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

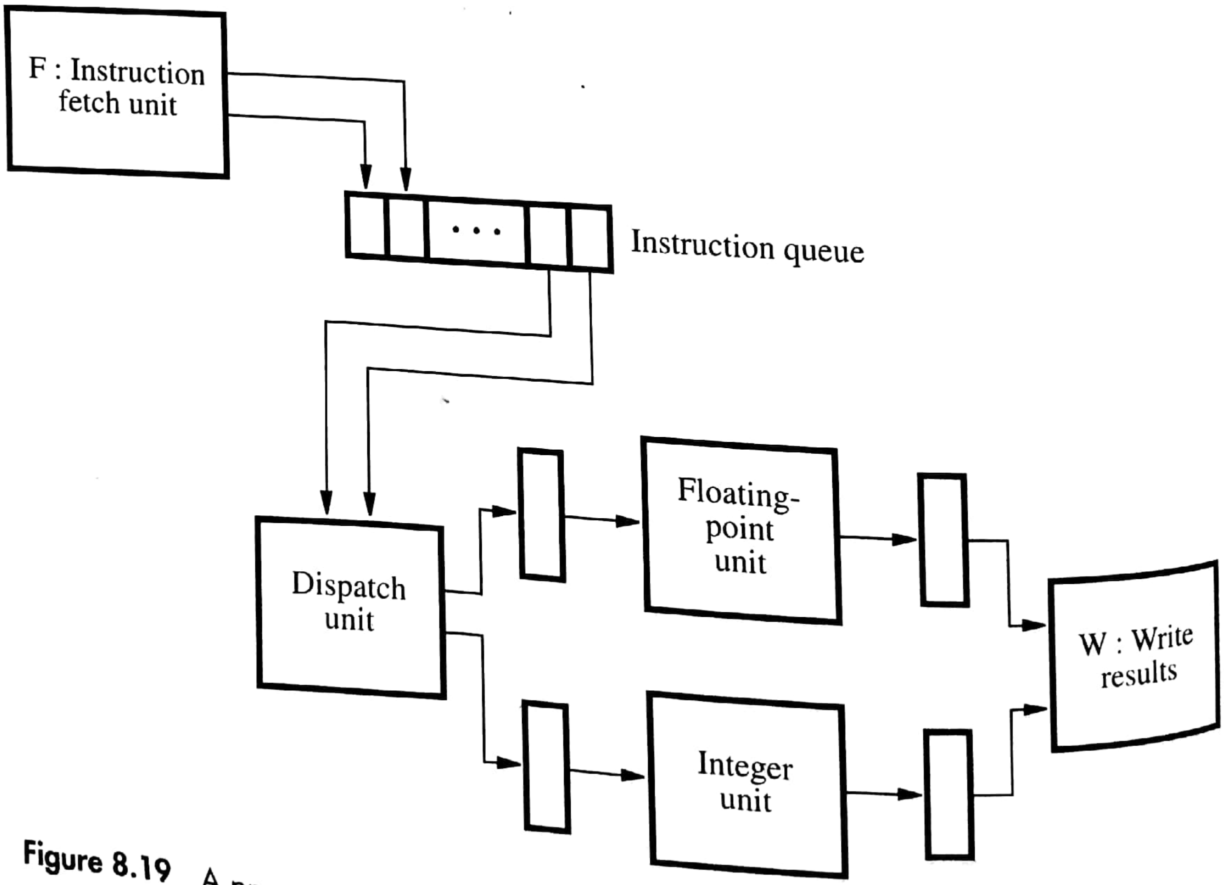


Figure 8.19 A processor with two execution units.

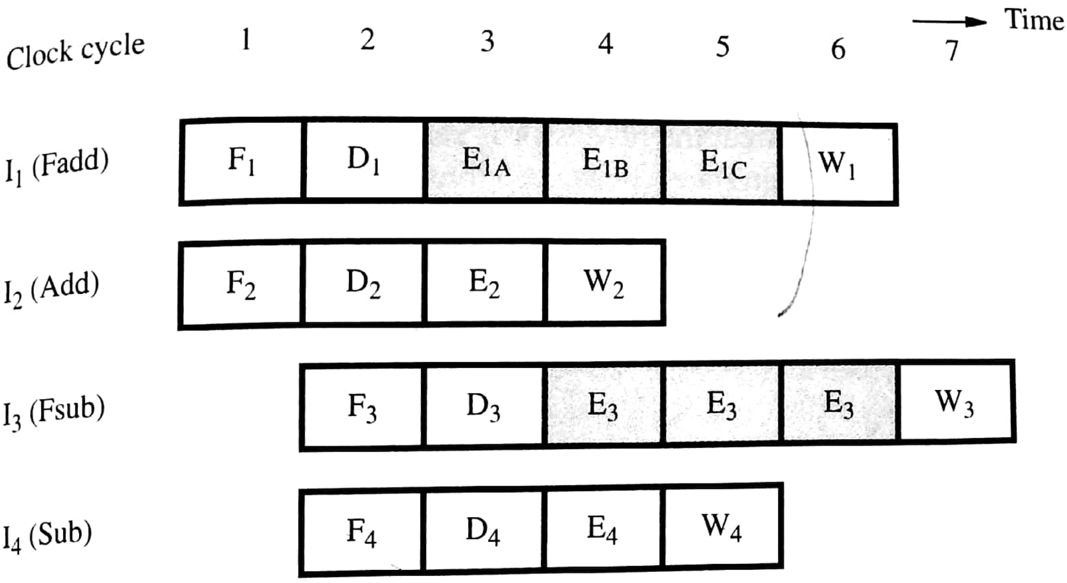


Figure 8.20 An example of instruction execution flow in the processor of Figure 8.19, assuming no hazards are encountered.

