose. Among other things, such compilers rearrange the sequence of operations to maximize the benefits of pipelined execution.

8.1 BASIC CONCEPTS

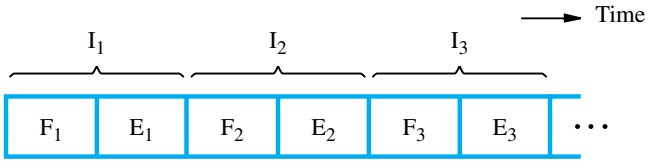
The speed of execution of programs is influenced by many factors. One way to improve performance is to use faster circuit technology to build the processor and the main memory. Another possibility is to arrange the hardware so that more than one operation can be performed at the same time. In this way, the number of operations performed per second is increased even though the elapsed time needed to perform any one operation is not changed.

We have encountered concurrent activities several times before. Chapter 1 introduced the concept of multiprogramming and explained how it is possible for I/O transfers and computational activities to proceed simultaneously. DMA devices make this possible because they can perform I/O transfers independently once these transfers are initiated by the processor.

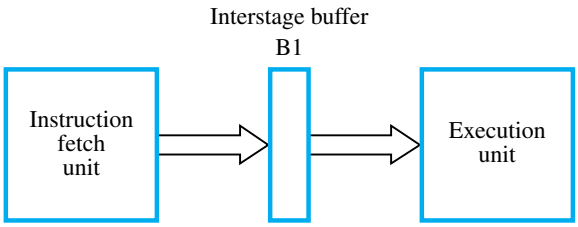
Pipelining is a particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly-line operation. Readers are undoubtedly familiar with the assembly line used in car manufacturing. The first station in an assembly line may prepare the chassis of a car, the next station adds the body, the next one installs the engine, and so on. While one group of workers is installing the engine on one car, another group is fitting a car body on the chassis of another car, and yet another group is preparing a new chassis for a third car. It may take days to complete work on a given car, but it is possible to have a new car rolling off the end of the assembly line every few minutes.

Consider how the idea of pipelining can be used in a computer. The processor executes a program by fetching and executing instructions, one after the other. Let F_i and E_i refer to the fetch and execute steps for instruction I_i. Execution of a program consists of a sequence of fetch and execute steps, as shown in Figure 8.1*a*.

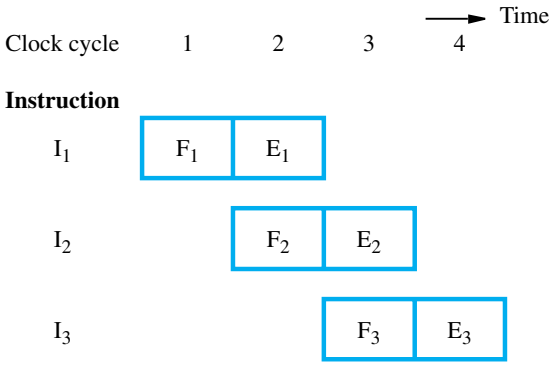
Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure 8.1*b*. The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1. This buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction. For the purposes of this discussion, we assume that both the source and the destination of the data operated on by the instructions are inside the block labeled "Execution unit."



(a) Sequential execution



(b) Hardware organization



(c) Pipelined execution

Figure 8.1 Basic idea of instruction pipelining.

The computer is controlled by a clock whose period is such that the fetch and execute steps of any instruction can each be completed in one clock cycle. Operation of the computer proceeds as in Figure 8.1*c*. In the first clock cycle, the fetch unit fetches an instruction I_1 (step F_1) and stores it in buffer B1 at the end of the clock cycle. In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction I_2 (step F_2). Meanwhile, the execution unit performs the operation specified by instruction I_1, which is available to it in buffer B1 (step E_1). By the end of the

second clock cycle, the execution of instruction I_1 is completed and instruction I_2 is available. Instruction I_2 is stored in B1, replacing I_1, which is no longer needed. Step E_2 is performed by the execution unit during the third clock cycle, while instruction I_3 is being fetched by the fetch unit. In this manner, both the fetch and execute units are kept busy all the time. If the pattern in Figure 8.1*c* can be sustained for a long time, the completion rate of instruction execution will be twice that achievable by the sequential operation depicted in Figure 8.1*a*.

In summary, the fetch and execute units in Figure 8.1*b* constitute a two-stage pipeline in which each stage performs one step in processing an instruction. An inter-stage storage buffer, B1, is needed to hold the information being passed from one stage to the next. New information is loaded into this buffer at the end of each clock cycle.

The processing of an instruction need not be divided into only two steps. For example, a pipelined processor may process each instruction in four steps, as follows:

F Fetch: read the instruction from the memory.

D Decode: decode the instruction and fetch the source operand(s).

E Execute: perform the operation specified by the instruction.

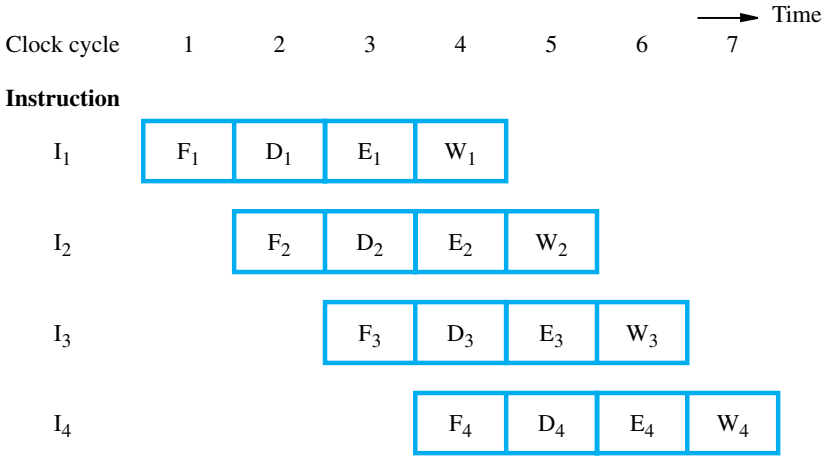
W Write: store the result in the destination location.

The sequence of events for this case is shown in Figure 8.2*a*. Four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Figure 8.2*b*. These units must be capable of performing their tasks simultaneously and without interfering with one another. Information is passed from one unit to the next through a storage buffer. As an instruction progresses through the pipeline, all the information needed by the stages downstream must be passed along. For example, during clock cycle 4, the information in the buffers is as follows:

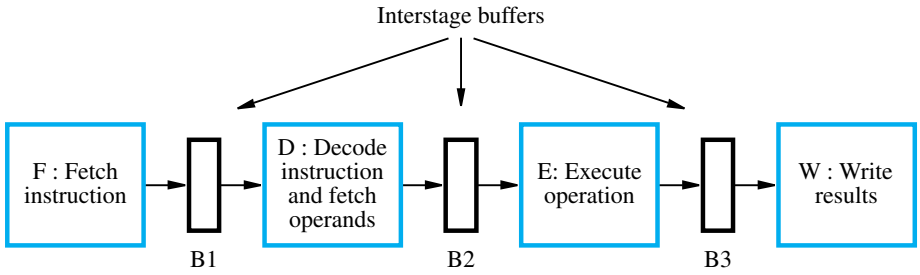
- Buffer B1 holds instruction I_3, which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.
- Buffer B2 holds both the source operands for instruction I_2 and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle 3. The buffer also holds the information needed for the write step of instruction I_2 (step W_2). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.
- Buffer B3 holds the results produced by the execution unit and the destination information for instruction I_1.

8.1.1 ROLE OF CACHE MEMORY

Each stage in a pipeline is expected to complete its operation in one clock cycle. Hence, the clock period should be sufficiently long to complete the task being performed in any stage. If different units require different amounts of time, the clock period must allow the longest task to be completed. A unit that completes its task early is idle for the remainder of the clock period. Hence, pipelining is most effective in improving



(a) Instruction execution divided into four steps



(b) Hardware organization

Figure 8.2 A 4-stage pipeline.

performance if the tasks being performed in different stages require about the same amount of time.

This consideration is particularly important for the instruction fetch step, which is assigned one clock period in Figure 8.2*a*. The clock cycle has to be equal to or greater than the time needed to complete a fetch operation. However, the access time of the main memory may be as much as ten times greater than the time needed to perform basic pipeline stage operations inside the processor, such as adding two numbers. Thus, if each instruction fetch required access to the main memory, pipelining would be of little value.

The use of cache memories solves the memory access problem. In particular, when a cache is included on the same chip as the processor, access time to the cache is usually the same as the time needed to perform other basic operations inside the processor. This

makes it possible to divide instruction fetching and processing into steps that are more or less equal in duration. Each of these steps is performed by a different pipeline stage, and the clock period is chosen to correspond to the longest one.

8.1.2 PIPELINE PERFORMANCE

The pipelined processor in Figure 8.2 completes the processing of one instruction in each clock cycle, which means that the rate of instruction processing is four times that of sequential operation. The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages. However, this increase would be achieved only if pipelined operation as depicted in Figure 8.2*a* could be sustained without interruption throughout program execution. Unfortunately, this is not the case.

For a variety of reasons, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted. For example, stage E in the four-stage pipeline of Figure 8.2*b* is responsible for arithmetic and logic operations, and one clock cycle is assigned for this task. Although this may be sufficient for most operations, some operations, such as divide, may require more time to complete. Figure 8.3 shows an example in which the operation specified in instruction I_2 requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the Write stage must be told to do nothing, because it has no data to work with. Meanwhile, the information in buffer B2 must remain intact until the Execute stage has completed its operation. This means that stage 2 and, in turn, stage 1 are blocked from accepting new instructions because the information in B1 cannot be overwritten. Thus, steps D_4 and F_5 must be postponed as shown.

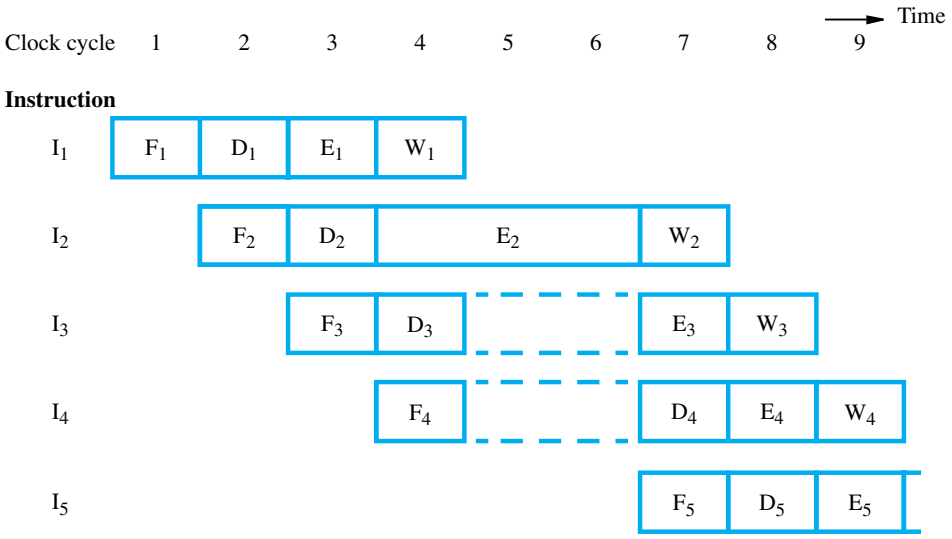
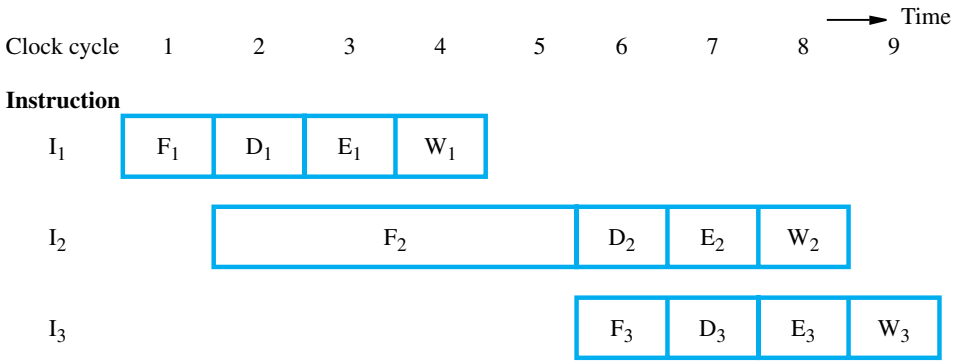


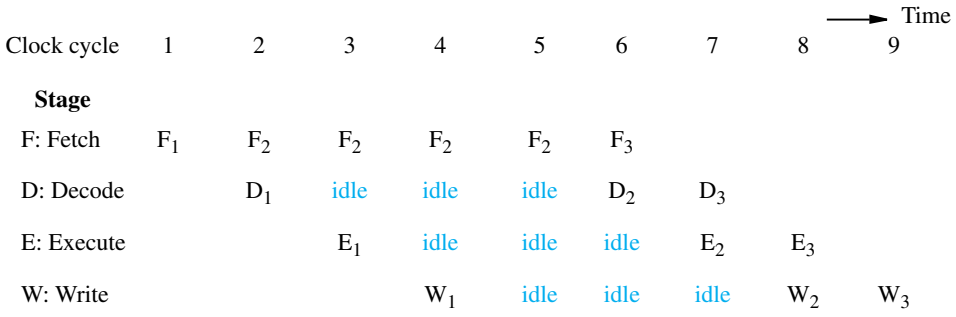
Figure 8.3 Effect of an execution operation taking more than one clock cycle.

Pipelined operation in Figure 8.3 is said to have been *stalled* for two clock cycles. Normal pipelined operation resumes in cycle 7. Any condition that causes the pipeline to stall is called a *hazard*. We have just seen an example of a *data hazard*. A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result some operation has to be delayed, and the pipeline stalls.

The pipeline may also be stalled because of a delay in the availability of an instruction. For example, this may be a result of a miss in the cache, requiring the instruction to be fetched from the main memory. Such hazards are often called *control hazards* or *instruction hazards*. The effect of a cache miss on pipelined operation is illustrated in Figure 8.4. Instruction I_1 is fetched from the cache in cycle 1, and its execution proceeds normally. However, the fetch operation for instruction I_2, which is started in cycle 2, results in a cache miss. The instruction fetch unit must now suspend any further fetch requests and wait for I_2 to arrive. We assume that instruction I_2 is received and loaded into buffer B1 at the end of cycle 5. The pipeline resumes its normal operation at that point.



(a) Instruction execution steps in successive clock cycles



(b) Function performed by each processor stage in successive clock cycles

Figure 8.4 Pipeline stall caused by a cache miss in F2.

An alternative representation of the operation of a pipeline in the case of a cache miss is shown in Figure 8.4*b*. This figure gives the function performed by each pipeline stage in each clock cycle. Note that the Decode unit is idle in cycles 3 through 5, the Execute unit is idle in cycles 4 through 6, and the Write unit is idle in cycles 5 through 7. Such idle periods are called *stalls*. They are also often referred to as *bubbles* in the pipeline. Once created as a result of a delay in one of the pipeline stages, a bubble moves downstream until it reaches the last unit.

A third type of hazard that may be encountered in pipelined operation is known as a *structural hazard*. This is the situation when two instructions require the use of a given hardware resource at the same time. The most common case in which this hazard may arise is in access to memory. One instruction may need to access memory as part of the Execute or Write stage while another instruction is being fetched. If instructions and data reside in the same cache unit, only one instruction can proceed and the other instruction is delayed. Many processors use separate instruction and data caches to avoid this delay.

An example of a structural hazard is shown in Figure 8.5. This figure shows how the load instruction

Load X(R1),R2

can be accommodated in our example 4-stage pipeline. The memory address, X+[R1], is computed in step E_2 in cycle 4, then memory access takes place in cycle 5. The operand read from memory is written into register R2 in cycle 6. This means that the execution step of this instruction takes two clock cycles (cycles 4 and 5). It causes the pipeline to stall for one cycle, because both instructions I_2 and I_3 require access to the register file in cycle 6. Even though the instructions and their data are all available, the pipeline is

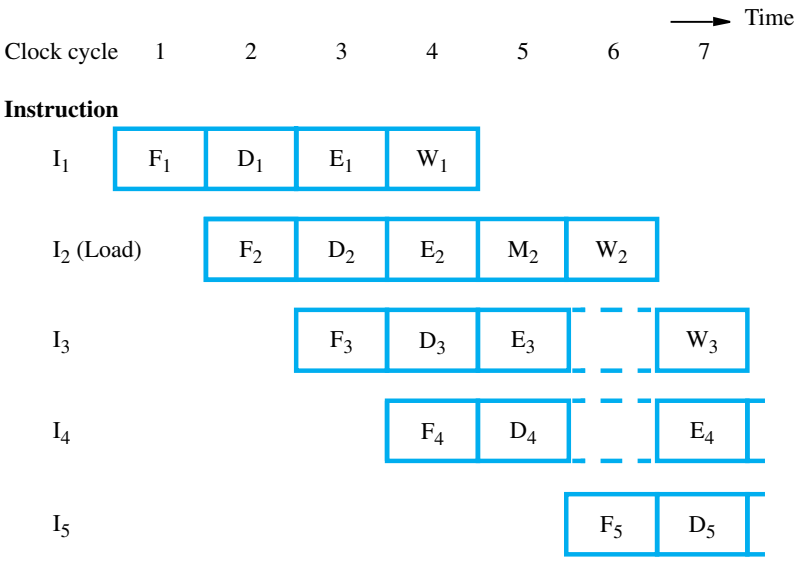


Figure 8.5 Effect of a Load instruction on pipeline timing.

stalled because one hardware resource, the register file, cannot handle two operations at once. If the register file had two input ports, that is, if it allowed two simultaneous write operations, the pipeline would not be stalled. In general, structural hazards are avoided by providing sufficient hardware resources on the processor chip.

