3 Pipelining

It is quite a three-pipe problem.

Sir Arthur Conan Doyle
The Adventures of Sherlock Holmes

3.1 What Is Pipelining?

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution. Today, pipelining is the key implementation technique used to make fast CPUs.

A pipeline is like an assembly line. In an automobile assembly line, there are many steps, each contributing something to the construction of the car. Each step operates in parallel with the other steps, though on a different car. In a computer pipeline, each step in the pipeline completes a part of an instruction. Like the assembly line, different steps are completing different parts of different instructions in parallel. Each of these steps is called a *pipe stage* or a *pipe segment*. The stages are connected one to the next to form a pipe—instructions enter at one end, progress through the stages, and exit at the other end, just as cars would in an assembly line.

In an automobile assembly line, *throughput* is defined as the number of cars per hour and is determined by how often a completed car exits the assembly line. Likewise, the throughput of an instruction pipeline is determined by how often an instruction exits the pipeline. Because the pipe stages are hooked together, all the

stages must be ready to proceed at the same time, just as we would require in an assembly line. The time required between moving an instruction one step down the pipeline is a *machine cycle*. Because all stages proceed at the same time, the length of a machine cycle is determined by the time required for the slowest pipe stage, just as in an auto assembly line, the longest step would determine the time between advancing the line. In a computer, this machine cycle is usually one clock cycle (sometimes it is two, rarely more), although the clock may have multiple phases.

The pipeline designer's goal is to balance the length of each pipeline stage, just as the designer of the assembly line tries to balance the time for each step in the process. If the stages are perfectly balanced, then the time per instruction on the pipelined machine—assuming ideal conditions—is equal to

$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

Under these conditions, the speedup from pipelining equals the number of pipe stages, just as an assembly line with n stages can ideally produce cars n times as fast. Usually, however, the stages will not be perfectly balanced; furthermore, pipelining does involve some overhead. Thus, the time per instruction on the pipelined machine will not have its minimum possible value, yet it can be close.

Pipelining yields a reduction in the average execution time per instruction. Depending on what you consider as the base line, the reduction can be viewed as decreasing the number of clock cycles per instruction (CPI), as decreasing the clock cycle time, or as a combination. If the starting point is a machine that takes multiple clock cycles per instruction, then pipelining is usually viewed as reducing the CPI. This is the primary view we will take. If the starting point is a machine that takes one (long) clock cycle per instruction, then pipelining decreases the clock cycle time.

Pipelining is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream. It has the substantial advantage that, unlike some speedup techniques (see Chapter 8 and Appendix B), it is not visible to the programmer. In this chapter we will first cover the concept of pipelining using DLX and a simple version of its pipeline. We use DLX because its simplicity makes it easy to demonstrate the principles of pipelining. In addition, to simplify the diagrams we do not include the jump instructions of DLX; adding them does not involve new concepts—only bigger diagrams. The principles of pipelining in this chapter apply to more complex instruction sets than DLX or its RISC relatives, although the resulting pipelines are more complex. Using the DLX example, we will look at the problems pipelining introduces and the performance attainable under typical situations. Section 3.9 examines the MIPS R4000 pipeline, which is similar to other recent machines with extensive pipelining. Chapter 4 looks at more advanced pipelining techniques being used in the highest-performance processors.

Before we proceed to basic pipelining, we need to review a simple implementation of an unpipelined version of DLX.

A Simple Implementation of DLX

To understand how DLX can be pipelined, we need to understand how it is implemented *without* pipelining. This section shows a simple implementation where every instruction takes at most five clock cycles. We will extend this basic implementation to a pipelined version, resulting in a much lower CPI. Our unpipelined implementation is not the most economical or the highest-performance implementation without pipelining. Instead, it is designed to lead naturally to a pipelined implementation. We will indicate where the implementation could be improved later in this section. Implementing the instruction set requires the introduction of several temporary registers that are not part of the architecture; these are introduced in this section to simplify pipelining.

In sections 3.1–3.5 we focus on a pipeline for an integer subset of DLX that consists of load-store word, branch, and integer ALU operations. Later in the chapter, we will incorporate the basic floating-point operations. Although we discuss only a subset of DLX, the basic principles can be extended to handle all the instructions.

Every DLX instruction can be implemented in at most five clock cycles. The five clock cycles are as follows.

1. *Instruction fetch cycle* (IF):

   $$\begin{aligned} &\texttt{IR} \leftarrow \texttt{Mem[PC]} \\ &\texttt{NPC} \leftarrow \texttt{PC + 4} \end{aligned}$$

   Operation: Send out the PC and fetch the instruction from memory into the instruction register (IR); increment the PC by 4 to address the next sequential instruction. The IR is used to hold the instruction that will be needed on subsequent clock cycles; likewise the register NPC is used to hold the next sequential PC.

2. *Instruction decode/register fetch cycle* (ID):

   $$\begin{aligned} &\texttt{A} \leftarrow \texttt{Regs}[\texttt{IR}_{6..10}]; \\ &\texttt{B} \leftarrow \texttt{Regs}[\texttt{IR}_{11..15}]; \\ &\texttt{Imm} \leftarrow ((\texttt{IR}_{16})^{16} \#\# \texttt{IR}_{16..31}) \end{aligned}$$

   Operation: Decode the instruction and access the register file to read the registers. The outputs of the general-purpose registers are read into two temporary registers (A and B) for use in later clock cycles.The lower 16 bits of the IR are also sign-extended and stored into the temporary register Imm, for use in the next cycle.

Decoding is done in parallel with reading registers, which is possible because these fields are at a fixed location in the DLX instruction format (see Figure 2.21 on page 99). This technique is known as *fixed-field decoding*. Note that we may read a register we don't use, which doesn't help but also doesn't hurt. Because the immediate portion of an instruction is located in an identical place in every DLX format, the sign-extended immediate is also calculated during this cycle in case it is needed in the next cycle.

3. *Execution/effective address cycle* (EX):

The ALU operates on the operands prepared in the prior cycle, performing one of four functions depending on the DLX instruction type.

- Memory reference:

```
ALUOutput ← A + Imm;
```

Operation: The ALU adds the operands to form the effective address and places the result into the register ALUOutput.

- Register-Register ALU instruction:

```
ALUOutput ← A func B;
```

Operation: The ALU performs the operation specified by the function code on the value in register A and on the value in register B. The result is placed in the temporary register ALUOutput.

- Register-Immediate ALU instruction:

```
ALUOutput ← A op Imm;
```

Operation: The ALU performs the operation specified by the opcode on the value in register A and on the value in register Imm. The result is placed in the temporary register ALUOutput.