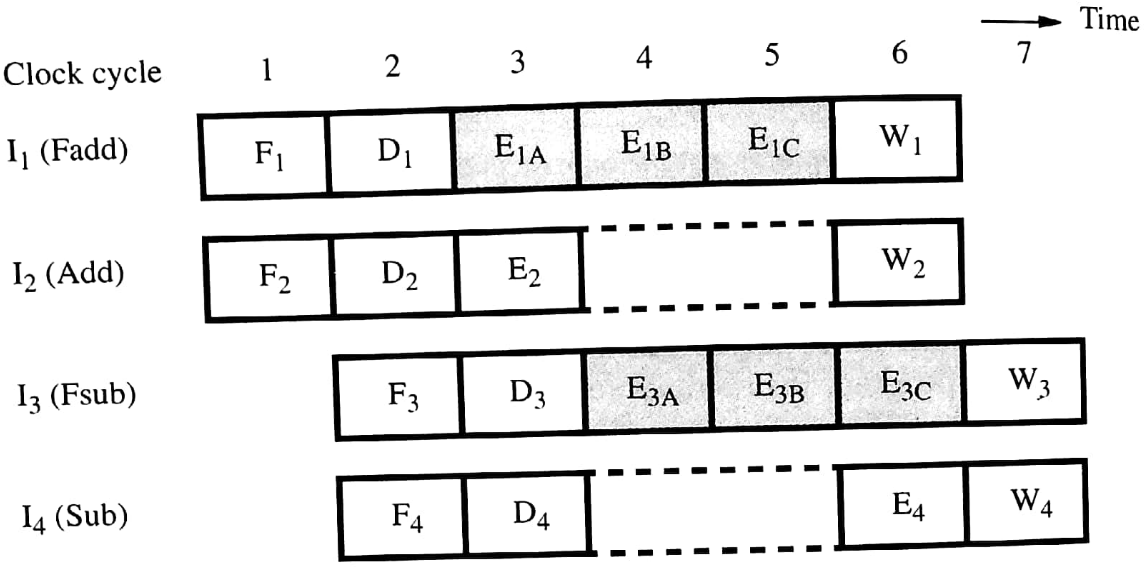
units busy most of the time. In general, high performance is achieved if the compiler is able to arrange program instructions to take maximum advantage of the available hardware units.

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.

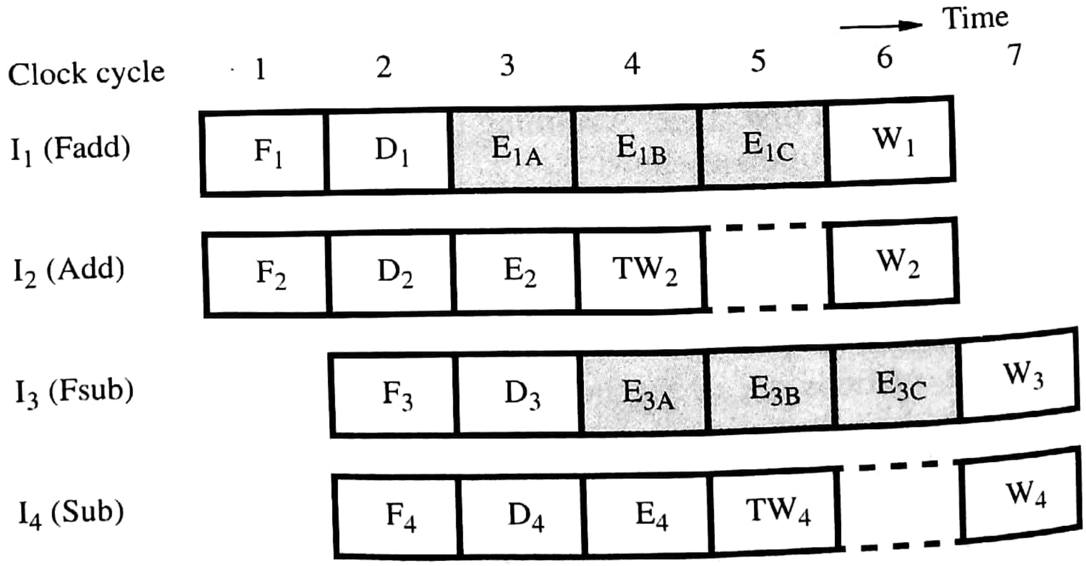
8.6.1 OUT-OF-ORDER EXECUTION

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.

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 nonpipelined 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.