It is important to understand that pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases, where throughput is measured by the rate at which instruction execution is completed. Any time one of the stages in the pipeline cannot complete its operation in one clock cycle, the pipeline stalls, and some degradation in performance occurs. Thus, the performance level of one instruction completion in each clock cycle is actually the upper limit for the throughput achievable in a pipelined processor organized as in Figure 8.2*b*.

An important goal in designing processors is to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact. In the following sections we discuss various hazards, starting with data hazards, followed by control hazards. In each case we present some of the techniques used to mitigate their negative effect on performance. We return to the issue of performance assessment in Section 8.8.

8.2 DATA HAZARDS

A data hazard is a situation in which the pipeline is stalled because the data to be operated on are delayed for some reason, as illustrated in Figure 8.3. We will now examine the issue of availability of data in some detail.

Consider a program that contains two instructions, I_1 followed by I_2. When this program is executed in a pipeline, the execution of I_2 can begin before the execution of I_1 is completed. This means that the results generated by I_1 may not be available for use by I_2. We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially. The potential for obtaining incorrect results when operations are performed concurrently can be demonstrated by a simple example. Assume that $A = 5$, and consider the following two operations:

$$A \leftarrow 3 + A$$

$$B \leftarrow 4 \times A$$

When these operations are performed in the order given, the result is $B = 32$. But if they are performed concurrently, the value of A used in computing B would be the original value, 5, leading to an incorrect result. If these two operations are performed by instructions in a program, then the instructions must be executed one after the other, because the data used in the second instruction depend on the result of the first instruction. On the other hand, the two operations

$$A \leftarrow 5 \times C$$

$$B \leftarrow 20 + C$$

can be performed concurrently, because these operations are independent.

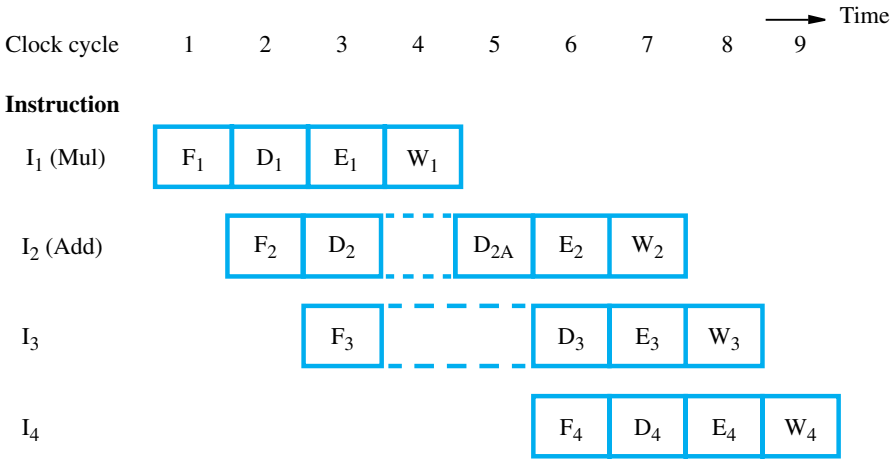


Figure 8.6 Pipeline stalled by data dependency between D_2 and W_1.

This example illustrates a basic constraint that must be enforced to guarantee correct results. When two operations depend on each other, they must be performed sequentially in the correct order. This rather obvious condition has far-reaching consequences. Understanding its implications is the key to understanding the variety of design alternatives and trade-offs encountered in pipelined computers.

Consider the pipeline in Figure 8.2. The data dependency just described arises when the destination of one instruction is used as a source in the next instruction. For example, the two instructions

```
Mul   R2,R3,R4
Add   R5,R4,R6
```

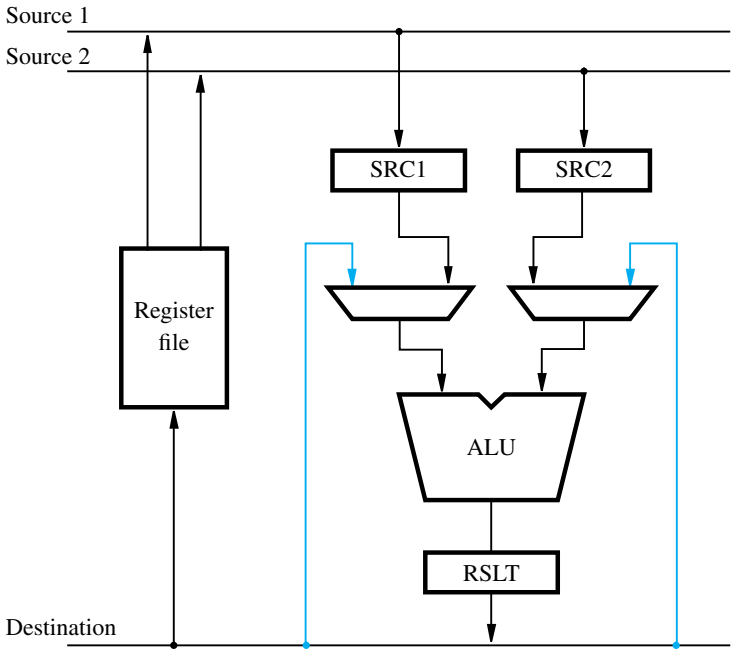
give rise to a data dependency. The result of the multiply instruction is placed into register R4, which in turn is one of the two source operands of the Add instruction. Assuming that the multiply operation takes one clock cycle to complete, execution would proceed as shown in Figure 8.6. As the Decode unit decodes the Add instruction in cycle 3, it realizes that R4 is used as a source operand. Hence, the D step of that instruction cannot be completed until the W step of the multiply instruction has been completed. Completion of step D_2 must be delayed to clock cycle 5, and is shown as step D_{2A} in the figure. Instruction I_3 is fetched in cycle 3, but its decoding must be delayed because step D_3 cannot precede D_2. Hence, pipelined execution is stalled for two cycles.

8.2.1 OPERAND FORWARDING

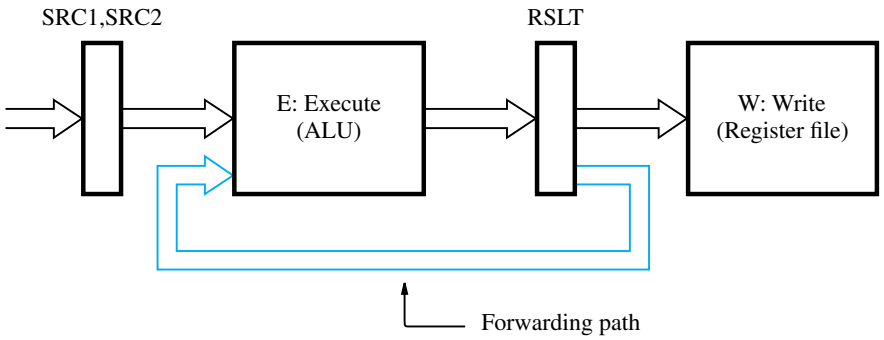
The data hazard just described arises because one instruction, instruction I_2 in Figure 8.6, is waiting for data to be written in the register file. However, these data are available at the output of the ALU once the Execute stage completes step E_1. Hence, the delay can

be reduced, or possibly eliminated, if we arrange for the result of instruction I_1 to be forwarded directly for use in step E_2.

Figure 8.7*a* shows a part of the processor datapath involving the ALU and the register file. This arrangement is similar to the three-bus structure in Figure 7.8, except that registers SRC1, SRC2, and RSLT have been added. These registers constitute the



(a) Datapath



(b) Position of the source and result registers in the processor pipeline

Figure 8.7 Operand forwarding in a pipelined processor.

interstage buffers needed for pipelined operation, as illustrated in Figure 8.7*b*. With reference to Figure 8.2*b*, registers SRC1 and SRC2 are part of buffer B2 and RSLT is part of B3. The data forwarding mechanism is provided by the blue connection lines. The two multiplexers connected at the inputs to the ALU allow the data on the destination bus to be selected instead of the contents of either the SRC1 or SRC2 register.

When the instructions in Figure 8.6 are executed in the datapath of Figure 8.7, the operations performed in each clock cycle are as follows. After decoding instruction I_2 and detecting the data dependency, a decision is made to use data forwarding. The operand not involved in the dependency, register R2, is read and loaded in register SRC1 in clock cycle 3. In the next clock cycle, the product produced by instruction I_1 is available in register RSLT, and because of the forwarding connection, it can be used in step E_2. Hence, execution of I_2 proceeds without interruption.

8.2.2 HANDLING DATA HAZARDS IN SOFTWARE

In Figure 8.6, we assumed the data dependency is discovered by the hardware while the instruction is being decoded. The control hardware delays reading register R4 until cycle 5, thus introducing a 2-cycle stall unless operand forwarding is used. An alternative approach is to leave the task of detecting data dependencies and dealing with them to the software. In this case, the compiler can introduce the two-cycle delay needed between instructions I_1 and I_2 by inserting NOP (No-operation) instructions, as follows:

```
I1:  Mul   R2,R3,R4
     NOP
     NOP
I2:  Add   R5,R4,R6
```

If the responsibility for detecting such dependencies is left entirely to the software, the compiler must insert the NOP instructions to obtain a correct result. This possibility illustrates the close link between the compiler and the hardware. A particular feature can be either implemented in hardware or left to the compiler. Leaving tasks such as inserting NOP instructions to the compiler leads to simpler hardware. Being aware of the need for a delay, the compiler can attempt to reorder instructions to perform useful tasks in the NOP slots, and thus achieve better performance. On the other hand, the insertion of NOP instructions leads to larger code size. Also, it is often the case that a given processor architecture has several hardware implementations, offering different features. NOP instructions inserted to satisfy the requirements of one implementation may not be needed and, hence, would lead to reduced performance on a different implementation.

8.2.3 SIDE EFFECTS

The data dependencies encountered in the preceding examples are explicit and easily detected because the register involved is named as the destination in instruction I_1 and as a source in I_2. Sometimes an instruction changes the contents of a register other

than the one named as the destination. An instruction that uses an autoincrement or autodecrement addressing mode is an example. In addition to storing new data in its destination location, the instruction changes the contents of a source register used to access one of its operands. All the precautions needed to handle data dependencies involving the destination location must also be applied to the registers affected by an autoincrement or autodecrement operation. When a location other than one explicitly named in an instruction as a destination operand is affected, the instruction is said to have a *side effect.* For example, stack instructions, such as push and pop, produce similar side effects because they implicitly use the autoincrement and autodecrement addressing modes.

Another possible side effect involves the condition code flags, which are used by instructions such as conditional branches and add-with-carry. Suppose that registers R1 and R2 hold a double-precision integer number that we wish to add to another double-precision number in registers R3 and R4. This may be accomplished as follows:

```
Add             R1,R3
AddWithCarry    R2,R4
```

An implicit dependency exists between these two instructions through the carry flag. This flag is set by the first instruction and used in the second instruction, which performs the operation

$$R4 \leftarrow [R2] + [R4] + \text{carry}$$

Instructions that have side effects give rise to multiple data dependencies, which lead to a substantial increase in the complexity of the hardware or software needed to resolve them. For this reason, instructions designed for execution on pipelined hardware should have few side effects. Ideally, only the contents of the destination location, either a register or a memory location, should be affected by any given instruction. Side effects, such as setting the condition code flags or updating the contents of an address pointer, should be kept to a minimum. However, Chapter 2 showed that the autoincrement and autodecrement addressing modes are potentially useful. Condition code flags are also needed for recording such information as the generation of a carry or the occurrence of overflow in an arithmetic operation. In Section 8.4 we show how such functions can be provided by other means that are consistent with a pipelined organization and with the requirements of optimizing compilers.

8.3 INSTRUCTION HAZARDS

The purpose of the instruction fetch unit is to supply the execution units with a steady stream of instructions. Whenever this stream is interrupted, the pipeline stalls, as Figure 8.4 illustrates for the case of a cache miss. A branch instruction may also cause the pipeline to stall. We will now examine the effect of branch instructions and the techniques that can be used for mitigating their impact. We start with unconditional branches.

8.3.1 UNCONDITIONAL BRANCHES

Figure 8.8 shows a sequence of instructions being executed in a two-stage pipeline. Instructions I_1 to I_3 are stored at successive memory addresses, and I_2 is a branch instruction. Let the branch target be instruction I_k. In clock cycle 3, the fetch operation for instruction I_3 is in progress at the same time that the branch instruction is being decoded and the target address computed. In clock cycle 4, the processor must discard I_3, which has been incorrectly fetched, and fetch instruction I_k. In the meantime, the hardware unit responsible for the Execute (E) step must be told to do nothing during that clock period. Thus, the pipeline is stalled for one clock cycle.

The time lost as a result of a branch instruction is often referred to as the *branch penalty*. In Figure 8.8, the branch penalty is one clock cycle. For a longer pipeline, the branch penalty may be higher. For example, Figure 8.9*a* shows the effect of a branch instruction on a four-stage pipeline. We have assumed that the branch address is computed in step E_2. Instructions I_3 and I_4 must be discarded, and the target instruction, I_k, is fetched in clock cycle 5. Thus, the branch penalty is two clock cycles.

Reducing the branch penalty requires the branch address to be computed earlier in the pipeline. Typically, the instruction fetch unit has dedicated hardware to identify a branch instruction and compute the branch target address as quickly as possible after an instruction is fetched. With this additional hardware, both of these tasks can be performed in step D_2, leading to the sequence of events shown in Figure 8.9*b*. In this case, the branch penalty is only one clock cycle.

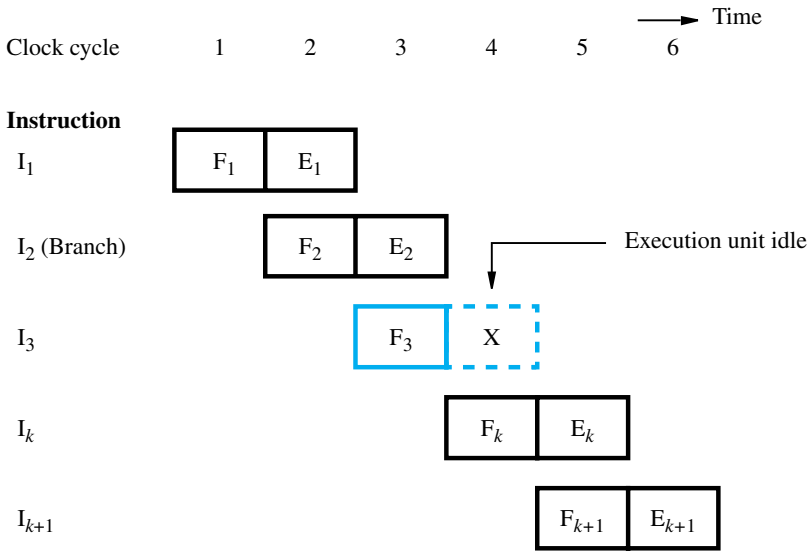
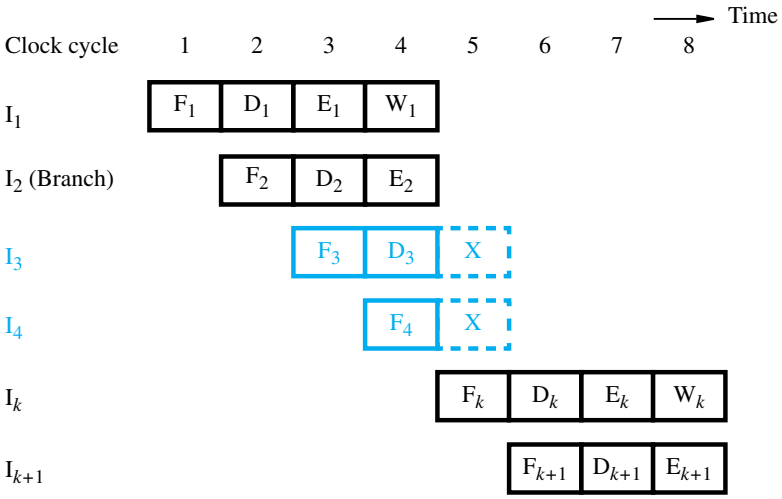
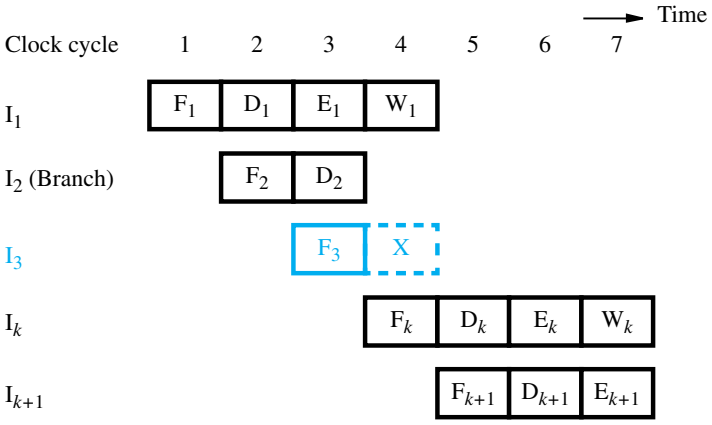


Figure 8.8 An idle cycle caused by a branch instruction.