- Branch:

```
ALUOutput ← NPC + Imm;
Cond ←(A op 0)
```

Operation: The ALU adds the NPC to the sign-extended immediate value in Imm to compute the address of the branch target. Register A, which has been read in the prior cycle, is checked to determine whether the branch is taken. The comparison operation *op* is the relational operator determined by the branch opcode; for example, *op* is "==" for the instruction `BEQZ`.

The load-store architecture of DLX means that effective address and execution cycles can be combined into a single clock cycle, since no instruction needs

to simultaneously calculate a data address, calculate an instruction target address, and perform an operation on the data. The other integer instructions not included above are jumps of various forms, which are similar to branches.

4. *Memory access/branch completion cycle* (MEM):

   The PC is updated for all instructions: `PC ← NPC;`

   - Memory reference:

     ```
     LMD ← Mem[ALUOutput] or
     Mem[ALUOutput] ← B;
     ```

     Operation: Access memory if needed. If instruction is a load, data returns from memory and is placed in the LMD (load memory data) register; if it is a store, then the data from the B register is written into memory. In either case the address used is the one computed during the prior cycle and stored in the register ALUOutput.

   - Branch:

     ```
     if (cond) PC ← ALUOutput
     ```

     Operation: If the instruction branches, the PC is replaced with the branch destination address in the register ALUOutput.

5. *Write-back cycle* (WB):

   - Register-Register ALU instruction:

     $$\texttt{Regs[IR}_{16..20}\texttt{]} \leftarrow \texttt{ALUOutput;}$$

   - Register-Immediate ALU instruction:

     $$\texttt{Regs[IR}_{11..15}\texttt{]} \leftarrow \texttt{ALUOutput;}$$

   - Load instruction:

     $$\texttt{Regs[IR}_{11..15}\texttt{]} \leftarrow \texttt{LMD;}$$

     Operation: Write the result into the register file, whether it comes from the memory system (which is in LMD) or from the ALU (which is in ALUOutput); the register destination field is also in one of two positions depending on the function code.

Figure 3.1 shows how an instruction flows through the datapath. At the end of each clock cycle, every value computed during that clock cycle and required on a later clock cycle (whether for this instruction or the next) is written into a storage

device, which may be memory, a general-purpose register, the PC, or a temporary register (i.e., LMD, Imm, A, B, IR, NPC, ALUOutput, or Cond). The temporary registers hold values between clock cycles for one instruction, while the other storage elements are visible parts of the state and hold values between successive instructions.

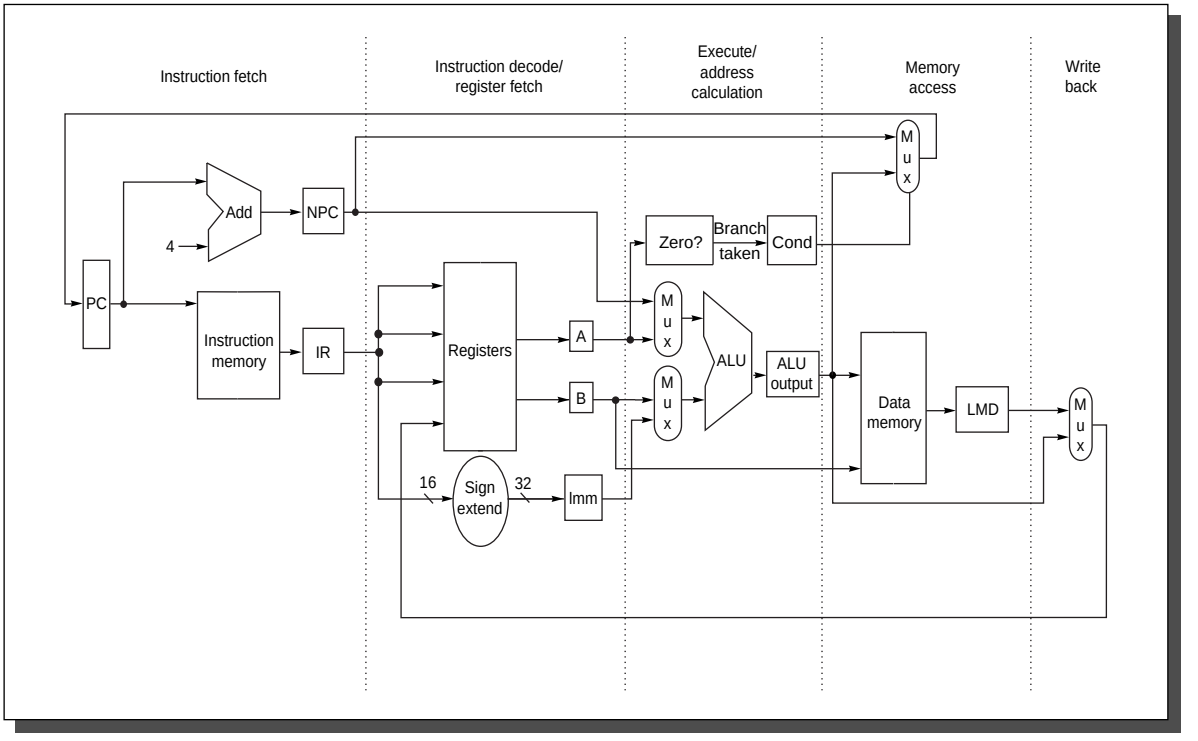

FIGURE 3.1 The implementation of the DLX datapath allows every instruction to be executed in four or five clock cycles. Although the PC is shown in the portion of the datapath that is used in instruction fetch and the registers are shown in the portion of the datapath that is used in instruction decode/register fetch, both of these functional units are read as well as written by an instruction. Although we show these functional units in the cycle corresponding to where they are read, the PC is written during the memory access clock cycle and the registers are written during the write back clock cycle. In both cases, the writes in later pipe stages are indicated by the multiplexer output (in memory access or write back) that carries a value back to the PC or registers. These backward-flowing signals introduce much of the complexity of pipelining, and we will look at them more carefully in the next few sections.

In this implementation, branch and store instructions require four cycles and all other instructions require five cycles. Assuming the branch frequency of 12% and a store frequency of 5% from the last chapter, this leads to an overall CPI of 4.83. This implementation, however, is not optimal either in achieving the best performance or in using the minimal amount of hardware given the performance

level. The CPI could be improved without affecting the clock rate by completing ALU instructions during the MEM cycle, since those instructions are idle during that cycle. Assuming ALU instructions occupy 47% of the instruction mix, as we measured in Chapter 2, this improvement would lead to a CPI of 4.35, or an improvement of 4.82/4.35 = 1.1. Beyond this simple change, any other attempts to decrease the CPI may increase the clock cycle time, since such changes would need to put more activity into a clock cycle. Of course, it may still be beneficial to trade an increase in the clock cycle time for a decrease in the CPI, but this requires a detailed analysis and is unlikely to produce large improvements, especially if the initial distribution of work among the clock cycles is reasonably balanced.