(a) Branch address computed in Execute stage



(b) Branch address computed in Decode stage

Figure 8.9 Branch timing.

Instruction Queue and Prefetching

Either a cache miss or a branch instruction stalls the pipeline for one or more clock cycles. To reduce the effect of these interruptions, many processors employ sophisticated fetch units that can fetch instructions before they are needed and put them in a queue. Typically, the instruction queue can store several instructions. A separate unit, which we call the *dispatch unit,* takes instructions from the front of the queue and

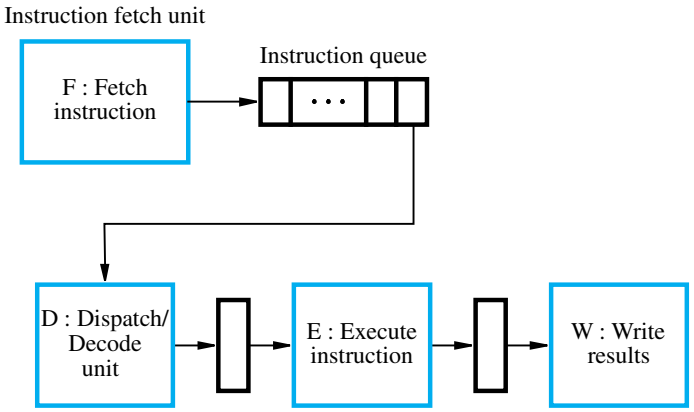


Figure 8.10 Use of an instruction queue in the hardware organization of Figure 8.2*b*.

sends them to the execution unit. This leads to the organization shown in Figure 8.10. The dispatch unit also performs the decoding function.

To be effective, the fetch unit must have sufficient decoding and processing capability to recognize and execute branch instructions. It attempts to keep the instruction queue filled at all times to reduce the impact of occasional delays when fetching instructions. When the pipeline stalls because of a data hazard, for example, the dispatch unit is not able to issue instructions from the instruction queue. However, the fetch unit continues to fetch instructions and add them to the queue. Conversely, if there is a delay in fetching instructions because of a branch or a cache miss, the dispatch unit continues to issue instructions from the instruction queue.

Figure 8.11 illustrates how the queue length changes and how it affects the relationship between different pipeline stages. We have assumed that initially the queue contains one instruction. Every fetch operation adds one instruction to the queue and every dispatch operation reduces the queue length by one. Hence, the queue length remains the same for the first four clock cycles. (There is both an F and a D step in each of these cycles.) Suppose that instruction I_1 introduces a 2-cycle stall. Since space is available in the queue, the fetch unit continues to fetch instructions and the queue length rises to 3 in clock cycle 6.

Instruction I_5 is a branch instruction. Its target instruction, I_k, is fetched in cycle 7, and instruction I_6 is discarded. The branch instruction would normally cause a stall in cycle 7 as a result of discarding instruction I_6. Instead, instruction I_4 is dispatched from the queue to the decoding stage. After discarding I_6, the queue length drops to 1 in cycle 8. The queue length will be at this value until another stall is encountered.

Now observe the sequence of instruction completions in Figure 8.11. Instructions I_1, I_2, I_3, I_4, and I_k complete execution in successive clock cycles. Hence, the branch instruction does not increase the overall execution time. This is because the instruction fetch unit has executed the branch instruction (by computing the branch address) concurrently with the execution of other instructions. This technique is referred to as *branch folding*.

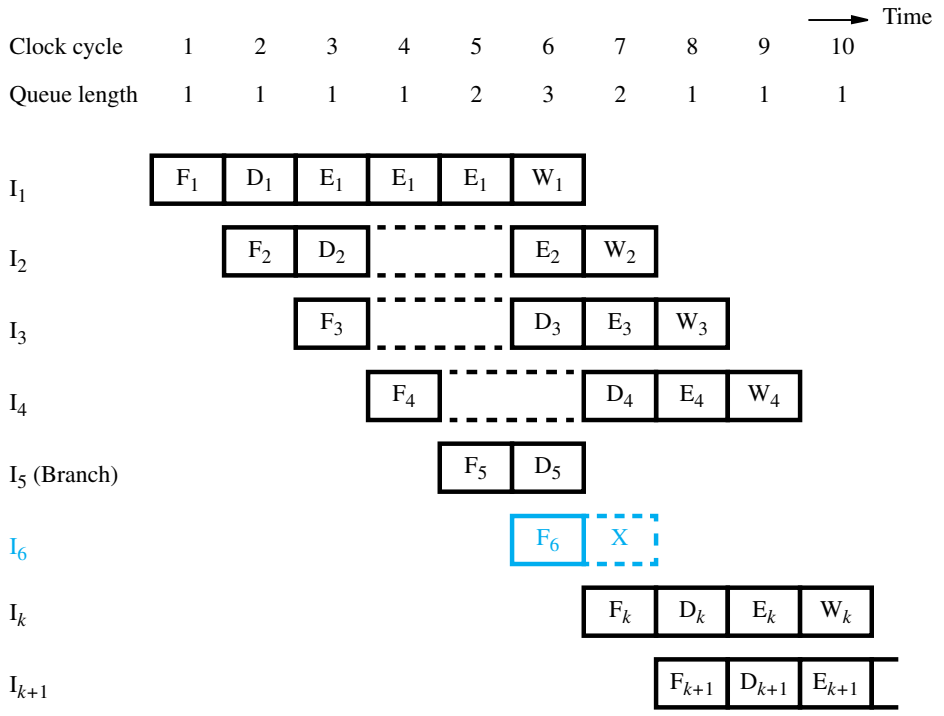


Figure 8.11 Branch timing in the presence of an instruction queue. Branch target address is computed in the D stage.

Note that branch folding occurs only if at the time a branch instruction is encountered, at least one instruction is available in the queue other than the branch instruction. If only the branch instruction is in the queue, execution would proceed as in Figure 8.9*b*. Therefore, it is desirable to arrange for the queue to be full most of the time, to ensure an adequate supply of instructions for processing. This can be achieved by increasing the rate at which the fetch unit reads instructions from the cache. In many processors, the width of the connection between the fetch unit and the instruction cache allows reading more than one instruction in each clock cycle. If the fetch unit replenishes the instruction queue quickly after a branch has occurred, the probability that branch folding will occur increases.

Having an instruction queue is also beneficial in dealing with cache misses. When a cache miss occurs, the dispatch unit continues to send instructions for execution as long as the instruction queue is not empty. Meanwhile, the desired cache block is read from the main memory or from a secondary cache. When fetch operations are resumed, the instruction queue is refilled. If the queue does not become empty, a cache miss will have no effect on the rate of instruction execution.

In summary, the instruction queue mitigates the impact of branch instructions on performance through the process of branch folding. It has a similar effect on stalls

caused by cache misses. The effectiveness of this technique is enhanced when the instruction fetch unit is able to read more than one instruction at a time from the instruction cache.

8.3.2 CONDITIONAL BRANCHES AND BRANCH PREDICTION

A conditional branch instruction introduces the added hazard caused by the dependency of the branch condition on the result of a preceding instruction. The decision to branch cannot be made until the execution of that instruction has been completed.

Branch instructions occur frequently. In fact, they represent about 20 percent of the dynamic instruction count of most programs. (The dynamic count is the number of instruction executions, taking into account the fact that some program instructions are executed many times because of loops.) Because of the branch penalty, this large percentage would reduce the gain in performance expected from pipelining. Fortunately, branch instructions can be handled in several ways to reduce their negative impact on the rate of execution of instructions.

Delayed Branch

In Figure 8.8, the processor fetches instruction I_3 before it determines whether the current instruction, I_2, is a branch instruction. When execution of I_2 is completed and a branch is to be made, the processor must discard I_3 and fetch the instruction at the branch target. The location following a branch instruction is called a *branch delay slot*. There may be more than one branch delay slot, depending on the time it takes to execute a branch instruction. For example, there are two branch delay slots in Figure 8.9*a* and one delay slot in Figure 8.9*b*. The instructions in the delay slots are always fetched and at least partially executed before the branch decision is made and the branch target address is computed.

A technique called *delayed branching* can minimize the penalty incurred as a result of conditional branch instructions. The idea is simple. The instructions in the delay slots are always fetched. Therefore, we would like to arrange for them to be fully executed whether or not the branch is taken. The objective is to be able to place useful instructions in these slots. If no useful instructions can be placed in the delay slots, these slots must be filled with NOP instructions. This situation is exactly the same as in the case of data dependency discussed in Section 8.2.

Consider the instruction sequence given in Figure 8.12*a*. Register R2 is used as a counter to determine the number of times the contents of register R1 are shifted left. For a processor with one delay slot, the instructions can be reordered as shown in Figure 8.12*b*. The shift instruction is fetched while the branch instruction is being executed. After evaluating the branch condition, the processor fetches the instruction at LOOP or at NEXT, depending on whether the branch condition is true or false, respectively. In either case, it completes execution of the shift instruction. The sequence of events during the last two passes in the loop is illustrated in Figure 8.13. Pipelined operation is not interrupted at any time, and there are no idle cycles. Logically, the program is executed as if the branch instruction were placed after the shift instruction. That is, branching takes place one instruction later than where the branch instruction appears in the instruction sequence in the memory, hence the name "delayed branch."

LOOP	Shift_left	R1
	Decrement	R2
	Branch=0	LOOP
NEXT	Add	R1,R3

(a) Original program loop

LOOP	Decrement	R2
	Branch=0	LOOP
	Shift_left	R1
NEXT	Add	R1,R3

(b) Reordered instructions

Figure 8.12 Reordering of instructions for a delayed branch.

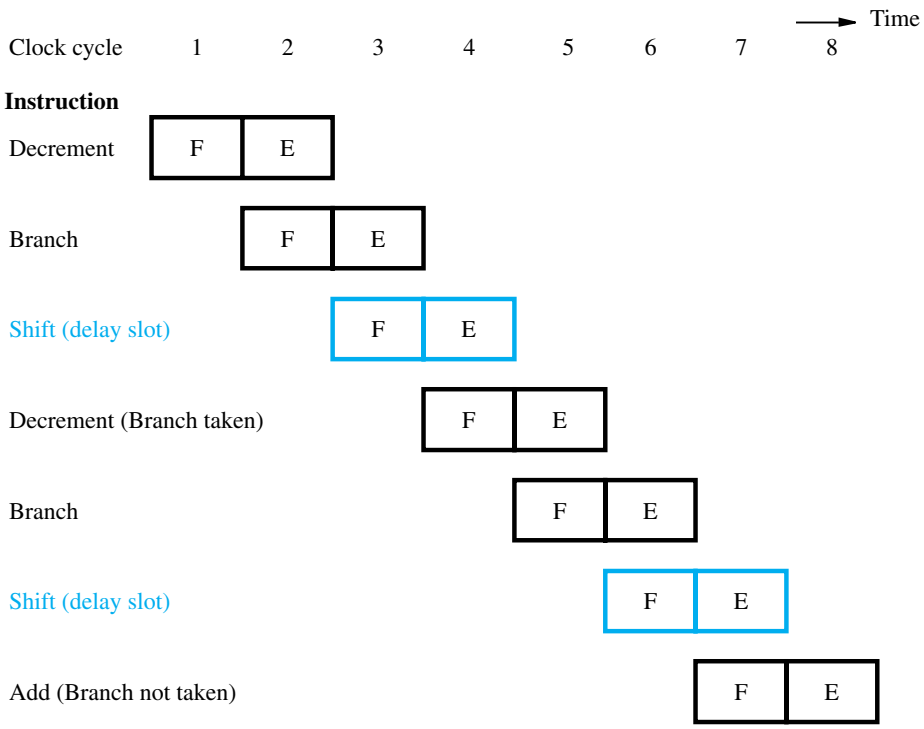


Figure 8.13 Execution timing showing the delay slot being filled during the last two passes through the loop in Figure 8.12*b*.

The effectiveness of the delayed branch approach depends on how often it is possible to reorder instructions as in Figure 8.12. Experimental data collected from many programs indicate that sophisticated compilation techniques can use one branch delay slot in as many as 85 percent of the cases. For a processor with two branch delay slots, the compiler attempts to find two instructions preceding the branch instruction that it can move into the delay slots without introducing a logical error. The chances of finding two such instructions are considerably less than the chances of finding one. Thus, if increasing the number of pipeline stages involves an increase in the number of branch delay slots, the potential gain in performance may not be fully realized.

Branch Prediction

Another technique for reducing the branch penalty associated with conditional branches is to attempt to predict whether or not a particular branch will be taken. The simplest form of branch prediction is to assume that the branch will not take place and to continue to fetch instructions in sequential address order. Until the branch condition is evaluated, instruction execution along the predicted path must be done on a speculative basis. *Speculative execution* means that instructions are executed before the processor is certain that they are in the correct execution sequence. Hence, care must be taken that no processor registers or memory locations are updated until it is confirmed that these instructions should indeed be executed. If the branch decision indicates otherwise, the instructions and all their associated data in the execution units must be purged, and the correct instructions fetched and executed.

An incorrectly predicted branch is illustrated in Figure 8.14 for a four-stage pipeline. The figure shows a Compare instruction followed by a Branch>0 instruction. Branch

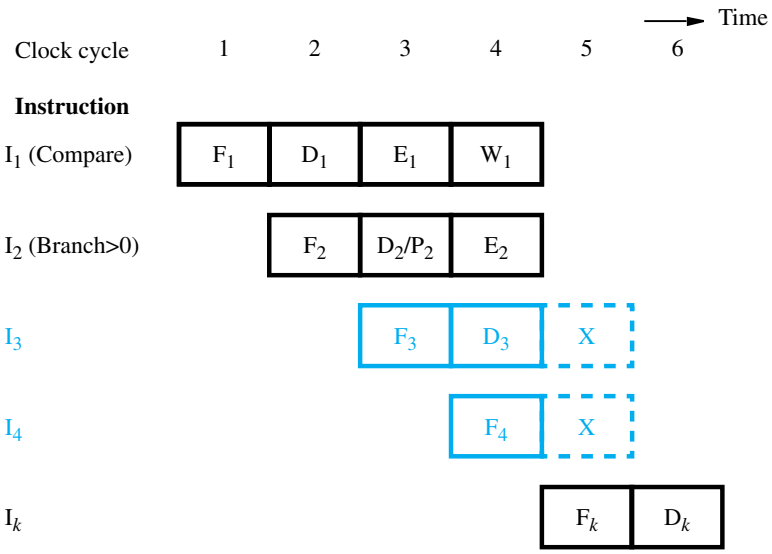


Figure 8.14 Timing when a branch decision has been incorrectly predicted as not taken.

prediction takes place in cycle 3, while instruction I_3 is being fetched. The fetch unit predicts that the branch will not be taken, and it continues to fetch instruction I_4 as I_3 enters the Decode stage. The results of the compare operation are available at the end of cycle 3. Assuming that they are forwarded immediately to the instruction fetch unit, the branch condition is evaluated in cycle 4. At this point, the instruction fetch unit realizes that the prediction was incorrect, and the two instructions in the execution pipe are purged. A new instruction, I_k, is fetched from the branch target address in clock cycle 5.

If branch outcomes were random, then half the branches would be taken. Then the simple approach of assuming that branches will not be taken would save the time lost to conditional branches 50 percent of the time. However, better performance can be achieved if we arrange for some branch instructions to be predicted as taken and others as not taken, depending on the expected program behavior. For example, a branch instruction at the end of a loop causes a branch to the start of the loop for every pass through the loop except the last one. Hence, it is advantageous to assume that this branch will be taken and to have the instruction fetch unit start to fetch instructions at the branch target address. On the other hand, for a branch instruction at the beginning of a program loop, it is advantageous to assume that the branch will not be taken.

A decision on which way to predict the result of the branch may be made in hardware by observing whether the target address of the branch is lower than or higher than the address of the branch instruction. A more flexible approach is to have the compiler decide whether a given branch instruction should be predicted taken or not taken. The branch instructions of some processors, such as SPARC, include a branch prediction bit, which is set to 0 or 1 by the compiler to indicate the desired behavior. The instruction fetch unit checks this bit to predict whether the branch will be taken or not taken.

With either of these schemes, the branch prediction decision is always the same every time a given instruction is executed. Any approach that has this characteristic is called *static branch prediction*. Another approach in which the prediction decision may change depending on execution history is called *dynamic branch prediction*.

Dynamic Branch Prediction

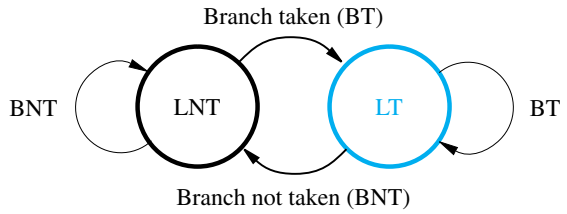
The objective of branch prediction algorithms is to reduce the probability of making a wrong decision, to avoid fetching instructions that eventually have to be discarded. In dynamic branch prediction schemes, the processor hardware assesses the likelihood of a given branch being taken by keeping track of branch decisions every time that instruction is executed.