Although all machines today are pipelined, this multicycle implementation is a reasonable approximation of how most machines would have been implemented in earlier times. A simple finite-state machine could be used to implement the control following the five-cycle structure shown above. For a much more complex machine, microcode control could be used. In either event, an instruction sequence like that above would determine the structure of the control.

In addition to these CPI improvements, there are some hardware redundancies that could be eliminated in this multicycle implementation. For example, there are two ALUs: one to increment the PC and one used for effective address and ALU computation. Since they are not needed on the same clock cycle, we could merge them by adding additional multiplexers and sharing the same ALU. Likewise, instructions and data could be stored in the same memory, since the data and instruction accesses happen on different clock cycles.

Rather than optimize this simple implementation, we will leave the design as it is in Figure 3.1, since this provides us with a better base for the pipelined implementation.

As an alternative to the multicycle design discussed in this section, we could also have implemented the machine so that every instruction takes one long clock cycle. In such cases, the temporary registers would be deleted, since there would not be any communication across clock cycles within an instruction. Every instruction would execute in one long clock cycle, writing the result into the data memory, registers, or PC at the end of the clock cycle. The CPI would be one for such a machine. However, the clock cycle would be roughly equal to five times the clock cycle of the multicycle machine, since every instruction would need to traverse all the functional units. Designers would never use this single-cycle implementation for two reasons. First, a single-cycle implementation would be very inefficient for most machines that have a reasonable variation among the amount of work, and hence in the clock cycle time, needed for different instructions. Second, a single-cycle implementation requires the duplication of functional units that could be shared in a multicycle implementation. Nonetheless, this single-cycle datapath allows us to illustrate how pipelining can improve the clock cycle time, as opposed to the CPI, of a machine.

3.2 The Basic Pipeline for DLX

We can pipeline the datapath of Figure 3.1 with almost no changes by starting a new instruction on each clock cycle. (See why we chose that design!) Each of the clock cycles from the previous section becomes a *pipe stage:* a cycle in the pipeline. This results in the execution pattern shown in Figure 3.2, which is the typical way a pipeline structure is drawn. While each instruction takes five clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions.

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

FIGURE 3.2 Simple DLX pipeline. On each clock cycle, another instruction is fetched and begins its five-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a machine that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the implementation on pages 127–129: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write back.

Your instinct is right if you find it hard to believe that pipelining is as simple as this, because it's not. In this and the following sections, we will make our DLX pipeline "real" by dealing with problems that pipelining introduces.

To begin with, we have to determine what happens on every clock cycle of the machine and make sure we don't try to perform two different operations with the same datapath resource on the same clock cycle. For example, a single ALU cannot be asked to compute an effective address and perform a subtract operation at the same time. Thus, we must ensure that the overlap of instructions in the pipeline cannot cause such a conflict. Fortunately, the simplicity of the DLX instruction set makes resource evaluation relatively easy. Figure 3.3 shows a simplified version of the DLX datapath drawn in pipeline fashion. As you can see, the major functional units are used in different cycles and hence overlapping the execution of multiple instructions introduces relatively few conflicts. There are three observations on which this fact rests.

First, the basic datapath of the last section already used separate instruction and data memories, which we would typically implement with separate instruction and data caches (discussed in Chapter 5). The use of separate caches eliminates a conflict for a single memory that would arise between instruction fetch

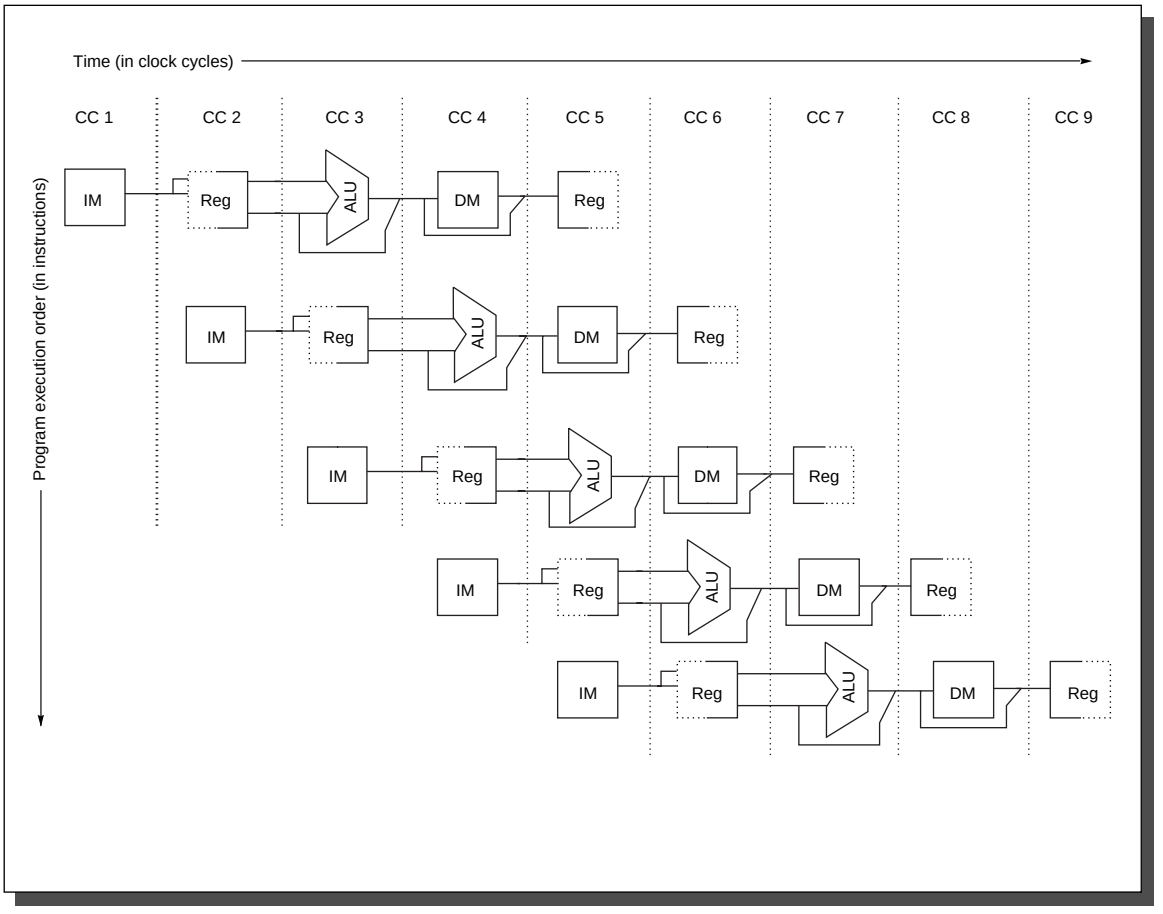

FIGURE 3.3 The pipeline can be thought of as a series of datapaths shifted in time. This shows the overlap among the parts of the datapath, with clock cycle 5 (CC 5) showing the steady state situation. Because the register file is used as a source in the ID stage and as a destination in the WB stage, it appears twice. We show that it is read in one stage and written in another by using a solid line, on the right or left, respectively, and a dashed line on the other side. The abbreviation IM is used for instruction memory, DM for data memory, and CC for clock cycle.

and data memory access. Notice that if our pipelined machine has a clock cycle that is equal to that of the unpipelined version, the memory system must deliver five times the bandwidth. This is one cost of higher performance.