In its simplest form, the execution history used in predicting the outcome of a given branch instruction is the result of the most recent execution of that instruction. The processor assumes that the next time the instruction is executed, the result is likely to be the same. Hence, the algorithm may be described by the two-state machine in Figure 8.15*a*. The two states are:

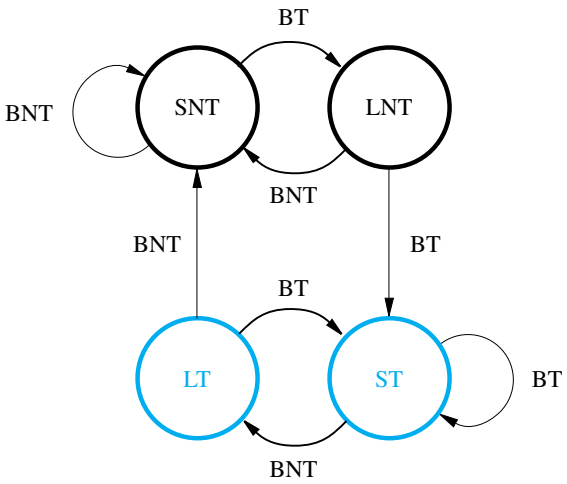
LT: Branch is likely to be taken

LNT: Branch is likely not to be taken

Suppose that the algorithm is started in state LNT. When the branch instruction is



(a) A 2-state algorithm



(b) A 4-state algorithm

Figure 8.15 State-machine representation of branch-prediction algorithms.

executed and if the branch is taken, the machine moves to state LT. Otherwise, it remains in state LNT. The next time the same instruction is encountered, the branch is predicted as taken if the corresponding state machine is in state LT. Otherwise it is predicted as not taken.

This simple scheme, which requires one bit of history information for each branch instruction, works well inside program loops. Once a loop is entered, the branch instruction that controls looping will always yield the same result until the last pass through the loop is reached. In the last pass, the branch prediction will turn out to be incorrect, and the branch history state machine will be changed to the opposite state. Unfortunately, this means that the next time this same loop is entered, and assuming that there will be more than one pass through the loop, the machine will lead to the wrong prediction.

Better performance can be achieved by keeping more information about execution history. An algorithm that uses 4 states, thus requiring two bits of history information for each branch instruction, is shown in Figure 8.15*b*. The four states are:

ST: Strongly likely to be taken

LT: Likely to be taken

LNT: Likely not to be taken

SNT: Strongly likely not to be taken

Again assume that the state of the algorithm is initially set to LNT. After the branch instruction has been executed, and if the branch is actually taken, the state is changed to ST; otherwise, it is changed to SNT. As program execution progresses and the same instruction is encountered again, the state of the branch prediction algorithm continues to change as shown. When a branch instruction is encountered, the instruction fetch unit predicts that the branch will be taken if the state is either LT or ST, and it begins to fetch instructions at the branch target address. Otherwise, it continues to fetch instructions in sequential address order.

It is instructive to examine the behavior of the branch prediction algorithm in some detail. When in state SNT, the instruction fetch unit predicts that the branch will not be taken. If the branch is actually taken, that is if the prediction is incorrect, the state changes to LNT. This means that the next time the same branch instruction is encountered, the instruction fetch unit will still predict that the branch will not be taken. Only if the prediction is incorrect twice in a row will the state change to ST. After that, the branch will be predicted as taken.

Let us reconsider what happens when executing a program loop. Assume that the branch instruction is at the end of the loop and that the processor sets the initial state of the algorithm to LNT. During the first pass, the prediction will be wrong (not taken), and hence the state will be changed to ST. In all subsequent passes the prediction will be correct, except for the last pass. At that time, the state will change to LT. When the loop is entered a second time, the prediction will be correct (branch taken).

We now add one final modification to correct the mispredicted branch at the time the loop is first entered. The cause of the misprediction in this case is the initial state of the branch prediction algorithm. In the absence of additional information about the nature of the branch instruction, we assumed that the processor sets the initial state to LNT. The information needed to set the initial state correctly can be provided by any of the static prediction schemes discussed earlier. Either by comparing addresses or by checking a prediction bit in the instruction, the processor sets the initial state of the algorithm to LNT or LT. In the case of a branch at the end of a loop, the compiler would indicate that the branch should be predicted as taken, causing the initial state to be set to LT. With this modification, branch prediction will be correct all the time, except for the final pass through the loop. Misprediction in this latter case is unavoidable.

The state information used in dynamic branch prediction algorithms may be kept by the processor in a variety of ways. It may be recorded in a look-up table, which is accessed using the low-order part of the branch instruction address. In this case, it is possible for two branch instructions to share the same table entry. This may lead to a

branch being mispredicted, but it does not cause an error in execution. Misprediction only introduces a small delay in execution time. An alternative approach is to store the history bits as a tag associated with branch instructions in the instruction cache. We will see in Section 8.7 how this information is handled in the SPARC processor.

8.4 INFLUENCE ON INSTRUCTION SETS

We have seen that some instructions are much better suited to pipelined execution than others. For example, instruction side effects can lead to undesirable data dependencies. In this section, we examine the relationship between pipelined execution and machine instruction features. We discuss two key aspects of machine instructions — addressing modes and condition code flags.

8.4.1 ADDRESSING MODES

Addressing modes should provide the means for accessing a variety of data structures simply and efficiently. Useful addressing modes include index, indirect, autoincrement, and autodecrement. Many processors provide various combinations of these modes to increase the flexibility of their instruction sets. Complex addressing modes, such as those involving double indexing, are often encountered.

In choosing the addressing modes to be implemented in a pipelined processor, we must consider the effect of each addressing mode on instruction flow in the pipeline. Two important considerations in this regard are the side effects of modes such as autoincrement and autodecrement and the extent to which complex addressing modes cause the pipeline to stall. Another important factor is whether a given mode is likely to be used by compilers.

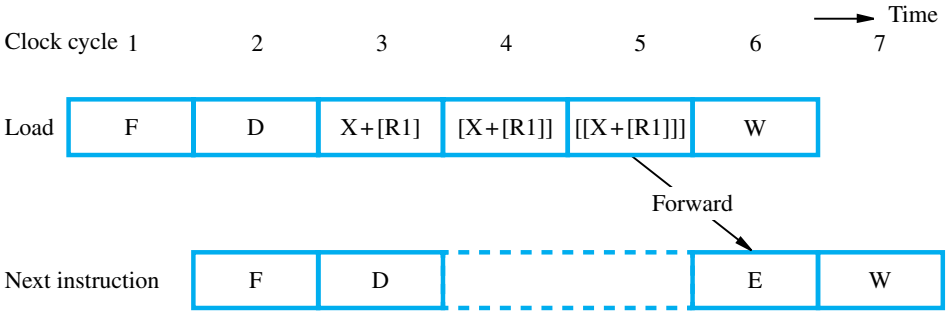
To compare various approaches, we assume a simple model for accessing operands in the memory. The load instruction Load X(R1),R2 takes five cycles to complete execution, as indicated in Figure 8.5. However, the instruction

Load (R1),R2

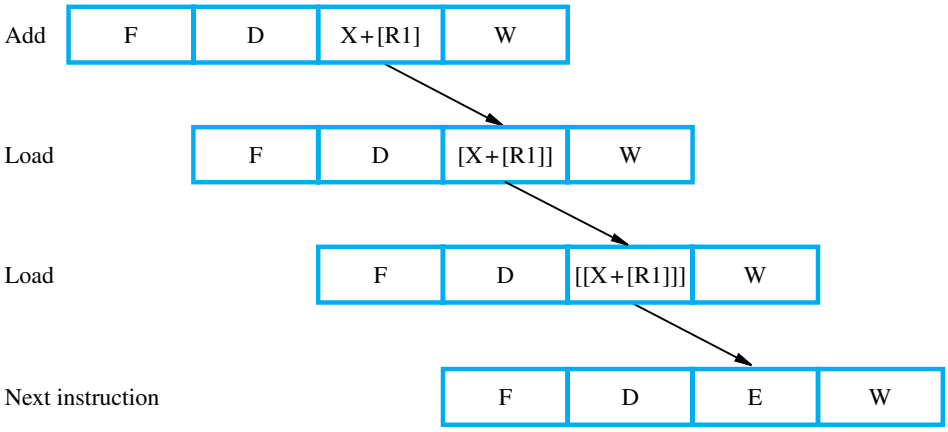
can be organized to fit a four-stage pipeline because no address computation is required. Access to memory can take place in stage E. A more complex addressing mode may require several accesses to the memory to reach the named operand. For example, the instruction

Load (X(R1)),R2

may be executed as shown in Figure 8.16*a*, assuming that the index offset, X, is given in the instruction word. After computing the address in cycle 3, the processor needs to access memory twice — first to read location X+[R1] in clock cycle 4 and then to read location [X+[R1]] in cycle 5. If R2 is a source operand in the next instruction, that instruction would be stalled for three cycles, which can be reduced to two cycles with operand forwarding, as shown.



(a) Complex addressing mode



(b) Simple addressing mode

Figure 8.16 Equivalent operations using complex and simple addressing modes.

To implement the same Load operation using only simple addressing modes requires several instructions. For example, on a computer that allows three operand addresses, we can use

```
Add    #X,R1,R2
Load   (R2),R2
Load   (R2),R2
```

The Add instruction performs the operation R2 ← X+ [R1]. The two Load instructions fetch the address and then the operand from the memory. This sequence of instructions takes exactly the same number of clock cycles as the original, single Load instruction, as shown in Figure 8.16*b*.

This example indicates that, in a pipelined processor, complex addressing modes that involve several accesses to the memory do not necessarily lead to faster execution. The main advantage of such modes is that they reduce the number of instructions needed to perform a given task and thereby reduce the program space needed in the main memory. Their main disadvantage is that their long execution times cause the pipeline to stall, thus reducing its effectiveness. They require more complex hardware to decode and execute them. Also, they are not convenient for compilers to work with.

The instruction sets of modern processors are designed to take maximum advantage of pipelined hardware. Because complex addressing modes are not suitable for pipelined execution, they should be avoided. The addressing modes used in modern processors often have the following features:

- Access to an operand does not require more than one access to the memory.
- Only load and store instructions access memory operands.
- The addressing modes used do not have side effects.

Three basic addressing modes that have these features are register, register indirect, and index. The first two require no address computation. In the index mode, the address can be computed in one cycle, whether the index value is given in the instruction or in a register. Memory is accessed in the following cycle. None of these modes has any side effects, with one possible exception. Some architectures, such as ARM, allow the address computed in the index mode to be written back into the index register. This is a side effect that would not be allowed under the guidelines above. Note also that relative addressing can be used; this is a special case of indexed addressing in which the program counter is used as the index register.

The three features just listed were first emphasized as part of the concept of RISC processors. The SPARC processor architecture, which adheres to these guidelines, is presented in Section 8.7.

8.4.2 CONDITION CODES

In many processors, such as those described in Chapter 3, the condition code flags are stored in the processor status register. They are either set or cleared by many instructions, so that they can be tested by subsequent conditional branch instructions to change the flow of program execution. An optimizing compiler for a pipelined processor attempts to reorder instructions to avoid stalling the pipeline when branches or data dependencies between successive instructions occur. In doing so, the compiler must ensure that reordering does not cause a change in the outcome of a computation. The dependency introduced by the condition-code flags reduces the flexibility available for the compiler to reorder instructions.

Consider the sequence of instructions in Figure 8.17*a*, and assume that the execution of the Compare and Branch=0 instructions proceeds as in Figure 8.14. The branch decision takes place in step E_2 rather than D_2 because it must await the result of the Compare instruction. The execution time of the Branch instruction can be reduced

Add	R1,R2
Compare	R3,R4
Branch=0	. . .

(a) A program fragment

Compare	R3,R4
Add	R1,R2
Branch=0	. . .

(b) Instructions reordered

Figure 8.17 Instruction reordering.

by interchanging the Add and Compare instructions, as shown in Figure 8.17*b*. This will delay the branch instruction by one cycle relative to the Compare instruction. As a result, at the time the Branch instruction is being decoded the result of the Compare instruction will be available and a correct branch decision will be made. There would be no need for branch prediction. However, interchanging the Add and Compare instructions can be done only if the Add instruction does not affect the condition codes.

These observations lead to two important conclusions about the way condition codes should be handled. First, to provide flexibility in reordering instructions, the condition-code flags should be affected by as few instructions as possible. Second, the compiler should be able to specify in which instructions of a program the condition codes are affected and in which they are not. An instruction set designed with pipelining in mind usually provides the desired flexibility. Figure 8.17*b* shows the instructions reordered assuming that the condition code flags are affected only when this is explicitly stated as part of the instruction OP code. The SPARC and ARM architectures provide this flexibility.

8.5 DATAPATH AND CONTROL CONSIDERATIONS

Organization of the internal datapath of a processor was introduced in Chapter 7. Consider the three-bus structure presented in Figure 7.8. To make it suitable for pipelined execution, it can be modified as shown in Figure 8.18 to support a 4-stage pipeline. The resources involved in stages F and E are shown in blue and those used in stages D and W in black. Operations in the data cache may happen during stage E or at a later stage, depending on the addressing mode and the implementation details. This section

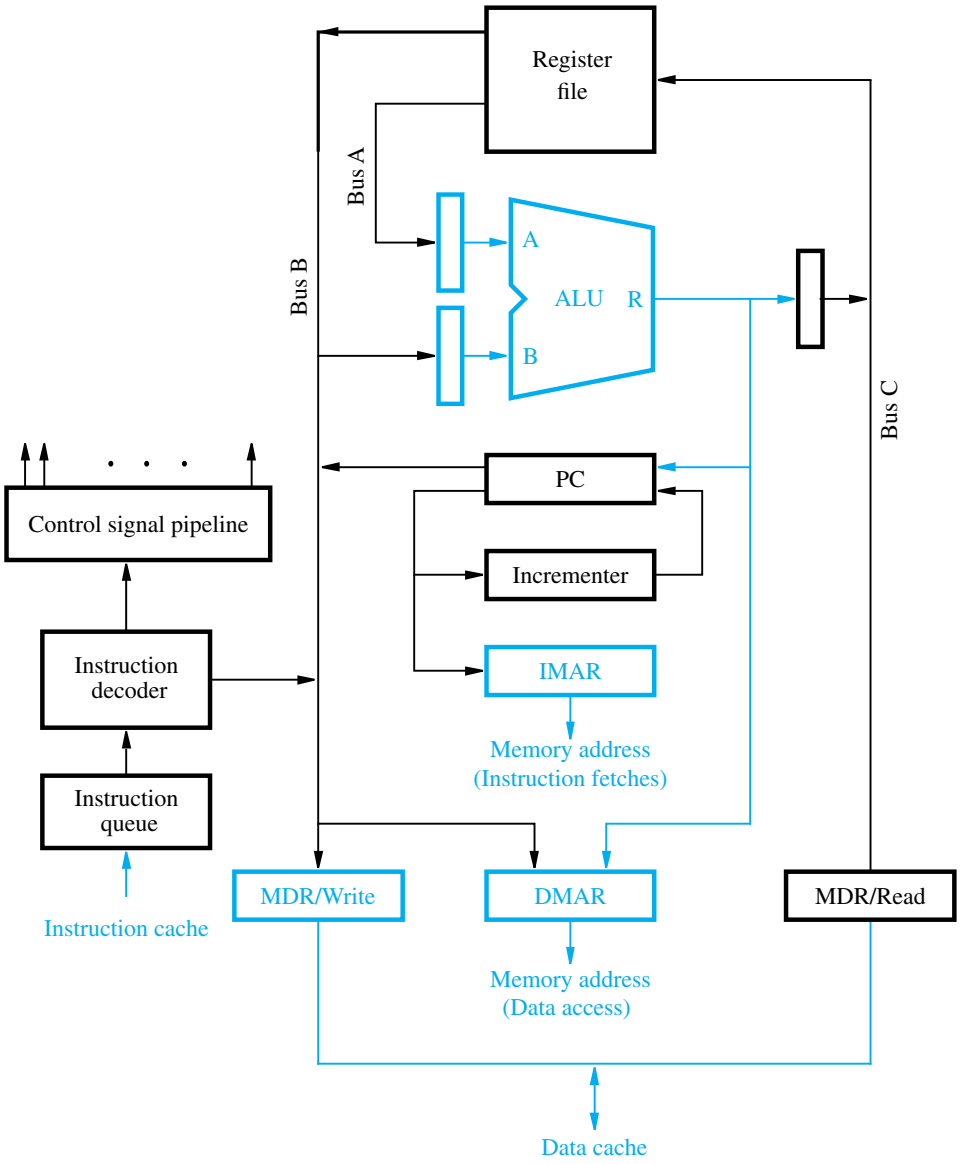


Figure 8.18 Datapath modified for pipelined execution with interstage buffers at the input and output of the ALU.

is shown in blue. Several important changes to Figure 7.8 should be noted:

1. There are separate instruction and data caches that use separate address and data connections to the processor. This requires two versions of the MAR register, IMAR for accessing the instruction cache and DMAR for accessing the data cache.
2. The PC is connected directly to the IMAR, so that the contents of the PC can be transferred to IMAR at the same time that an independent ALU operation is taking place.

3. The data address in DMAR can be obtained directly from the register file or from the ALU to support the register indirect and indexed addressing modes.

4. Separate MDR registers are provided for read and write operations. Data can be transferred directly between these registers and the register file during load and store operations without the need to pass through the ALU.

5. Buffer registers have been introduced at the inputs and output of the ALU. These are registers SRC1, SRC2, and RSLT in Figure 8.7. Forwarding connections are not included in Figure 8.18. They may be added if desired.