Second, the register file is used in the two stages: for reading in ID and for writing in WB. These uses are distinct, so we simply show the register file in two places. This does mean that we need to perform two reads and one write every clock cycle. What if a read and write are to the same register? For now, we ignore this problem, but we will focus on it in the next section.

Third, Figure 3.3 does not deal with the PC. To start a new instruction every clock, we must increment and store the PC every clock, and this must be done during the IF stage in preparation for the next instruction. The problem arises

when we consider the effect of branches, which changes the PC also, but not until the MEM stage. This is not a problem in our multicycle, unpipelined datapath, since the PC is written once in the MEM stage. For now, we will organize our pipelined datapath to write the PC in IF and write either the incremented PC or the value of the branch target of an earlier branch. This introduces a problem in how branches are handled that we will explain in the next section and explore in detail in section 3.5.

Because every pipe stage is active on every clock cycle, all operations in a pipe stage must complete in one clock cycle and any combination of operations must be able to occur at once. Furthermore, pipelining the datapath requires that values passed from one pipe stage to the next must be placed in registers. Figure 3.4 shows the DLX pipeline with the appropriate registers, called *pipeline registers* or *pipeline latches*, between each pipeline stage. The registers are labeled with the names of the stages they connect. Figure 3.4 is drawn so that connections through the pipeline registers from one stage to another are clear.

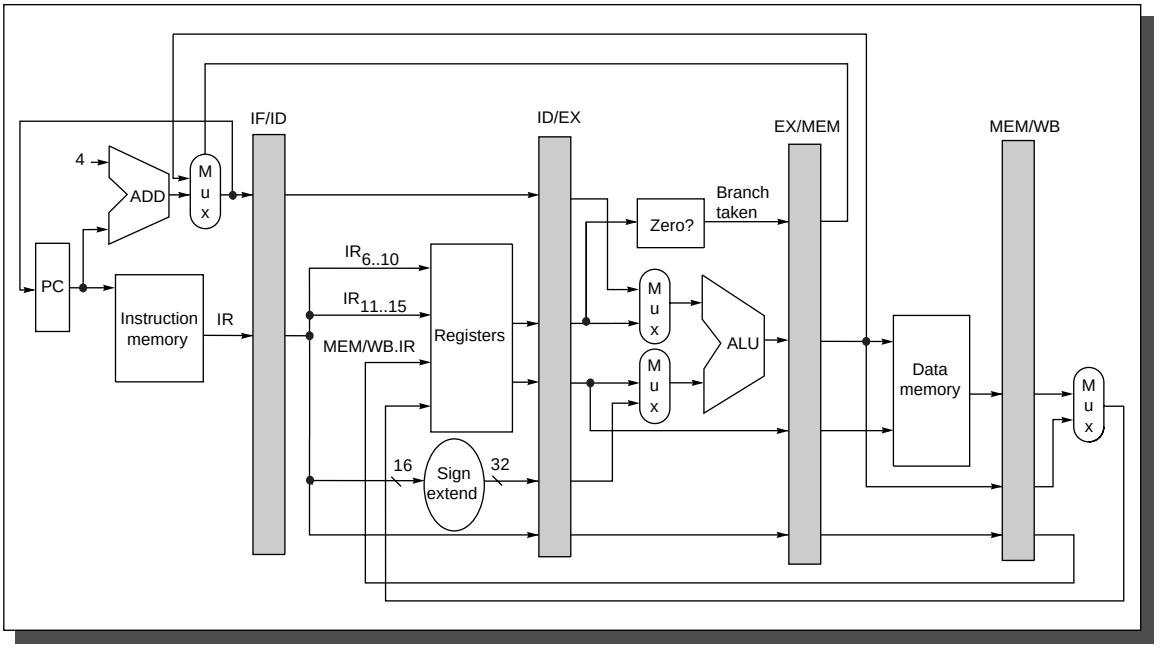

FIGURE 3.4 The datapath is pipelined by adding a set of registers, one between each pair of pipe stages. The registers serve to convey values and control information from one stage to the next. We can also think of the PC as a pipeline register, which sits before the IF stage of the pipeline, leading to one pipeline register for each pipe stage. Recall that the PC is an edge-triggered register written at the end of the clock cycle; hence there is no race condition in writing the PC. The selection multiplexer for the PC has been moved so that the PC is written in exactly one stage (IF). If we didn't move it, there would be a conflict when a branch occurred, since two instructions would try to write different values into the PC. Most of the datapaths flow from left to right, which is from earlier in time to later. The paths flowing from right to left (which carry the register write-back information and PC information on a branch) introduce complications into our pipeline, which we will spend much of this chapter overcoming.

All of the registers needed to hold values temporarily between clock cycles within one instruction are subsumed into these pipeline registers. The fields of the instruction register (IR), which is part of the IF/ID register, are labeled when they are used to supply register names. The pipeline registers carry both data and control from one pipeline stage to the next. Any value needed on a later pipeline stage must be placed in such a register and copied from one pipeline register to the next, until it is no longer needed. If we tried to just use the temporary registers we had in our earlier unpipelined datapath, values could be overwritten before all uses were completed. For example, the field of a register operand used for a write on a load or ALU operation is supplied from the MEM/WB pipeline register rather than from the IF/ID register. This is because we want a load or ALU operation to write the register designated by that operation, not the register field of the instruction currently transitioning from IF to ID! This destination register field is simply copied from one pipeline register to the next, until it is needed during the WB stage.

Any instruction is active in exactly one stage of the pipeline at a time; therefore, any actions taken on behalf of an instruction occur between a pair of pipeline registers. Thus, we can also look at the activities of the pipeline by examining what has to happen on any pipeline stage depending on the instruction type. Figure 3.5 shows this view. Fields of the pipeline registers are named so as to show the flow of data from one stage to the next. Notice that the actions in the first two stages are independent of the current instruction type; they must be independent because the instruction is not decoded until the end of the ID stage. The IF activity depends on whether the instruction in EX/MEM is a taken branch. If so, then the branch target address of the branch instruction in EX/MEM is written into the PC at the end of IF; otherwise the incremented PC will be written back. (As we said earlier, this effect of branches leads to complications in the pipeline that we deal with in the next few sections.) The fixed-position encoding of the register source operands is critical to allowing the registers to be fetched during ID.

To control this simple pipeline we need only determine how to set the control for the four multiplexers in the datapath of Figure 3.4. The two multiplexers in the ALU stage are set depending on the instruction type, which is dictated by the IR field of the ID/EX register. The top ALU input multiplexer is set by whether the instruction is a branch or not, and the bottom multiplexer is set by whether the instruction is a register-register ALU operation or any other type of operation. The multiplexer in the IF stage chooses whether to use the value of the incremented PC or the value of the EX/MEM.ALUOutput (the branch target) to write into the PC. This multiplexer is controlled by the field EX/MEM.cond. The fourth multiplexer is controlled by whether the instruction in the WB stage is a load or a ALU operation. In addition to these four multiplexers, there is one additional multiplexer needed that is not drawn in Figure 3.4, but whose existence is clear from looking at the WB stage of an ALU operation. The destination register field is in one of two different places depending on the instruction type (register-register ALU versus either ALU immediate or load). Thus, we will need a multiplexer to choose the correct portion of the IR in the MEM/WB register to specify the register destination field, assuming the instruction writes a register.

Stage	Any instruction		
IF	IF/ID.IR ← Mem[PC]; IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) & EX/MEM.cond){EX/MEM. ALUOutput} else {PC+4});		
ID	ID/EX.A ← Regs[IF/ID.IR$_{6..10}$]; ID/EX.B ← Regs[IF/ID.IR$_{11..15}$]; ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$;		
	ALU instruction	**Load or store instruction**	**Branch instruction**
EX	EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput← ID/EX.A *func* ID/EX.B; or EX/MEM.ALUOutput ← ID/EX.A *op* ID/EX.Imm; EX/MEM.cond ← 0;	EX/MEM.IR← ID/EX.IR EX/MEM.ALUOutput ← ID/EX.A + ID/EX.Imm; EX/MEM.cond ← 0; EX/MEM.B← ID/EX.B;	 EX/MEM.ALUOutput ← ID/EX.NPC+ID/EX.Imm; EX/MEM.cond ← (ID/EX.A *op* 0);
MEM	MEM/WB.IR ← EX/MEM.IR; MEM/WB.ALUOutput ← EX/MEM.ALUOutput;	MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput]; or Mem[EX/MEM.ALUOutput] ← EX/MEM.B;	
WB	Regs[MEM/WB.IR$_{16..20}$] ← MEM/WB.ALUOutput; or Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.ALUOutput;	For load only: Regs[MEM/WB.IR$_{11..15}$] ← MEM/WB.LMD;	

FIGURE 3.5 Events on every pipe stage of the DLX pipeline. Let's review the actions in the stages that are specific to the pipeline organization. In IF, in addition to fetching the instruction and computing the new PC, we store the incremented PC both into the PC and into a pipeline register (NPC) for later use in computing the branch target address. This structure is the same as the organization in Figure 3.4, where the PC is updated in IF from one or two sources. In ID, we fetch the registers, extend the sign of the lower 16 bits of the IR, and pass along the IR and NPC. During EX, we perform an ALU operation or an address calculation; we pass along the IR and the B register (if the instruction is a store). We also set the value of cond to 1 if the instruction is a taken branch. During the MEM phase, we cycle the memory, write the PC if needed, and pass along values needed in the final pipe stage. Finally, during WB, we update the register field from either the ALU output or the loaded value. For simplicity we always pass the entire IR from one stage to the next, though as an instruction proceeds down the pipeline, less and less of the IR is needed.

Basic Performance Issues in Pipelining

Pipelining increases the CPU instruction throughput—the number of instructions completed per unit of time—but it does not reduce the execution time of an individual instruction. In fact, it usually slightly increases the execution time of each instruction due to overhead in the control of the pipeline. The increase in instruction throughput means that a program runs faster and has lower total execution time, even though no single instruction runs faster!

The fact that the execution time of each instruction does not decrease puts limits on the practical depth of a pipeline, as we will see in the next section. In addition to limitations arising from pipeline latency, limits arise from imbalance among the pipe stages and from pipelining overhead. Imbalance among the pipe stages reduces performance since the clock can run no faster than the time needed for the slowest pipeline stage. Pipeline overhead arises from the combination of pipeline register delay and clock skew. The pipeline registers add setup time, which is the time that a register input must be stable before the clock signal that triggers a write occurs, plus propagation delay to the clock cycle. Clock skew, which is maximum delay between when the clock arrives at any two registers, also contributes to the lower limit on the clock cycle. Once the clock cycle is as small as the sum of the clock skew and latch overhead, no further pipelining is useful, since there is no time left in the cycle for useful work.

EXAMPLE Consider the unpipelined machine in the previous section. Assume that it has 10-ns clock cycles and that it uses four cycles for ALU operations and branches and five cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the machine adds 1 ns of overhead to the clock. Ignoring any latency impact, how much speedup in the instruction execution rate will we gain from a pipeline?

ANSWER The average instruction execution time on the unpipelined machine is

$$\begin{aligned}\text{Average instruction execution time} &= \text{Clock cycle} \times \text{Average CPI}\\ &= 10 \text{ ns} \times ((40\% + 20\%) \times 4 + 40\% \times 5)\\ &= 10 \text{ ns} \times 4.4\\ &= 44 \text{ ns}\end{aligned}$$

In the pipelined implementation, the clock must run at the speed of the slowest stage plus overhead, which will be 10 + 1 or 11 ns; this is the average instruction execution time. Thus, the speedup from pipelining is

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}\\ &= \frac{44 \text{ ns}}{11 \text{ ns}} = 4 \text{ times}\end{aligned}$$

The 1-ns overhead essentially establishes a limit on the effectiveness of pipelining. If the overhead is not affected by changes in the clock cycle, Amdahl's Law tells us that the overhead limits the speedup. ■

Alternatively, if our base machine already has a CPI of 1 (with a longer clock cycle), then pipelining will enable us to have a shorter clock cycle. The datapath of the previous section can be made into a single-cycle datapath by simply removing the latches and letting the data flow from one cycle of execution to the next. How would the speedup of the pipelined version compare to the single-cycle machine?

EXAMPLE Assume that the times required for the five functional units, which operate in each of the five cycles, are as follows: 10 ns, 8 ns, 10 ns, 10 ns, and 7 ns. Assume that pipelining adds 1 ns of overhead. Find the speedup versus the single-cycle datapath.