6. The instruction register has been replaced with an instruction queue, which is loaded from the instruction cache.

7. The output of the instruction decoder is connected to the control signal pipeline. The need for buffering control signals and passing them from one stage to the next along with the instruction is discussed in Section 8.1. This pipeline holds the control signals in buffers B2 and B3 in Figure 8.2*a*.

The following operations can be performed independently in the processor of Figure 8.18:

- Reading an instruction from the instruction cache
- Incrementing the PC
- Decoding an instruction
- Reading from or writing into the data cache
- Reading the contents of up to two registers from the register file
- Writing into one register in the register file
- Performing an ALU operation

Because these operations do not use any shared resources, they can be performed simultaneously in any combination. The structure provides the flexibility required to implement the four-stage pipeline in Figure 8.2. For example, let I_1, I_2, I_3, and I_4 be a sequence of four instructions. As shown in Figure 8.2*a*, the following actions all happen during clock cycle 4:

- Write the result of instruction I_1 into the register file
- Read the operands of instruction I_2 from the register file
- Decode instruction I_3
- Fetch instruction I_4 and increment the PC.

8.6 SUPERSCALAR OPERATION

Pipelining makes it possible to execute instructions concurrently. Several instructions are present in the pipeline at the same time, but they are in different stages of their execution. While one instruction is performing an ALU operation, another instruction is being decoded and yet another is being fetched from the memory. Instructions enter the pipeline in strict program order. In the absence of hazards, one instruction enters the pipeline and one instruction completes execution in each clock cycle. This means that the maximum throughput of a pipelined processor is one instruction per clock cycle.

A more aggressive approach is to equip the processor with multiple processing units to handle several instructions in parallel in each processing stage. With this arrangement, several instructions start execution in the same clock cycle, and the processor is said to use *multiple-issue*. Such processors are capable of achieving an instruction execution throughput of more than one instruction per cycle. They are known as *superscalar* processors. Many modern high-performance processors use this approach.

We introduced the idea of an instruction queue in Section 8.3. We pointed out that to keep the instruction queue filled, a processor should be able to fetch more than one instruction at a time from the cache. For superscalar operation, this arrangement is essential. Multiple-issue operation requires a wider path to the cache and multiple execution units. Separate execution units are provided for integer and floating-point instructions.

Figure 8.19 shows an example of a processor with two execution units, one for integer and one for floating-point operations. The Instruction fetch unit is capable of reading two instructions at a time and storing them in the instruction queue. In each clock cycle, the Dispatch unit retrieves and decodes up to two instructions from the front of the queue. If there is one integer, one floating-point instruction, and no hazards, both instructions are dispatched in the same clock cycle.

In a superscalar processor, the detrimental effect on performance of various hazards becomes even more pronounced. The compiler can avoid many hazards through judicious selection and ordering of instructions. For example, for the processor in Figure 8.19, the compiler should strive to interleave floating-point and integer instructions. This would enable the dispatch unit to keep both the integer and floating-point

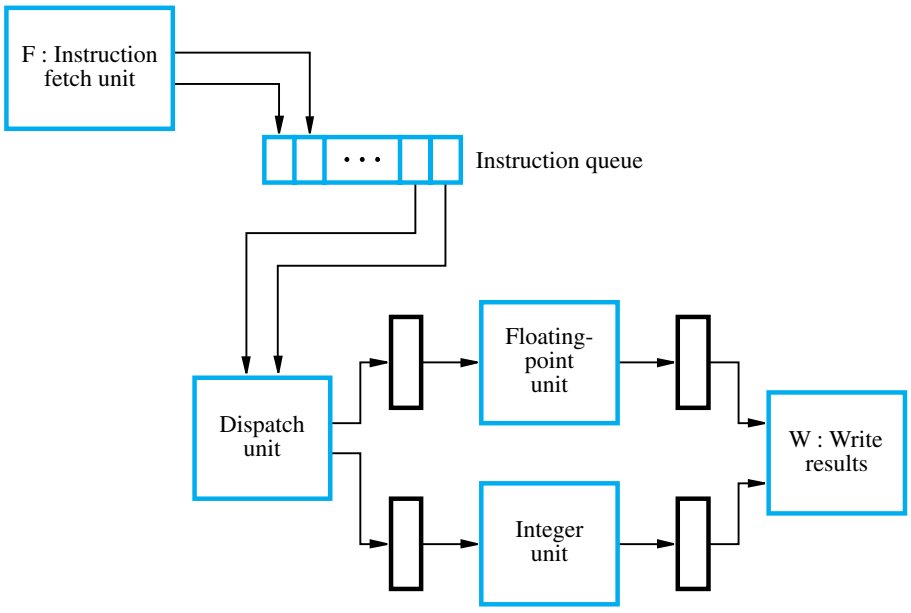


Figure 8.19 A processor with two execution units.

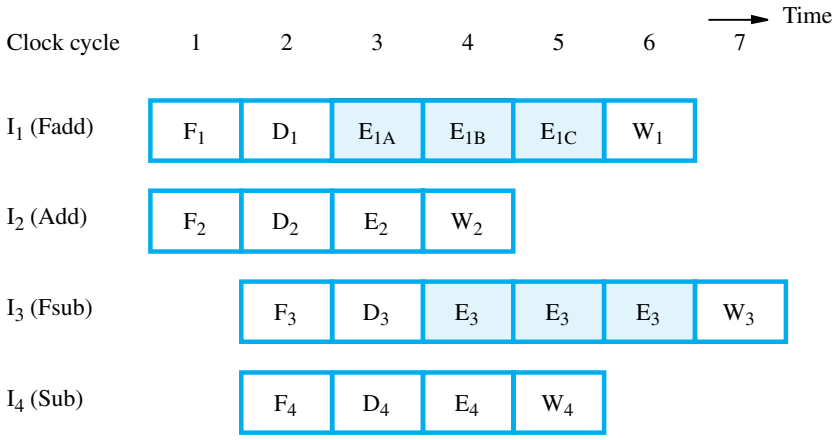


Figure 8.20 An example of instruction execution flow in the processor of Figure 8.19, assuming no hazards are encountered.

units busy most of the time. In general, high performance is achieved if the compiler is able to arrange program instructions to take maximum advantage of the available hardware units.

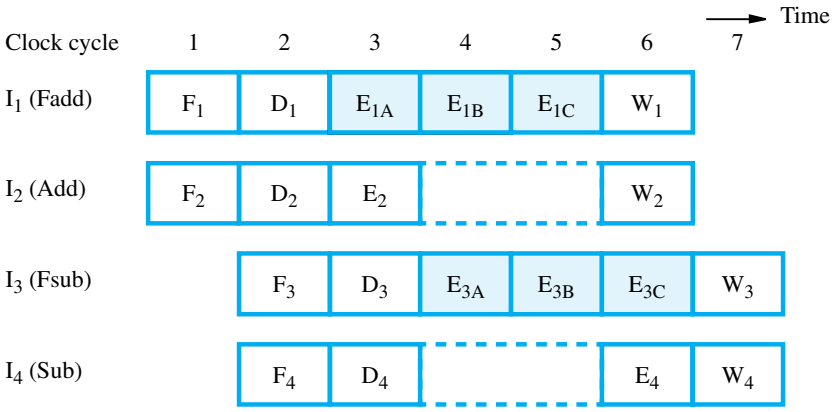
Pipeline timing is shown in Figure 8.20. The blue shading indicates operations in the floating-point unit. The floating-point unit takes three clock cycles to complete the floating-point operation specified in I_1. The integer unit completes execution of I_2 in one clock cycle. We have also assumed that the floating-point unit is organized internally as a three-stage pipeline. Thus, it can still accept a new instruction in each clock cycle. Hence, instructions I_3 and I_4 enter the dispatch unit in cycle 3, and both are dispatched in cycle 4. The integer unit can receive a new instruction because instruction I_2 has proceeded to the Write stage. Instruction I_1 is still in the execution phase, but it has moved to the second stage of the internal pipeline in the floating-point unit. Therefore, instruction I_3 can enter the first stage. Assuming that no hazards are encountered, the instructions complete execution as shown.

8.6.1 OUT-OF-ORDER EXECUTION

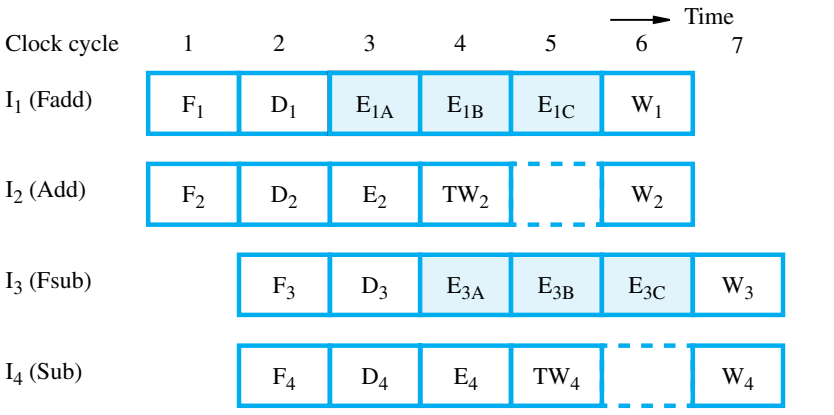
In Figure 8.20, instructions are dispatched in the same order as they appear in the program. However, their execution is completed out of order. Does this lead to any problems? We have already discussed the issues arising from dependencies among instructions. For example, if instruction I_2 depends on the result of I_1, the execution of I_2 will be delayed. As long as such dependencies are handled correctly, there is no reason to delay the execution of an instruction. However, a new complication arises when we consider the possibility of an instruction causing an exception. Exceptions may be caused by a bus error during an operand fetch or by an illegal operation, such as an attempt to divide by zero. The results of I_2 are written back into the register file in

cycle 4. If instruction I_1 causes an exception, program execution is in an inconsistent state. The program counter points to the instruction in which the exception occurred. However, one or more of the succeeding instructions have been executed to completion. If such a situation is permitted, the processor is said to have *imprecise exceptions*.

To guarantee a consistent state when exceptions occur, the results of the execution of instructions must be written into the destination locations strictly in program order. This means we must delay step W_2 in Figure 8.20 until cycle 6. In turn, the integer execution unit must retain the result of instruction I_2, and hence it cannot accept instruction I_4 until cycle 6, as shown in Figure 8.21*a*. If an exception occurs during an instruction,



(a) Delayed write



(b) Using temporary registers

Figure 8.21 Instruction completion in program order.

all subsequent instructions that may have been partially executed are discarded. This is called a *precise exception.*

It is easier to provide precise exceptions in the case of external interrupts. When an external interrupt is received, the Dispatch unit stops reading new instructions from the instruction queue, and the instructions remaining in the queue are discarded. All instructions whose execution is pending continue to completion. At this point, the processor and all its registers are in a consistent state, and interrupt processing can begin.

8.6.2 EXECUTION COMPLETION

It is desirable to use out-of-order execution, so that an execution unit is freed to execute other instructions as soon as possible. At the same time, instructions must be completed in program order to allow precise exceptions. These seemingly conflicting requirements are readily resolved if execution is allowed to proceed as shown in Figure 8.20, but the results are written into temporary registers. The contents of these registers are later transferred to the permanent registers in correct program order. This approach is illustrated in Figure 8.21*b*. Step TW is a write into a temporary register. Step W is the final step in which the contents of the temporary register are transferred into the appropriate permanent register. This step is often called the *commitment* step because the effect of the instruction cannot be reversed after that point. If an instruction causes an exception, the results of any subsequent instruction that has been executed would still be in temporary registers and can be safely discarded.

A temporary register assumes the role of the permanent register whose data it is holding and is given the same name. For example, if the destination register of I_2 is R5, the temporary register used in step TW_2 is treated as R5 during clock cycles 6 and 7. Its contents would be forwarded to any subsequent instruction that refers to R5 during that period. Because of this feature, this technique is called *register renaming*. Note that the temporary register is used only for instructions that *follow* I_2 in program order. If an instruction that precedes I_2 needs to read R5 in cycle 6 or 7, it would access the actual register R5, which still contains data that have not been modified by instruction I_2.

When out-of-order execution is allowed, a special control unit is needed to guarantee in-order commitment. This is called the *commitment unit*. It uses a queue called the *reorder buffer* to determine which instruction(s) should be committed next. Instructions are entered in the queue strictly in program order as they are dispatched for execution. When an instruction reaches the head of that queue and the execution of that instruction has been completed, the corresponding results are transferred from the temporary registers to the permanent registers and the instruction is removed from the queue. All resources that were assigned to the instruction, including the temporary registers, are released. The instruction is said to have been *retired* at this point. Because an instruction is retired only when it is at the head of the queue, all instructions that were dispatched before it must also have been retired. Hence, instructions may complete execution out of order, but they are retired in program order.

8.6.3 DISPATCH OPERATION

We now return to the dispatch operation. When dispatching decisions are made, the dispatch unit must ensure that all the resources needed for the execution of an instruction are available. For example, since the results of an instruction may have to be written in a temporary register, the required register must be free, and it is reserved for use by that instruction as a part of the dispatch operation. A location in the reorder buffer must also be available for the instruction. When all the resources needed are assigned, including an appropriate execution unit, the instruction is dispatched.

Should instructions be dispatched out of order? For example, if instruction I_2 in Figure 8.20*b* is delayed because of a cache miss for a source operand, the integer unit will be busy in cycle 4, and I_4 cannot be dispatched. Should I_5 be dispatched instead? In principle this is possible, provided that a place is reserved in the reorder buffer for instruction I_4 to ensure that all instructions are retired in the correct order. Dispatching instructions out of order requires considerable care. If I_5 is dispatched while I_4 is still waiting for some resource, we must ensure that there is no possibility of a deadlock occurring.

A *deadlock* is a situation that can arise when two units, A and B, use a shared resource. Suppose that unit B cannot complete its task until unit A completes its task. At the same time, unit B has been assigned a resource that unit A needs. If this happens, neither unit can complete its task. Unit A is waiting for the resource it needs, which is being held by unit B. At the same time, unit B is waiting for unit A to finish before it can release that resource.

If instructions are dispatched out of order, a deadlock can arise as follows. Suppose that the processor has only one temporary register, and that when I_5 is dispatched, that register is reserved for it. Instruction I_4 cannot be dispatched because it is waiting for the temporary register, which, in turn, will not become free until instruction I_5 is retired. Since instruction I_5 cannot be retired before I_4, we have a deadlock.

To prevent deadlocks, the dispatcher must take many factors into account. Hence, issuing instructions out of order is likely to increase the complexity of the Dispatch unit significantly. It may also mean that more time is required to make dispatching decisions. For these reasons, most processors use only in-order dispatching. Thus, the program order of instructions is enforced at the time instructions are dispatched and again at the time instructions are retired. Between these two events, the execution of several instructions can proceed at their own speed, subject only to any interdependencies that may exist among instructions.

In the next section, we present the UltraSPARC II as a case study of a commercially successful, superscalar, highly pipelined processor. The way in which the various issues raised in this chapter have been handled in this processor and the choices made are highly instructive.

8.7 UltraSPARC II EXAMPLE

Processor design has advanced greatly in recent years. The classification of processors as either purely RISC or CISC is no longer appropriate because modern high-performance processors contain elements of both design styles.

The early RISC processors showed how certain features can contribute to high performance. The following two observations proved to be particularly important:

- Pipelining, which enables a processor to execute several instructions at the same time, can lead to significant performance enhancements provided that the pipeline is not stalled frequently.
- A close synergy between the hardware and compiler design enables the compiler to take maximum advantage of the pipelined structure by reducing the events that lead to pipeline stalls.

It is these factors, rather than simply a reduced instruction set, that have contributed to the success of RISC processors. Of particular importance in this regard is the close coordination between the design of the hardware, particularly the structure of the pipeline, and the compiler. Much of the credit for today's high levels of performance goes to developments in compiler technology, which in turn have led to new hardware features that would have been of little use a few years ago.

The SPARC architecture, which is the basis for the processors used in Sun workstations, is an excellent case in point. One of Sun's implementations of the SPARC architecture is called UltraSPARC II. This is the processor we will discuss. We have chosen it instead of one of the processors presented in Chapter 3 because it illustrates very well superscalar operation as well as most of the pipeline design options and trade-offs discussed in this chapter. We will start with a brief introduction to the SPARC architecture. For a complete description, the reader should consult the SPARC Architecture Manual [1].

8.7.1 SPARC Architecture

SPARC stands for Scalable Processor ARChitecture. It is a specification of the instruction set architecture of a processor, that is, it is a specification of the processor's instruction set and register organization, regardless of how these may be implemented in hardware. Furthermore, SPARC is an "open architecture," which means that computer companies other than Sun Microsystems can develop their own hardware to implement the same instruction set.

The SPARC architecture was first announced in 1987, based on ideas developed at the University of California at Berkeley in the early eighties, in a project that coined the name reduced instruction set computer and its corresponding acronym RISC. The Sun Corporation and several other processor chip manufacturers have designed and built many processors based on this architecture, covering a wide range of performance. The SPARC architecture specifications are controlled by an international consortium, which introduces new enhanced versions every few years. The most recent version is SPARC-V9.

The instruction set of the SPARC architecture has a distinct RISC style. The architecture specifications describe a processor in which data and memory addresses are 64 bits long. Instructions are of equal length, and they are all 32 bits long. Both integer and floating-point instructions are provided.