ANSWER Since the unpipelined machine executes all instructions in a single clock cycle, its average time per instruction is simply the clock cycle time. The clock cycle time is equal to the sum of the times for each step in the execution:

$$\begin{aligned}\text{Average instruction execution time} &= 10 + 8 + 10 + 10 + 7\\ &= 45 \text{ ns}\end{aligned}$$

The clock cycle time on the pipelined machine must be the largest time for any stage in the pipeline (10 ns) plus the overhead of 1 ns, for a total of 11 ns. Since the CPI is 1, this yields an average instruction execution time of 11 ns. Thus,

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}\\ &= \frac{45 \text{ ns}}{11 \text{ ns}} = 4.1 \text{ times}\end{aligned}$$

Pipelining can be thought of as improving the CPI, which is what we typically do, as increasing the clock rate—especially compared to another pipelined machine, or sometimes as doing both. ■

Because the latches in a pipelined design can have a significant impact on the clock speed, designers have looked for latches that permit the highest possible clock rate. The Earle latch (invented by J. G. Earle [1965]) has three properties that make it especially useful in pipelined machines. First, it is relatively insensitive to clock skew. Second, the delay through the latch is always a constant two-gate delay, avoiding the introduction of skew in the data passing through the latch. Finally, two levels of logic can be performed in the latch without increasing the latch delay time. This means that two levels of logic in the pipeline can be overlapped with the latch, so the overhead from the latch can be hidden. We will not be analyzing the pipeline designs in this chapter at this level of detail. The interested reader should see Kunkel and Smith [1986].

The pipeline we now have for DLX would function just fine for integer instructions if every instruction were independent of every other instruction in the pipeline. In reality, instructions in the pipeline can depend on one another; this is the topic of the next section. The complications that arise in the floating-point pipeline will be treated in section 3.7, and section 3.9 will look at a complete real pipeline.

3.3 The Major Hurdle of Pipelining—Pipeline Hazards

There are situations, called *hazards*, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

1. *Structural hazards* arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
2. *Data hazards* arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
3. *Control hazards* arise from the pipelining of branches and other instructions that change the PC.

Hazards in pipelines can make it necessary to *stall* the pipeline. In Chapter 1, we mentioned that the processor could stall on an event such as a cache miss. Stalls arising from hazards in pipelined machines are more complex than the simple stall for a cache miss. Eliminating a hazard often requires that some instructions in the pipeline be allowed to proceed while others are delayed. For the pipelines we discuss in this chapter, when an instruction is stalled, all instructions issued *later* than the stalled instruction—and hence not as far along in the pipeline—are also stalled. Instructions issued *earlier* than the stalled instruction—and hence farther along in the pipeline—must continue, since otherwise the hazard will never clear. As a result, no new instructions are fetched during the stall. In contrast to this process of stalling only a portion of the pipeline, a cache miss stalls *all* the instructions in the pipeline both before and after the instruction causing the miss. (For the simple pipelines of this chapter there is no advantage in selecting stalling instructions on a cache miss, but in future chapters we will examine pipelines and caches that reduce cache miss costs by selectively stalling on a cache miss.) We will see several examples of how pipeline stalls operate in this section—don't worry, they aren't as complex as they might sound!

Performance of Pipelines with Stalls

A stall causes the pipeline performance to degrade from the ideal performance. Let's look at a simple equation for finding the actual speedup from pipelining, starting with the formula from the previous section.

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}\\ &= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}\\ &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\end{aligned}$$

Remember that pipelining can be thought of as decreasing the CPI or the clock cycle time. Since it is traditional to use the CPI to compare pipelines, let's start with that assumption. The ideal CPI on a pipelined machine is almost always 1. Hence, we can compute the pipelined CPI:

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}\\ &= 1 + \text{Pipeline stall clock cycles per instruction}\end{aligned}$$

If we ignore the cycle time overhead of pipelining and assume the stages are perfectly balanced, then the cycle time of the two machines can be equal, leading to

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

One important simple case is where all instructions take the same number of cycles, which must also equal the number of pipeline stages (also called the *depth of the pipeline*). In this case, the unpipelined CPI is equal to the depth of the pipeline, leading to

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls, this leads to the intuitive result that pipelining can improve performance by the depth of the pipeline.

Alternatively, if we think of pipelining as improving the clock cycle time, then we can assume that the CPI of the unpipelined machine, as well as that of the pipelined machine, is 1. This leads to

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\\ &= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\end{aligned}$$

In cases where the pipe stages are perfectly balanced and there is no overhead, the clock cycle on the pipelined machine is smaller than the clock cycle of the unpipelined machine by a factor equal to the pipelined depth:

$$\text{Clock cycle pipelined} = \frac{\text{Clock cycle unpipelined}}{\text{Pipeline depth}}$$

$$\text{Pipeline depth} = \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

This leads to the following:

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}} \\ &= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}\end{aligned}$$

Thus, if there are no stalls, the speedup is equal to the number of pipeline stages, matching our intuition for the ideal case.

Structural Hazards

When a machine is pipelined, the overlapped execution of instructions requires pipelining of functional units and duplication of resources to allow all possible combinations of instructions in the pipeline. If some combination of instructions cannot be accommodated because of resource conflicts, the machine is said to have a *structural hazard*. The most common instances of structural hazards arise when some functional unit is not fully pipelined. Then a sequence of instructions using that unpipelined unit cannot proceed at the rate of one per clock cycle. Another common way that structural hazards appear is when some resource has not been duplicated enough to allow all combinations of instructions in the pipeline to execute. For example, a machine may have only one register-file write port, but under certain circumstances, the pipeline might want to perform two writes in a clock cycle. This will generate a structural hazard. When a sequence of instructions encounters this hazard, the pipeline will stall one of the instructions until the required unit is available. Such stalls will increase the CPI from its usual ideal value of 1.

Some pipelined machines have shared a single-memory pipeline for data and instructions. As a result, when an instruction contains a data-memory reference, it will conflict with the instruction reference for a later instruction, as shown in Figure 3.6. To resolve this, we stall the pipeline for one clock cycle when the data memory access occurs. Figure 3.7 shows our pipeline datapath figure with the stall cycle added. A stall is commonly called a *pipeline bubble* or just *bubble*, since it floats through the pipeline taking space but carrying no useful work. We will see another type of stall when we talk about data hazards.

Rather than draw the pipeline datapath every time, designers often just indicate stall behavior using a simpler diagram with only the pipe stage names, as in Figure 3.8. The form of Figure 3.8 shows the stall by indicating the cycle when no action occurs and simply shifting instruction 3 to the right (which delays its

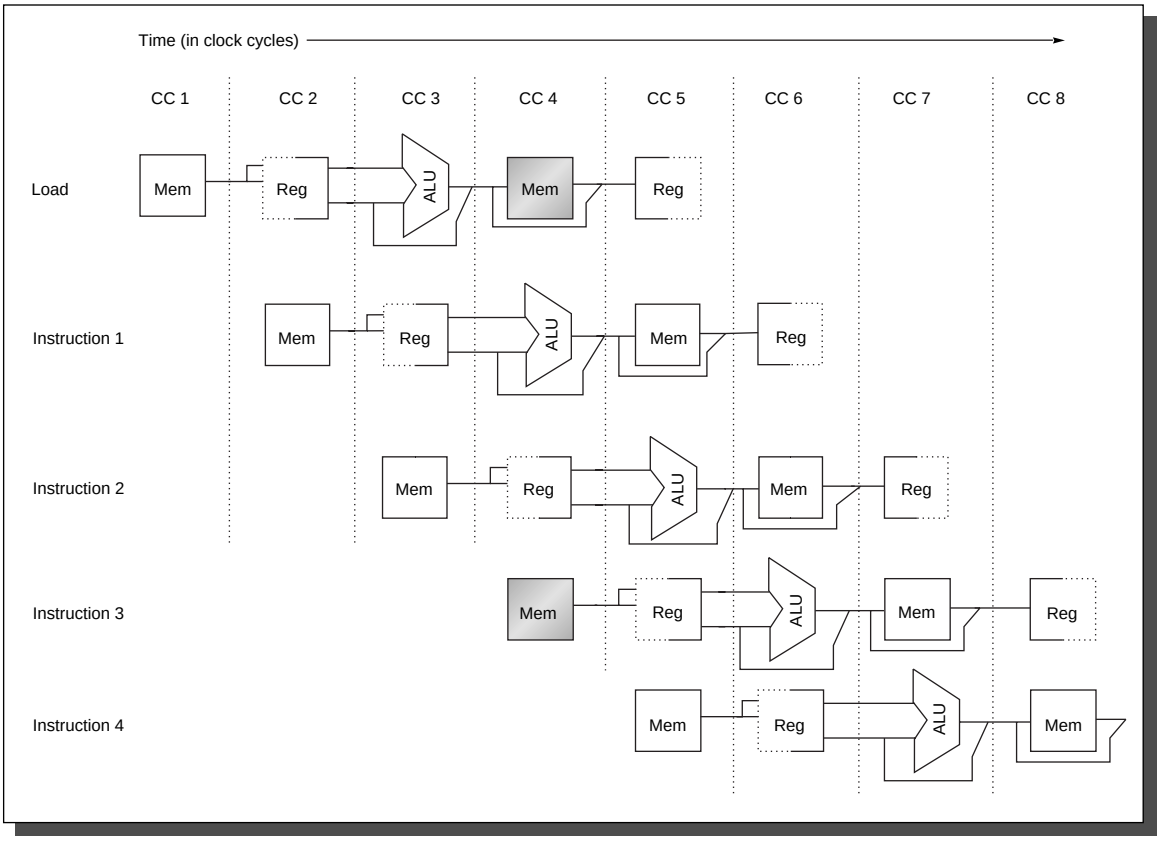

FIGURE 3.6 A machine with only one memory port will generate a conflict whenever a memory reference occurs. In this example the load instruction uses the memory for a data access at the same time instruction 3 wants to fetch an instruction from memory.

execution start and finish by one cycle). The effect of the pipeline bubble is actually to occupy the resources for that instruction slot as it travels through the pipeline, just as Figure 3.7 shows. Although Figure 3.7 shows how the stall is actually implemented, the performance impact indicated by the two figures is the same: Instruction 3 does not complete until clock cycle 9, and no instruction completes during clock cycle 8.

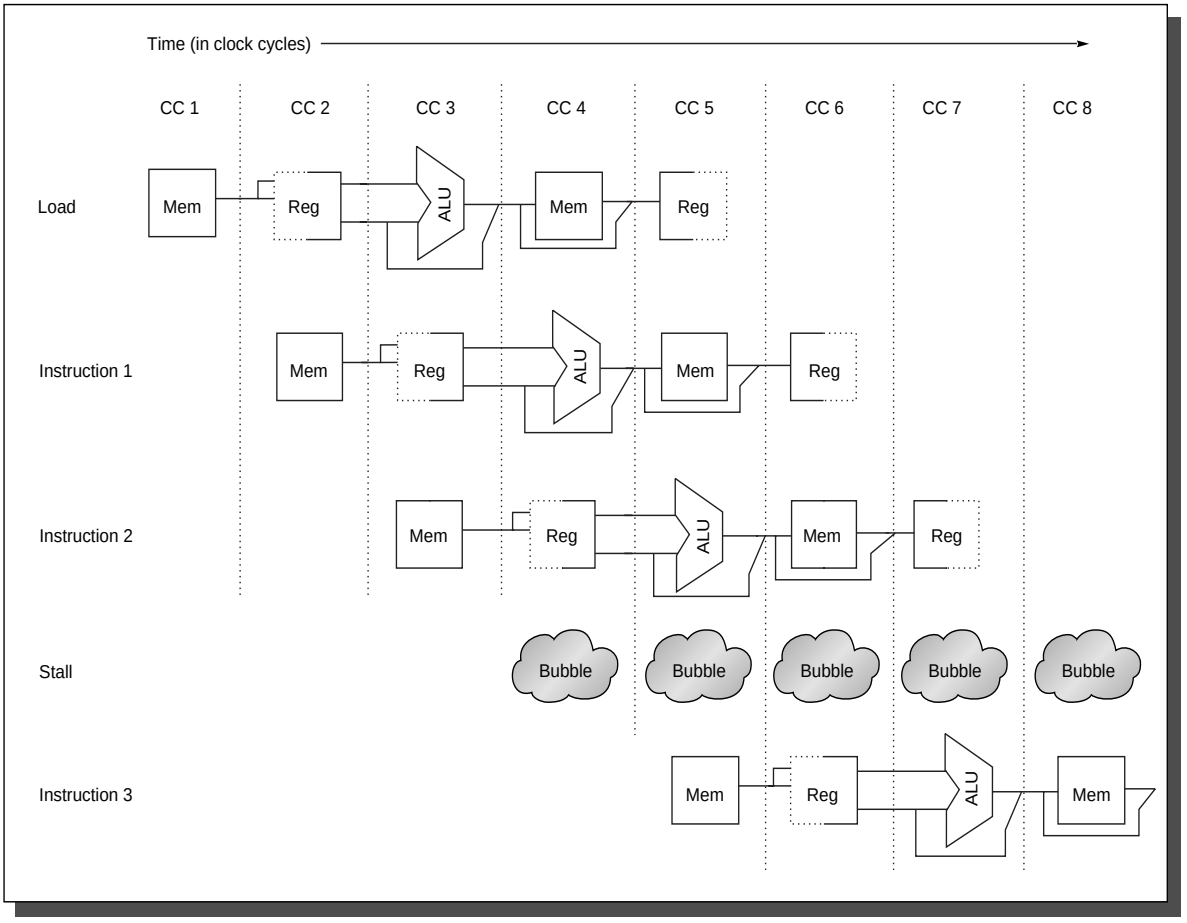

FIGURE 3.7 The structural hazard causes pipeline bubbles to be inserted. The effect is that no instruction will finish during clock cycle 8, when instruction 3 would normally have finished. Instruction 1 is assumed to not be a load or store; otherwise, instruction 3 cannot start execution.