There are two register files, one for integer data and one for floating-point data. Integer registers are 64 bits long. Their number is implementation dependent and can vary from 64 to 528. SPARC uses a scheme known as *register windows*. At any given time, an application program sees only 32 registers, called R0 to R31. Of these, the first eight are global registers that are always accessible. The remaining 24 registers are local to the current context.

Floating-point registers are only 32 bits long because this is the length of single-precision floating-point numbers according to the IEEE Standard described in Chapter 6. The instruction set includes floating-point instructions for double- and quad-precision operations. Two sequentially numbered floating-point registers are used to hold a double-precision operand and four are used for quad precision. There is a total of 64 registers, F0 to F63. Single precision operands can be stored in F0 to F31, double precision operands in F0, F2, F4, . . . , F62, and quad-precision in F0, F4, F8, . . . , F60.

Load and Store Instructions

Only load and store instructions access the memory, where an operand may be an 8-bit byte, a 16-bit half word, or a 32-bit word. Load and store instructions also handle 64-bit quantities, which come in two varieties: extended word or doubleword. An LDX (Load extended) instruction loads a 64-bit quantity, called an *extended* word, into one of the processor's integer registers. A *doubleword* consists of two 32-bit words. The two words are loaded into two sequentially numbered processor registers using a single LDD (Load double) instruction. They are loaded into the low-order 32 bits of each register, and the high order bits are filled with 0s. The first of the two registers, which is the register named in the instruction, must be even numbered. Load and store instructions that handle doublewords are useful for moving multiple-precision floating-point operands between the memory and floating-point registers.

Load and store instructions use one of two indexed addressing modes, as follows:

1. The effective address is the sum of the contents of two registers:

$$EA = [Radr1] + [Radr2]$$

2. The effective address is the sum of the contents of one register plus an immediate operand that is included in the instruction

$$EA = [Radr1] + Immediate$$

For most instructions, the immediate operand is a signed 13-bit value. It is sign-extended to 64 bits and then added to the contents of Radr1.

A load instruction that uses the first addressing mode is written as

```
Load    [Radr1+Radr2], Rdst
```

It generates the effective address [Radr1] + [Radr2] and loads the contents of that location into register Rdst. For an immediate displacement, Radr2 is replaced with the

immediate operand value, which yields

Load [Radr1+Imm], Rdst

Store instructions use a similar syntax, with the first operand specifying the source register from which data will be stored in the memory, as follows:

Store Rsrc, [Radr1+Radr2]

Store Rsrc, [Radr1+Imm]

In the recommended syntax for SPARC instructions, a register is specified by a % sign followed by the register number. Either %r2 or %2 refers to register number 2. However, for better readability and consistency with earlier chapters, we will use R0, R1, and so on, to refer to integer registers and F0, F1, . . . for floating-point registers.

As an example, consider the Load unsigned byte instruction

LDUB [R2+R3], R4

This instruction loads one byte from memory location [R2] + [R3] into the low-order 8 bits of register R4, and fills the high-order 56 bits with 0s. The Load signed word instruction:

LDSW [R2+2500], R4

reads a 32-bit word from location [R2] + 2500, sign extends it to 64 bits, and then stores it in register R4.

Arithmetic and Logic Instructions

The usual set of arithmetic and logic instructions is provided. A few examples are shown in Table 8.1. We pointed out in Section 8.4.2 that an instruction should set the condition code flags only when these flags are going to be tested by a subsequent conditional branch instruction. This maximizes the flexibility the compiler has in reordering instructions to avoid stalling the pipeline. The SPARC instruction set has been designed with this feature in mind. Arithmetic and logic instructions are available in two versions, one sets the condition code flags and the other does not. The suffix cc in an OP code is used to indicate that the flags should be set. For example, the instructions ADD, SUB, SMUL (signed multiply), OR, and XOR do not affect the flags, while ADDcc and SUBcc do.

Register R0 always contains the value 0. When it is used as the destination operand, the result of the instruction is discarded. For example, the instruction

SUBcc R2, R3, R0

subtracts the contents of R3 from R2, sets the condition code flags, and discards the result of the subtraction operation. In effect, this is a compare instruction, and it has the alternative syntax

CMP R2, R3

In the SPARC nomenclature, CMP is called a synthetic instruction. It is not a real

Table 8.1 Examples of SPARC instructions

Instruction		Description
ADD	R5, R6, R7	Integer add: R7 ← [R5] + [R6]
ADDcc	R2, R3, R5	R5 ← [R2] + [R3], set condition code flags
SUB	R5, Imm, R7	Integer subtract: R7 ← [R5] − Imm(sign-extended)
AND	R3, Imm, R5	Bitwise AND: R5 ← [R3] AND Imm(sign-extended)
XOR	R3, R4, R5	Bitwise Exclusive OR: R5 ← [R3] XOR [R4]
FADDq	F4, F12, F16	Floating-point add, quad precision: F12 ← [F4] + [F12]
FSUBs	F2, F5, F7	Floating-point subtract, single precision: F7 ← [F2] − [F5]
FDIVs	F5, F10, F18	Floating-point divide, single precision, F18 ← [F5]/[F10]
LDSW	R3, R5, R7	R7 ← 32-bit word at [R3] + [R5] sign extended to a 64-bit value
LDX	R3, R5, R7	R7 ← 64-bit extended word at [R3] + [R5]
LDUB	R4, Imm, R5	Load unsigned byte from memory location [R4] + Imm, the byte is loaded into the least significant 8 bits of register R5, and all higher-order bits are filled with 0s
STW	R3, R6, R12	Store word from register R3 into memory location [R6] + [R12]
LDF	R5, R6, F3	Load a 32-bit word at address [R5] + [R6] into floating-point register F3
LDDF	R5, R6, F8	Load doubleword (two 32-bit words) at address [R5] + [R6] into floating-point registers F8 and F9
STF	F14, R6, Imm	Store word from floating-register F14 into memory location [R6] + Imm
BLE	icc, Label	Test the icc flags and branch to Label if less than or equal to zero
BZ,pn	xcc, Label	Test the xcc flags and branch to Label if equal to zero, branch is predicted not taken
BGT,a,pt	icc, Label	Test the 32-bit integer condition codes and branch to Label if greater than zero, set annul bit, branch is predicted taken
FBNE,pn	Label	Test floating-point status flags and branch if not equal, the annul bit is set to zero, and the branch is predicted not taken

instruction recognized by the hardware. It is provided only for the convenience of the programmer. The assembler replaces it with a SUBcc instruction.

A condition code register, CCR, is provided, which contains two sets of condition code flags, *icc* and *xcc,* for integer and extended condition codes, respectively. Each set consists of four flags N, Z, V, and C. Instructions that set the condition code flags, such as ADDcc, will set both the *icc* and *xcc* bits; the *xcc* flags are set based on the 64-bit result of the instruction, and the *icc* flags are set based on the low-order 32 bits only.

The condition codes for floating-point operations are held in a 64-bit register called the floating-point state register, FSR.

Branch Instructions

The way in which branches are handled is an important factor in determining performance. Branch instructions in the SPARC instruction set contain several features that are intended to enhance performance of a pipelined processor and to help the compiler in optimizing the code it emits.

A SPARC processor uses delayed branching with one delay slot (see Section 8.3.2). Branch instructions include a branch prediction bit, which the compiler can use to give the hardware a hint about the expected behavior of the branch. Branch instructions also contain an Annul bit, which is intended to increase flexibility in handling the instruction in the delay slot. This instruction is always executed, but its results are not committed until after the branch decision is known. If the branch is taken, execution of the instruction in the delay slot is completed and the results are committed. If the branch is not taken, this instruction is annulled if the Annul bit is equal to 1. Otherwise, execution of the instruction is completed.

The compiler may be able to place in the delay slot an instruction that is needed whether or not the branch is taken. This may be an instruction that logically belongs before the branch instruction but can be moved into the delay slot. The Annul bit should be set to 0 in this case. Otherwise, the delay slot should be filled with an instruction that is to be executed only if the branch is taken, in which case the Annul bit should be set to 1.

Conditional branch instructions can test the *icc, xcc,* or FSR flags. For example, the instruction

```
BGT,a,pt    icc, Label
```

will cause a branch to location Label if the previous instruction that set the flags in *icc* produced a greater-than-zero result. The instruction will have both the Annul bit and the branch prediction bit set to 1. The instruction

```
FBGT,a,pt    Label
```

is exactly the same, except that it will test the FSR flags. If neither pt (predicted taken) nor pn (predicted not taken) is specified, the assembler will default to pt.

An example that illustrates the prediction and annul facilities in branch instructions is given in Figure 8.22, which shows a program loop that adds a list of n 64-bit integers. We have assumed that the number of items in the list is stored at address LIST as a 64-bit integer, followed by the numbers to be added in successive 64-bit locations. We have also assumed that there is at least one item in the list and that the address LIST has been loaded into register R3 earlier in the program.

Figure 8.22*a* shows the desired loop as it would be written for execution on a nonpipelined processor. For execution on a SPARC processor, we should first reorganize the instructions to make effective use of the branch delay slot. Observe that the ADD instruction following LOOPSTART is executed during every pass through the loop.

```
            LDX     R3, 0, R6           Load number of items in the list.
            OR      R0, R0, R4          R4 to be used as offset in the list
            OR      R0, R0, R7          Clear R7 to be used as accumulator.
LOOPSTART   LDX     R3, R4, R5          Load list item into R5.
            ADD     R5, R7, R7          Add number to accumulator.
            ADD     R4, 8, R4           Point to the next entry.
            SUBcc   R6, 1, R6           Decrement R6 and set condition flags.
            BG      xcc, LOOPSTART      Loop if more items in the list.
NEXT        ...
```

(a) Desired program loop

```
            LDX     R3, 0, R6
            OR      R0, R0, R4
            OR      R0, R0, R7
LOOPSTART   LDX     R3, R4, R5
            ADD     R4, 8, R4
            SUBcc   R6, 1, R6
            BG,pt   xcc, LOOPSTART      Predicted taken, Annul bit = 0
            ADD     R5, R7, R7
NEXT        ...
```

(b) Instructions reorganized to use the delay slot

Figure 8.22 An addition loop showing the use of the branch delay slot and branch prediction.

Also, none of the instructions following it depends on its result. Hence, this instruction may be moved into the delay slot following the branch at the end of the loop, as shown in Figure 8.22*b*. Since it is to be executed regardless of the branch outcome, the Annul bit in the branch instruction is set to 0 (this is the default condition).

As for branch prediction, observe that the number of times the loop will be executed is equal to the number of items in the list. This means that, except for the trivial case of $n = 1$, the branch will be taken a number of times before exiting the loop. Hence, we have set the branch prediction bit in the BG instruction to indicate that the branch is expected to be taken.

Conditional branch instructions are not the only instructions that check the condition code flags. For example, there is a conditional move instruction, MOVcc, which copies data from one register into another only if the condition codes satisfy the condition specified in the instruction suffix, cc. Consider the two instructions

```
CMP     R5, R6
MOVle   icc, R5, R6
```

The MOVle instruction copies the contents of R5 into R6 if the condition code flags in *icc* indicate a less-than-or-equal-to condition ($Z + (N \oplus V) = 1$). The net result is

to place the smaller of the two values in register R6. In the absence of a conditional move instruction, the same task would require a branch instruction, as in the following sequence

```
            CMP     R5, R6
            BG      icc, GREATER
            MOVA    icc, R5, R6
GREATER     ...
```

where MOVA is the move-always instruction. The MOVle instruction not only reduces the number of instructions needed, but more importantly, it avoids the performance degradation caused by branch instructions in pipelined execution.

The instruction set has many other features that are intended to maximize performance in a highly pipelined superscalar processor. We will discuss some of these features in the context of the UltraSPARC II processor. The ideas behind these features have already been introduced earlier in the chapter.

8.7.2 UltraSPARC II

The main building blocks of the UltraSPARC II processor are shown in Figure 8.23. The processor uses two levels of cache: an external cache (E-cache) and two internal caches, one for instructions (I-cache) and one for data (D-cache). The external cache controller is on the processor chip, as is the control hardware for memory management. The memory management unit uses two translation lookaside buffers, one for instructions, iTLB, and one for data, dTLB. The processor communicates with the memory and the I/O subsystem over the system interconnection bus.

There are two execution units, one for integer and one for floating-point operations. Each of these units contains a register set and two independent pipelines for instruction execution. Thus, the processor can simultaneously start the execution of up to four instructions, two integer and two floating-point. These four instructions proceed in parallel, each through its own pipeline. If instructions are available and none of the four pipelines is stalled, four new instructions can enter the execution phase every clock cycle.

The Prefetch and Dispatch Unit (PDU) of the processor is responsible for maintaining a continuous supply of instructions for the execution units. It does so by prefetching instructions before they are needed and placing them in a temporary storage buffer called the instruction buffer, which performs the role of the instruction queue in Figure 8.19.

8.7.3 PIPELINE STRUCTURE

The UltraSPARC II has a nine-stage instruction execution pipeline, shown in Figure 8.24. The function of each stage is completed in one processor clock cycle. We will give an overview of the operation of the pipeline, then discuss each stage in detail.

The first three stages of the pipeline are common to all instructions. Instructions

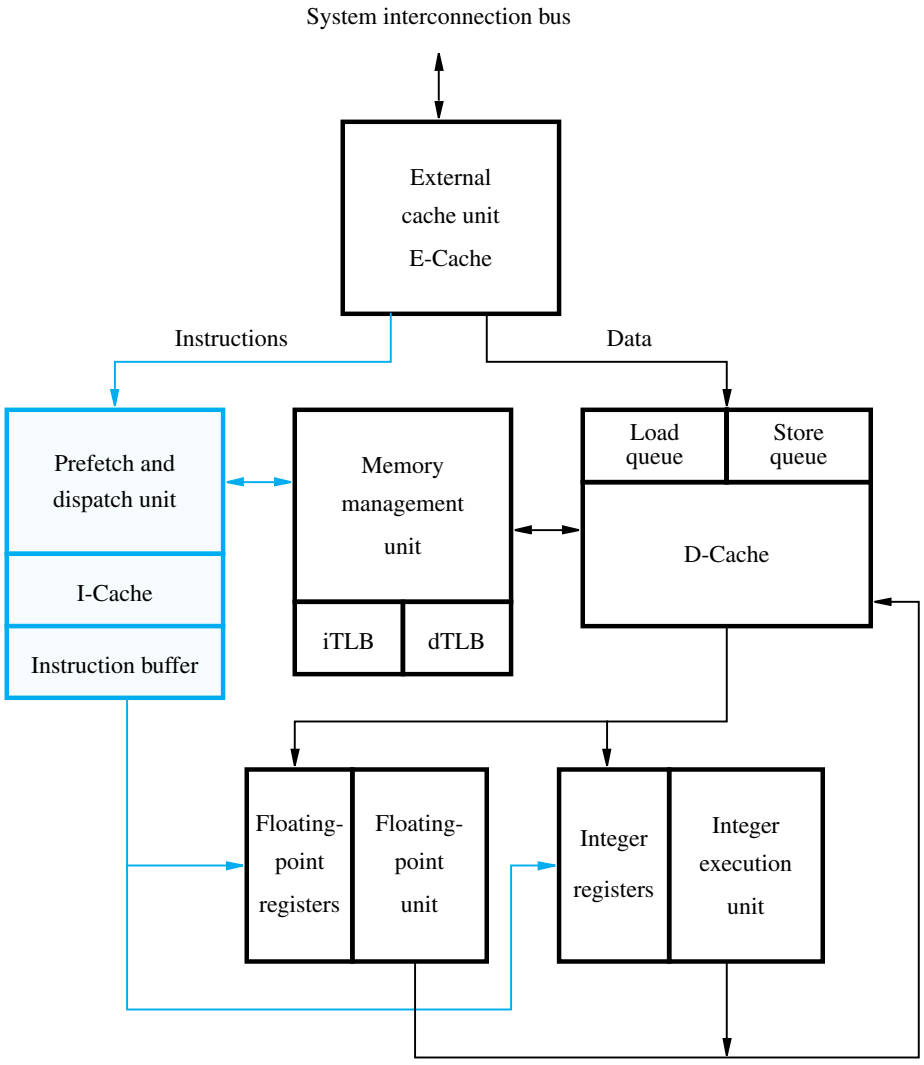


Figure 8.23 Main building blocks of the UltraSPARC II processor.

are fetched from the instruction cache in the first stage (F) and partially decoded in the second stage (D). Then, in the third stage (G), a group of up to four instructions is selected for execution in parallel. The instructions are then dispatched to the integer and floating-point execution units.

Each of the two execution units consists of two parallel pipelines with six stages each. The first four stages are available to perform the operation specified by the instruction, and the last two are used to check for exceptions and store the result of the instruction.

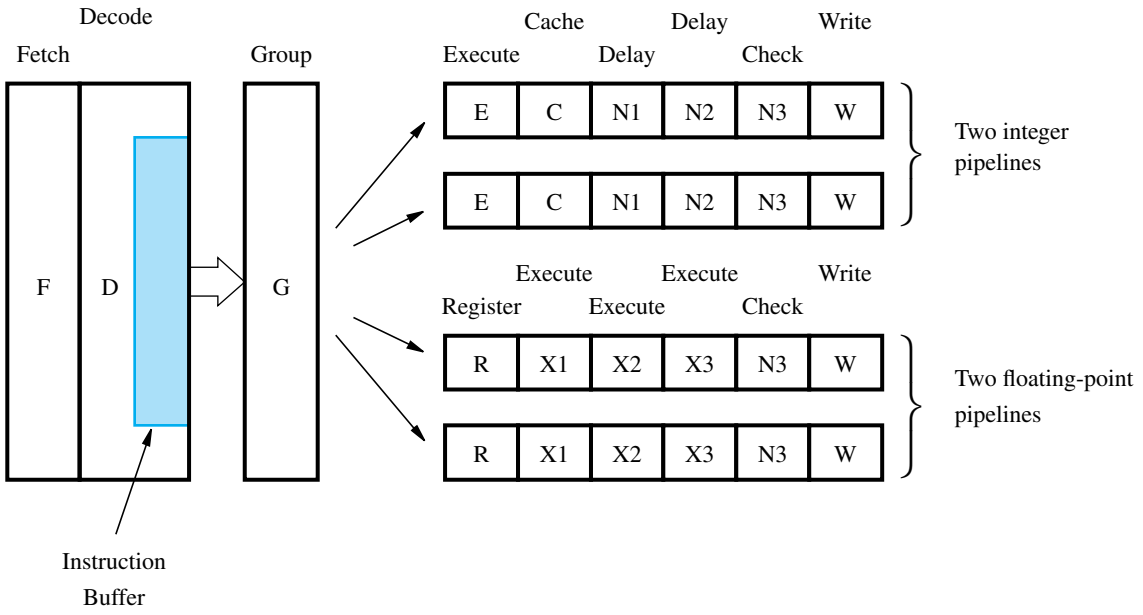


Figure 8.24 Pipeline organization of the UltraSPARC II processor.

Instruction Fetch and Decode

The PDU fetches up to four instructions from the instruction cache, partially decodes them, and stores the results in the instruction buffer, which can hold up to 12 instructions. The decoding that takes place in this stage enables the PDU to determine whether the instruction is a branch instruction. It also detects salient features that can be used to speed up the decisions to be made later in the pipeline.

A cache block in the instruction cache consists of 32 bytes. It contains eight instructions. As instructions are loaded into the cache they are stored based on their virtual addresses, so that they can be fetched quickly by the PDU without requiring address translation. The PDU can maintain the rate of four instructions per cycle as long as each group does not cross cache block boundaries. If there are fewer than four instructions left in a cache block, the unit will read only the remaining instructions in the current block.

The PDU uses a four-state branch prediction algorithm similar to that described in Figure 8.15. It uses the branch prediction bit in the branch instruction to set the initial state to either LT or LNT. For every two instructions in the instruction cache, the PDU uses two bits to record the state of the branch prediction algorithm. These bits are stored in the cache, in a tag associated with the instructions.

For each four instructions in the instruction cache, a tag field is provided called Next Address. The PDU computes the target address of a branch instruction when the instruction is first fetched for execution, and it records this address in the Next Address field. This field makes it possible to continue prefetching instructions in subsequent passes, without having to recompute the target address each time. Since there is only

one Next Address field for each half of a cache line, its benefit can be fully realized only if there is at most one branch instruction in each group of four instructions.

Grouping

In the third stage of the pipeline, stage G, the Grouping Logic selects a group of up to four instructions to be executed in parallel and dispatches them to the integer and floating-point execution units. Figure 8.25 shows a short instruction sequence and the way these instructions would be dispatched. Parts *b* and *c* of the figure show the instruction grouping when the PDU predicts that the branch will be taken and not taken, respectively. Note that the instruction in the delay slot, FCMP, is included in the selected group in both cases. It will be executed, but not committed until the branch decision is made. Its results will be annulled if the branch is not taken, because the Annul bit in the branch instruction is set to 1. The first two instructions in each group are dispatched to the integer unit and the next two to the floating-point unit.

```
        ADDcc   R3, R4, R7   R7 ← [R3] + [R4],
                             Set condition codes
        BRZ,a   Label        Branch if zero, set Annul bit to 1
        FCMP    F1, F5       FP: Compare [F2] and [F5]
        FADD    F2, F3, F6   FP: F6 ← [F2] + [F3]
        FMOVs   F3, F4       Move single precision operand from F3 to F4
        ⋮
Label   FSUB    F2, F3, F6   FP: F6 ← [F2] – [F3]
        LDSW    R3, R4, R7   Load single word at location [R3] + [R4] into R7
        ⋮
```

(a) Program fragment

```
ADDcc   R3, R4, R7
BRZ,a   Label
FCMP    F1, F5
FSUB    F2, F3, F6
```

(b) Instruction grouping, branch taken

```
ADDcc   R3, R4, R7
BRZ,a   Label
FCMP    R1, R5
FADD    R2, R3, R6
```

(c) Instruction grouping, branch not taken

Figure 8.25 Example of instruction grouping.

The grouping logic circuit is responsible for ensuring that the instructions it dispatches are ready for execution. For example, all the operands referenced by the instruction in a group must be available. No two instructions can be included in the same group if one of them depends on the result of the other. Branch instructions are excepted from this condition, as will be explained shortly.

Instructions are dispatched in program order. Recall that if a group includes a branch instruction, that instruction will have already been tentatively executed as a result of branch prediction in the prefetch and decode unit. Hence, the instructions in the instruction buffer will be in correct order based on this prediction. The grouping logic simply examines the instructions in the instruction buffer in order, with the objective of selecting the largest number at the head of the queue that satisfy the grouping constraints.

Some of the constraints that the grouping logic takes into account in selecting instructions to include in a group are:

1. Instructions can only be dispatched in sequence. If one instruction cannot be included in a group, no later instruction can be selected.

2. The source operand of an instruction cannot depend on the destination operand of any other instruction in the same group. There are two exceptions to this rule:

- A store instruction, which stores the contents of a register in the memory, may be grouped with an earlier instruction that has that register as a destination. This is allowed because, as we will see shortly, the store instruction does not require the data until a later stage in the pipeline.
- A branch instruction may be grouped with an earlier instruction that sets the condition codes.

3. No two instructions in a group can have the same destination operand, unless the destination is register R0. For example, the LDSW instruction in Figure 8.26*a* cannot be grouped with the ADD instruction and must be delayed to the next group as shown.

4. In some cases, certain instructions must be delayed two or three clock cycles relative to other instructions. For example, the conditional instruction

```
MOVRZ   R1, R6, R7
```

(Move on register condition) moves the contents of R6 into R7 if the contents of R1 are equal to zero. This instruction requires an additional clock cycle to check if the contents

```
ADD     R3, R5, R6      G   E   C   N1  N2  N3  W
LDSW    R4, R7, R6          G   E   C   N1  N2  N3  W
```

(a) Instructions with common destination

```
MOVRZ   R1, R6, R7      G   E   C   N1  N2  N3  W
OR      R7, R8, R9              G   E   C   N1  N2  N3  W
```

(b) Delay caused by MOVR instruction

Figure 8.26 Dispatch delays due to hazards.

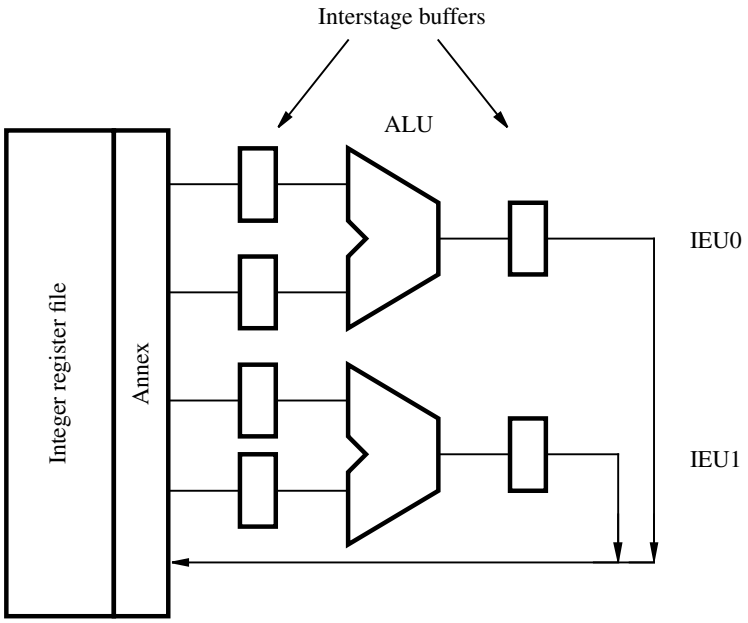


Figure 8.27 Integer execution unit.

of R1 are equal to zero. Hence, an instruction that reads register R7 cannot be in the same group or in the following group. The earliest dispatch for such an instruction is as shown in Figure 8.26*b*.

When the grouping logic dispatches an instruction to the integer unit, it also fetches the source operands of that instruction from the integer register file. The information needed to access the register file is available in the decoded bits that were entered into the instruction buffer by the prefetch and decode unit. Thus, by the end of the clock cycle of stage G, one or two integer instructions will be ready to enter the execution phase. The data read from the register file are stored in interstage buffers, as shown in Figure 8.27. Access to operands in the floating-point register file takes place in stage R, after the instruction has been forwarded to the floating-point unit.

Execution Units

The Integer execution unit consists of two similar but not identical units, IEU0 and IEU1. Only unit IEU0 is equipped to handle shift instructions, while only IEU1 can generate condition codes. Instructions that do not involve these operations can be executed in either unit.

The ALU operation for most integer instructions is completed in one clock cycle. This is stage E in the pipeline. At the end of this clock cycle, the result is stored in the buffer shown at the output of the ALU in Figure 8.27. In the next clock cycle, stage C, the contents of this buffer are transferred to a part of the register file called the Annex. The Annex contains the temporary registers used in register renaming, as explained in

Section 8.6. The contents of a temporary register are transferred to the corresponding permanent register in stage W of the pipeline.

Another action that takes place during stage C is the generation of condition codes. Of course, this is done only for instructions such as ADDcc, which specify that the condition code flags are to be set. Such instructions must be executed in unit IEU1.

Consider an instruction Icc that sets the condition code flags and a subsequent conditional branch instruction, BRcc, that checks these flags. When BRcc is encountered by the prefetch and dispatch unit, the results of execution of Icc may not yet be available. The PDU predicts the outcome of the branch and continues prefetching instructions on that basis. Later, the condition codes are generated when Icc reaches stage C of the pipeline, and they are sent to the PDU during the same clock cycle. The PDU checks whether its branch prediction was correct. If it was, execution continues without interruption. Otherwise, the contents of the pipeline and the instruction buffer are flushed, and the PDU begins to fetch the correct instructions. Aborting instructions at this point is possible because these instructions will not have reached stage W of the pipeline.

When a branch is incorrectly predicted, many instructions may be incorrectly prefetched and partially executed. The situation is illustrated in Figure 8.28. We have assumed that the grouping logic has been able to dispatch four instructions in three successive clock cycles. Instruction Icc at the beginning of the first group sets the condition codes, which are tested by the following instruction, BRcc. The test is performed when the first group reaches stage C. At this time, the third group, I_9 to I_{12}, is entering stage G of the pipeline. If the branch prediction was incorrect, the nine instructions I_4 to I_{12} will be aborted (recall that instruction I_3 in the delay slot is always executed). In addition, any instructions that may have been prefetched and loaded into the instruction buffer will also be discarded. Hence, in the extreme case, up to 21 intrustions may be discarded.

No operation is performed in pipeline stages N1 and N2. These stages introduce a delay of two clock cycles, to make the total length of the integer pipeline the same

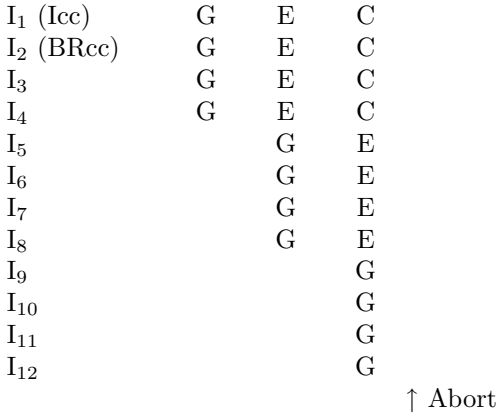


Figure 8.28 Worst-case timing for an incorrectly predicted branch.

as that of the floating-point pipeline. For integer instructions that do not complete their execution in stage C, such as divide instructions, execution continues through stages N1 and N2. If more time is needed, additional clock cycles are inserted between N1 and N2. The instruction enters N2 only in the last clock cycle of its execution. For example, if the operation performed by an instruction requires 16 clock cycles, 12 clock cycles are inserted after stage N1.

The Floating-point execution unit also has two independent pipelines. Register operands are fetched in stage R, and the operation is performed in up to three pipeline stages (X1 to X3). Here also, if additional clock cycles are needed, such as for the square-root instruction, additional clock cycles are inserted between X2 and X3.

In stage N3, the processor examines various exception conditions to determine whether a trap (interrupt) should be taken. Finally, the result of an instruction is stored in the destination location, either in a register or in the data cache, during the Write stage (W). An instruction may be aborted and all its effects annulled at any time up to this stage. Once the Write stage is entered, the execution of the instruction cannot be stopped.

Load and Store Unit

The instruction

LDUW R5, R6, R7

loads an unsigned 32-bit word from location [R5] + [R6] in the memory into register R7. As for other integer instructions, the contents of registers R5 and R6 are fetched during stage G of the pipeline. However, instead of this data being sent to one of the integer execution units, the instruction and its operands are forwarded to the Load and Store Unit, shown in Figure 8.29. The unit begins by adding the contents of registers R5 and R6 during stage E to generate the effective address of the memory location to be accessed. The result is a virtual address value, which is sent to the data cache. At the same time, it is sent to the data lookaside buffer, dTLB, to be translated into a physical address.

Data are stored in the cache according to their virtual address, so that they can be accessed quickly without waiting for address translation to be completed. Both the data and the corresponding tag information are read from the D-cache in stage C, and the physical address is read from the dTLB. The tag used in the D-cache is a part of the physical address of the data. During stage N1, the tag read from the D-cache is checked against the physical address obtained from the dTLB. In the case of a hit, the data are loaded into an Annex register, to be transferred to the destination register in stage W. If the tags do not match, the instruction enters the Load/store queue, where it waits for a cache block to be loaded from the external cache into the D-cache.

Once an instruction enters the Load/store queue it is no longer considered to be in the execution pipeline. Other instructions may proceed to completion while a load instruction is waiting in the queue, unless one of these instructions references the register awaiting data from the memory (R7 in the example above). Thus, the Load/store queue decouples the operation of the pipeline from external data access operations so that the two can proceed independently.

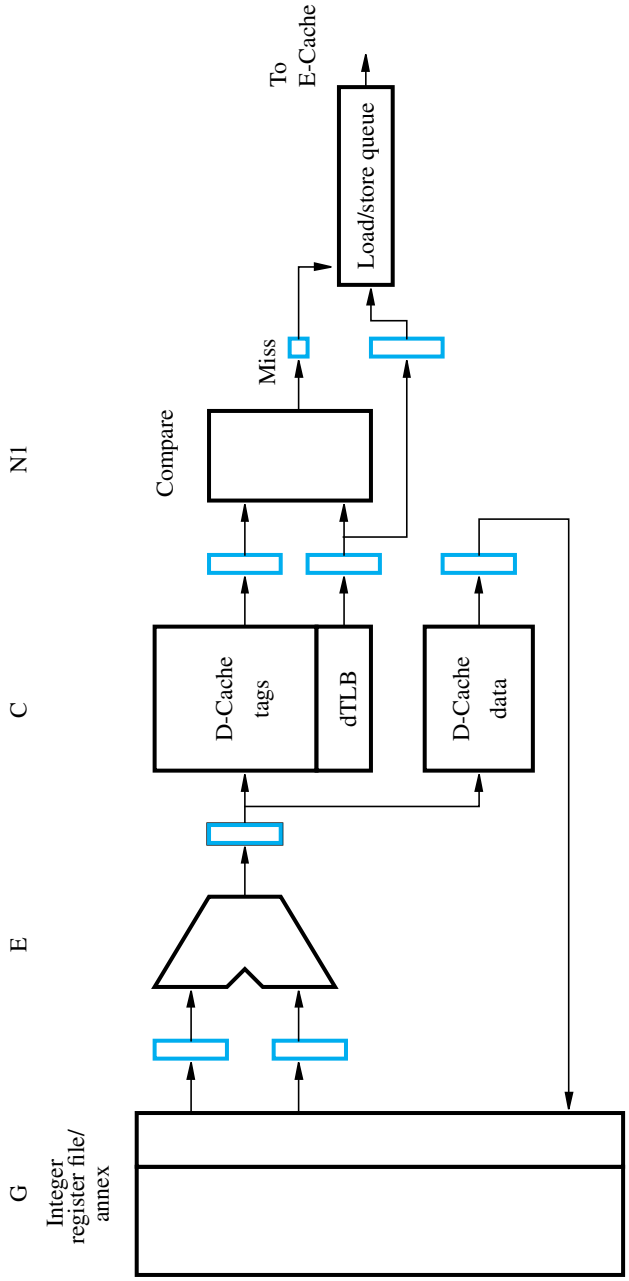


Figure 8.29 Load and store unit.

Execution Flow

It is instructive to examine the flow of instructions and data in the UltraSPARC II processor and between it and the external cache and the memory. Figure 8.30 shows the main functional units of Figure 8.23 reorganized to illustrate the flow of instructions and data and the role that the instruction and data queues play.

Instructions are fetched from the I-cache and loaded into the instruction buffer, which can store up to 12 instructions. From there, instructions are forwarded, up to four at a time, to the block labeled "Internal registers and execution units," where they are executed. On average, the speed with which the PDU can fill the instruction buffer is higher than the speed with which the grouping logic dispatches instructions. Hence, the instruction buffer tends to be full most of the time. In the absence of cache misses and mispredicted branches, the internal execution units are never starved for instructions. Similarly, the memory operands of load and store instructions are likely to be found in the data cache most of the time, where they are accessed in one clock cycle. Hence execution proceeds without delay.

When a miss occurs in the instruction cache, there is a delay of a few clock cycles while the appropriate block is loaded from the external cache. During that time, the

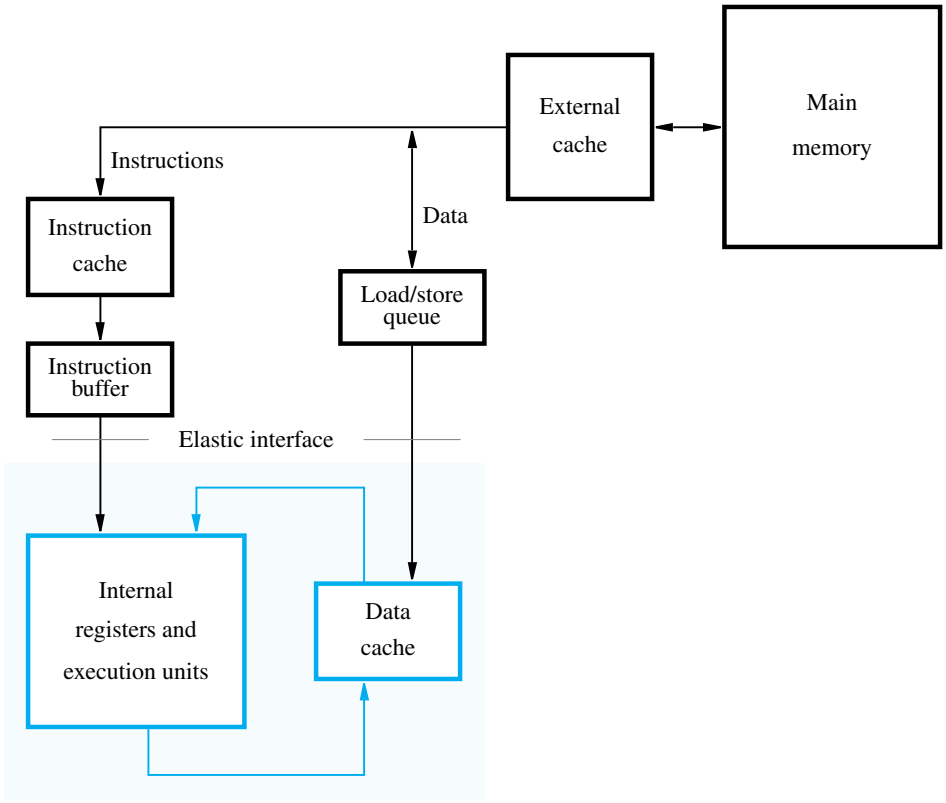


Figure 8.30 Execution flow.

grouping logic continues to dispatch instructions from the instruction buffer until the buffer becomes empty. It takes three or four clock cycles to load a cache block (eight instructions) from the external cache, depending on the processor model. This is about the same length of time it takes the grouping logic to dispatch the instructions in a full instruction buffer. (Recall that it is not always possible to dispatch four instructions in every clock cycle.) Hence, if the instruction buffer is full at the time a cache miss occurs, operation of the execution pipeline may not be interrupted at all. If a miss also occurs in the external cache, considerably more time will be needed to access the memory. In this case, it is inevitable that the pipeline will be stalled.

A load operation that causes a cache miss enters the Load/store queue and waits for a transfer from the external cache or the memory. However, as long as the destination register of the load operation is not referenced by later instructions, internal instruction execution continues. Thus, the instruction buffer and the Load/store queue isolate the internal processor pipeline from external data transfers. They act as elastic interfaces that allow the internal high-speed pipeline to continue to run while slow external data transfers are taking place.

8.8 PERFORMANCE CONSIDERATIONS

We pointed out in Section 1.6 that the execution time, T, of a program that has a dynamic instruction count N is given by

$$T = \frac{N \times S}{R}$$

where S is the average number of clock cycles it takes to fetch and execute one instruction, and R is the clock rate. This simple model assumes that instructions are executed one after the other, with no overlap. A useful performance indicator is the *instruction throughput,* which is the number of instructions executed per second. For sequential execution, the throughput, P_s is given by

$$P_s = R/S$$

In this section, we examine the extent to which pipelining increases instruction throughput. However, we should reemphasize the point made in Chapter 1 regarding performance measures. The only real measure of performance is the total execution time of a program. Higher instruction throughput will not necessarily lead to higher performance if a larger number of instructions is needed to implement the desired task. For this reason, the SPEC ratings described in Chapter 1 provide a much better indicator when comparing two processors.

Figure 8.2 shows that a four-stage pipeline may increase instruction throughput by a factor of four. In general, an n-stage pipeline has the potential to increase throughput n times. Thus, it would appear that the higher the value of n, the larger the performance gain. This leads to two questions:

- How much of this potential increase in instruction throughput can be realized in practice?
- What is a good value for n?

Any time a pipeline is stalled, the instruction throughput is reduced. Hence, the performance of a pipeline is highly influenced by factors such as branch and cache miss penalties. First, we discuss the effect of these factors on performance, and then we return to the question of how many pipeline stages should be used.

8.8.1 EFFECT OF INSTRUCTION HAZARDS

The effects of various hazards have been examined qualitatively in the previous sections. We now assess the impact of cache misses and branch penalties in quantitative terms.

Consider a processor that uses the four-stage pipeline of Figure 8.2. The clock rate, hence the time allocated to each step in the pipeline, is determined by the longest step. Let the delay through the ALU be the critical parameter. This is the time needed to add two integers. Thus, if the ALU delay is 2 ns, a clock of 500 MHz can be used. The on-chip instruction and data caches for this processor should also be designed to have an access time of 2 ns. Under ideal conditions, this pipelined processor will have an instruction throughput, P_p, given by

$$P_p = R = 500 \text{ MIPS (million instructions per second)}$$

To evaluate the effect of cache misses, we use the same parameters as in Section 5.6.2. The cache miss penalty, M_p, in that system is computed to be 17 clock cycles. Let T_I be the time between two successive instruction completions. For sequential execution, $T_I = S$. However, in the absence of hazards, a pipelined processor completes the execution of one instruction each clock cycle, thus, $T_I = 1$ cycle. A cache miss stalls the pipeline by an amount equal to the cache miss penalty. This means that the value of T_I increases by an amount equal to the cache miss penalty for the instruction in which the miss occurs. A cache miss can occur for either instructions or data. Consider a computer that has a shared cache for both instructions and data, and let d be the percentage of instructions that refer to data operands in the memory. The average increase in the value of T_I as a result of cache misses is given by

$$\delta_{miss} = ((1 - h_i) + d(1 - h_d)) \times M_p$$

where h_i and h_d are the hit ratios for instructions and data, respectively. Assume that 30 percent of the instructions access data in memory. With a 95-percent instruction hit rate and a 90-percent data hit rate, δ_{miss} is given by

$$\delta_{miss} = (0.05 + 0.3 \times 0.1) \times 17 = 1.36 \text{ cycles}$$

Taking this delay into account, the processor's throughput would be

$$P_p = \frac{R}{T_I} = \frac{R}{1 + \delta_{miss}} = 0.42R$$

Note that with R expressed in MHz, the throughput is obtained directly in millions of instructions per second. For $R = 500$ MHz, $P_p = 210$ MIPS.

Let us compare this value to the throughput obtainable without pipelining. A processor that uses sequential execution requires four cycles per instruction. Its throughput

would be

$$P_s = \frac{R}{4 + \delta_{miss}} = 0.19R$$

For $R = 500$ MHz, $P_s = 95$ MIPS. Clearly, pipelining leads to significantly higher throughput. But the performance gain of $0.42/0.19 = 2.2$ is only slightly better than one-half the ideal case.

Reducing the cache miss penalty is particularly worthwhile in a pipelined processor. As Chapter 5 explains, this can be achieved by introducing a secondary cache between the primary, on-chip cache and the memory. Assume that the time needed to transfer an 8-word block from the secondary cache is 10 ns. Hence, a miss in the primary cache for which the required block is found in the secondary cache introduces a penalty, M_s, of 5 cycles. In the case of a miss in the secondary cache, the full 17-cycle penalty (M_p) is still incurred. Hence, assuming a hit rate h_s of 94 percent in the secondary cache, the average increase in T_I is

$$\delta_{miss} = ((1 - h_i) + d(1 - h_d)) \times (h_s \times M_s + (1 - h_s) \times M_p) = 0.46 \text{ cycle}$$

The instruction throughput in this case is $0.68R$, or 340 MIPS. An equivalent non-pipelined processor would have a throughput of $0.22R$, or 110 MIPS. Thus, pipelining provides a performance gain of $0.68/0.22 = 3.1$.

The values of 1.36 and 0.46 are, in fact, somewhat pessimistic, because we have assumed that every time a data miss occurs, the entire miss penalty is incurred. This is the case only if the instruction immediately following the instruction that references memory is delayed while the processor waits for the memory access to be completed. However, an optimizing compiler attempts to increase the distance between two instructions that create a dependency by placing other instructions between them whenever possible. Also, in a processor that uses an instruction queue, the cache miss penalty during instruction fetches may have a much reduced effect as the processor is able to dispatch instructions from the queue.

8.8.2 NUMBER OF PIPELINE STAGES

The fact that an n-stage pipeline may increase instruction throughput by a factor of n suggests that we should use a large number of stages. However, as the number of pipeline stages increases, so does the probability of the pipeline being stalled, because more instructions are being executed concurrently. Thus, dependencies between instructions that are far apart may still cause the pipeline to stall. Also, branch penalties may become more significant, as Figure 8.9 shows. For these reasons, the gain from increasing the value of n begins to diminish, and the associated cost is not justified.

Another important factor is the inherent delay in the basic operations performed by the processor. The most important among these is the ALU delay. In many processors, the cycle time of the processor clock is chosen such that one ALU operation can be completed in one cycle. Other operations are divided into steps that take about the same time as an add operation. It is also possible to use a pipelined ALU. For example, the ALU of the Compaq Alpha 21064 processor consists of a two-stage pipeline, in which each stage completes its operation in 5 ns.

Many pipelined processors use four to six stages. Others divide instruction execution into smaller steps and use more pipeline stages and a faster clock. For example, the UltraSPARC II uses a 9-stage pipeline and Intel's Pentium Pro uses a 12-stage pipeline. The latest Intel processor, Pentium 4, has a 20-stage pipeline and uses a clock speed in the range 1.3 to 1.5 GHz. For fast operations, there are two pipeline stages in one clock cycle.

8.9 CONCLUDING REMARKS

Two important features have been introduced in this chapter, pipelining and multiple issue. Pipelining enables us to build processors with instruction throughput approaching one instruction per clock cycle. Multiple issue makes possible superscalar operation, with instruction throughput of several instructions per clock cycle.

The potential gain in performance can only be realized by careful attention to three aspects:

- The instruction set of the processor
- The design of the pipeline hardware
- The design of the associated compiler

It is important to appreciate that there are strong interactions among all three. High performance is critically dependent on the extent to which these interactions are taken into account in the design of a processor. Instruction sets that are particularly well-suited for pipelined execution are key features of modern processors.

PROBLEMS

8.1 Consider the following sequence of instructions

```
Add   #20,R0,R1
Mul   #3,R2,R3
And   #$3A,R2,R4
Add   R0,R2,R5
```

In all instructions, the destination operand is given last. Initially, registers R0 and R2 contain 2000 and 50, respectively. These instructions are executed in a computer that has a four-stage pipeline similar to that shown in Figure 8.2. Assume that the first instruction is fetched in clock cycle 1, and that instruction fetch requires only one clock cycle.

(*a*) Draw a diagram similar to Figure 8.2*a*. Describe the operation being performed by each pipeline stage during each of clock cycles 1 through 4.

(*b*) Give the contents of the interstage buffers, B1, B2, and B3, during clock cycles 2 to 5.

8.2 Repeat Problem 8.1 for the following program:

```
Add   #20,R0,R1
Mul   #3,R2,R3
And   #$3A,R1,R4
Add   R0,R2,R5
```

8.3 Instruction I_2 in Figure 8.6 is delayed because it depends on the results of I_1. By occupying the Decode stage, instruction I_2 blocks I_3, which, in turn, blocks I_4. Assuming that I_3 and I_4 do not depend on either I_1 or I_2 and that the register file allows two Write steps to proceed in parallel, how would you use additional storage buffers to make it possible for I_3 and I_4 to proceed earlier than in Figure 8.6? Redraw the figure, showing the new order of steps.

8.4 The delay bubble in Figure 8.6 arises because instruction I_2 is delayed in the Decode stage. As a result, instructions I_3 and I_4 are delayed even if they do not depend on either I_1 or I_2. Assume that the Decode stage allows two Decode steps to proceed in parallel. Show that the delay bubble can be completely eliminated if the register file also allows two Write steps to proceed in parallel.

8.5 Figure 8.4 shows an instruction being delayed as a result of a cache miss. Redraw this figure for the hardware organization of Figure 8.10. Assume that the instruction queue can hold up to four instructions and that the instruction fetch unit reads two instructions at a time from the cache.

8.6 A program loop ends with a conditional branch to the beginning of the loop. How would you implement this loop on a pipelined computer that uses delayed branching with one delay slot? Under what conditions would you be able to put a useful instruction in the delay slot?

8.7 The branch instruction of the UltraSPARC II processor has an Annul bit. When set by the compiler, the instruction in the delay slot is discarded if the branch is not taken. An alternative choice is to have the instruction discarded if the branch is taken. When is each of these choices advantageous?

8.8 A computer has one delay slot. The instruction in this slot is always executed, but only on a speculative basis. If a branch does not take place, the results of that instruction are discarded. Suggest a way to implement program loops efficiently on this computer.

8.9 Rewrite the sort routine shown in Figure 2.34 for the SPARC processor. Recall that the SPARC architecture has one delay slot with an associated Annul bit and uses branch prediction. Attempt to fill the delay slots with useful instructions wherever possible.

8.10 Consider a statement of the form

IF A>B THEN action 1 ELSE action 2

Write a sequence of assembly language instructions, first using branch instructions only, then using conditional instructions such as those available on the ARM processor.

Assume a simple two-stage pipeline, and draw a diagram similar to that in Figure 8.8 to compare execution times for the two approaches.

8.11 The feed-forward path in Figure 8.7 (blue lines) allows the content of the RSLT register to be used directly in an ALU operation. The result of that operation is stored back in the RSLT register, replacing its previous contents. What type of register is needed to make such an operation possible?

Consider the two instructions

I_1: Add R1,R2,R3

I_2: Shift_left R3

Assume that before instruction I_1 is executed, R1, R2, R3, and RSLT contain the values 30, 100, 45, and 198, respectively. Draw a timing diagram for a 4-stage pipeline, showing the clock signal and the contents of the RSLT register during each cycle. Use your diagram to show that correct results will be obtained during the forwarding operation.

8.12 Write the program in Figure 2.37 for a processor in which only load and store instructions access memory. Identify all dependencies in the program and show how you would optimize it for execution on a pipelined processor.

8.13 Assume that 20 percent of the dynamic count of the instructions executed on a computer are branch instructions. Delayed branching is used, with one delay slot. Estimate the gain in performance if the compiler is able to use 85 percent of the delay slots.

8.14 A pipelined processor has two branch delay slots. An optimizing compiler can fill one of these slots 85 percent of the time and can fill the second slot only 20 percent of the time. What is the percentage improvement in performance achieved by this optimization, assuming that 20 percent of the instructions executed are branch instructions?

8.15 A pipelined processor uses the delayed branch technique. You are asked to recommend one of two possibilities for the design of this processor. In the first possibility, the processor has a 4-stage pipeline and one delay slot, and in the second possibility, it has a 6-stage pipeline with two delay slots. Compare the performance of these two alternatives, taking only the branch penalty into account. Assume that 20 percent of the instructions are branch instructions and that an optimizing compiler has an 80 percent success rate in filling the single delay slot. For the second alternative, the compiler is able to fill the second slot 25 percent of the time.

8.16 Consider a processor that uses the branch prediction mechanism represented in Figure 8.15*b*. The initial state is either LT or LNT, depending on information provided in the branch instruction. Discuss how the compiler should handle the branch instructions used to control "do while" and "do until" loops, and discuss the suitability of the branch prediction mechanism in each case.

8.17 Assume that the instruction queue in Figure 8.10 can hold up to six instructions. Redraw Figure 8.11 assuming that the queue is full in clock cycle 1 and that the fetch unit can read up to two instructions at a time from the cache. When will the queue become full again after instruction I_k is fetched?

8.18 Redraw Figure 8.11 for the case of the mispredicted branch in Figure 8.14.

8.19 Figure 8.16 shows that one instruction that uses a complex addressing mode takes the same time to execute as an equivalent sequence of instructions that use simpler addressing modes. Yet, the use of simple addressing modes is one of the tenets of the RISC philosophy. How would you design a pipeline to handle complex addressing modes? Discuss the pros and cons of this approach.

REFERENCE

1. The SPARC Architecture Manual, Version 9, D. Weaver and T. Germond, ed., PTR Prentice Hall, Englewood Cliffs, New Jersey, 1994.