	Clock cycle number									
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Load instruction	IF	ID	EX	MEM	WB					
Instruction *i* + 1		IF	ID	EX	MEM	WB				
Instruction *i* + 2			IF	ID	EX	MEM	WB			
Instruction *i* + 3				stall	IF	ID	EX	MEM	WB	
Instruction *i* + 4						IF	ID	EX	MEM	WB
Instruction *i* + 5							IF	ID	EX	MEM
Instruction *i* + 6								IF	ID	EX

FIGURE 3.8 A pipeline stalled for a structural hazard—a load with one memory port. As shown here, the load instruction effectively steals an instruction-fetch cycle, causing the pipeline to stall—no instruction is initiated on clock cycle 4 (which normally would initiate instruction *i* + 3). Because the instruction being fetched is stalled, all other instructions in the pipeline before the stalled instruction can proceed normally. The stall cycle will continue to pass through the pipeline, so that no instruction completes on clock cycle 8. Sometimes these pipeline diagrams are drawn with the stall occupying an entire horizontal row and instruction 3 being moved to the next row; in either case, the effect is the same, since instruction 3 does not begin execution until cycle 5. We use the form above, since it takes less space.

EXAMPLE Let's see how much the load structural hazard might cost. Suppose that data references constitute 40% of the mix, and that the ideal CPI of the pipelined machine, ignoring the structural hazard, is 1. Assume that the machine with the structural hazard has a clock rate that is 1.05 times higher than the clock rate of the machine without the hazard. Disregarding any other performance losses, is the pipeline with or without the structural hazard faster, and by how much?

ANSWER There are several ways we could solve this problem. Perhaps the simplest is to compute the average instruction time on the two machines:

$$\text{Average instruction time} = \text{CPI} \times \text{Clock cycle time}$$

Since it has no stalls, the average instruction time for the ideal machine is simply the Clock cycle time$_{\text{ideal}}$. The average instruction time for the machine with the structural hazard is

$$\begin{aligned}\text{Average instruction time} &= \text{CPI} \times \text{Clock cycle time} \\ &= (1 + 0.4 \times 1) \times \frac{\text{Clock cycle time}_{\text{ideal}}}{1.05} \\ &= 1.3 \times \text{Clock cycle time}_{\text{ideal}}\end{aligned}$$

Clearly, the machine without the structural hazard is faster; we can use the ratio of the average instruction times to conclude that the machine without the hazard is 1.3 times faster.

As an alternative to this structural hazard, the designer could provide a separate memory access for instructions, either by splitting the cache into separate instruction and data caches, or by using a set of buffers, usually called *instruction buffers*, to hold instructions. Both the split cache and instruction buffer ideas are discussed in Chapter 5. ■

If all other factors are equal, a machine without structural hazards will always have a lower CPI. Why, then, would a designer allow structural hazards? There are two reasons: to reduce cost and to reduce the latency of the unit. Pipelining all the functional units, or duplicating them, may be too costly. For example, machines that support both an instruction and a data cache access every cycle (to prevent the structural hazard of the above example) require twice as much total memory bandwidth and often have higher bandwidth at the pins. Likewise, fully pipelining a floating-point multiplier consumes lots of gates. If the structural hazard would not occur often, it may not be worth the cost to avoid it. It is also usually possible to design an unpipelined unit, or one that isn't fully pipelined, with a somewhat shorter total delay than a fully pipelined unit. The shorter latency comes from the lack of pipeline registers that introduce overhead. For example, both the CDC 7600 and the MIPS R2010 floating-point unit choose shorter latency (fewer clocks per operation) versus full pipelining. As we will see shortly, reducing latency has other performance benefits and may overcome the disadvantage of the structural hazard.

EXAMPLE Many recent machines do not have fully pipelined floating-point units. For example, suppose we had an implementation of DLX with a floating-point multiply unit but no pipelining. Assume the multiplier could accept a new multiply operation every five clock cycles. (This rate is called the *repeat* or initiation *interval*.) Will this structural hazard have a large or small performance impact on mdljdp2 running on DLX? For simplicity, assume that the floating-point multiplies are uniformly distributed.

ANSWER From Chapter 2 we find that floating-point multiply has a frequency of 14% in mdljdp2. Our proposed pipeline can handle up to a 20% frequency of floating-point multiplies—one every five clock cycles. This means that the performance benefit of fully pipelining the floating-point multiply on mdljdp2 is likely to be limited, as long as the floating-point multiplies are not clustered but are distributed uniformly. In the best case, multiplies are overlapped with other operations, and there is no performance penalty at all. In the worst case, the multiplies are all clustered with no intervening instructions, and 14% of the instructions take 5 cycles each. Assuming a base CPI of 1, this amounts to an increase of 0.7 in the CPI.

In practice, examining the performance of mdljdp2 on a machine with a five-cycle-deep FP multiply pipeline shows that this structural hazard increases execution time by less than 3%. One reason this loss is so low is that data hazards (the topic of the next section) cause the pipeline to stall, preventing multiply instructions that might cause structural hazards from being initiated. Of course, other benchmarks make heavier use of floating-point multiply or have fewer data hazards, and thus would show a larger impact. In the rest of this chapter we will examine the contributions of these different types of stalls in the DLX pipeline. ■

3.4 Data Hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This introduces data and control hazards. Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined machine. Consider the pipelined execution of these instructions:

```
ADD     R1,R2,R3
SUB     R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
XOR     R10,R1,R11
```

All the instructions after the ADD use the result of the ADD instruction. As shown in Figure 3.9, the ADD instruction writes the value of R1 in the WB pipe stage, but the SUB instruction reads the value during its ID stage. This problem is called a *data hazard*. Unless precautions are taken to prevent it, the SUB instruction will read the wrong value and try to use it. In fact, the value used by the SUB instruction is not even deterministic: Though we might think it logical to assume that SUB would always use the value of R1 that was assigned by an instruction prior to ADD, this is not always the case. If an interrupt should occur between the ADD and SUB instructions, the WB stage of the ADD will complete, and the value of R1 at that point will be the result of the ADD. This unpredictable behavior is obviously unacceptable.

The AND instruction is also affected by this hazard. As we can see from Figure 3.9, the write of R1 does not complete until the end of clock cycle 5. Thus, the AND instruction that reads the registers during clock cycle 4 will receive the wrong results.

The XOR instruction operates properly, because its register read occurs in clock cycle 6, after the register write. The OR instruction can also be made to operate without incurring a hazard by a simple implementation technique, implied in our pipeline diagrams. The technique is to perform the register file reads in the second half of the cycle and the writes in the first half. This technique,

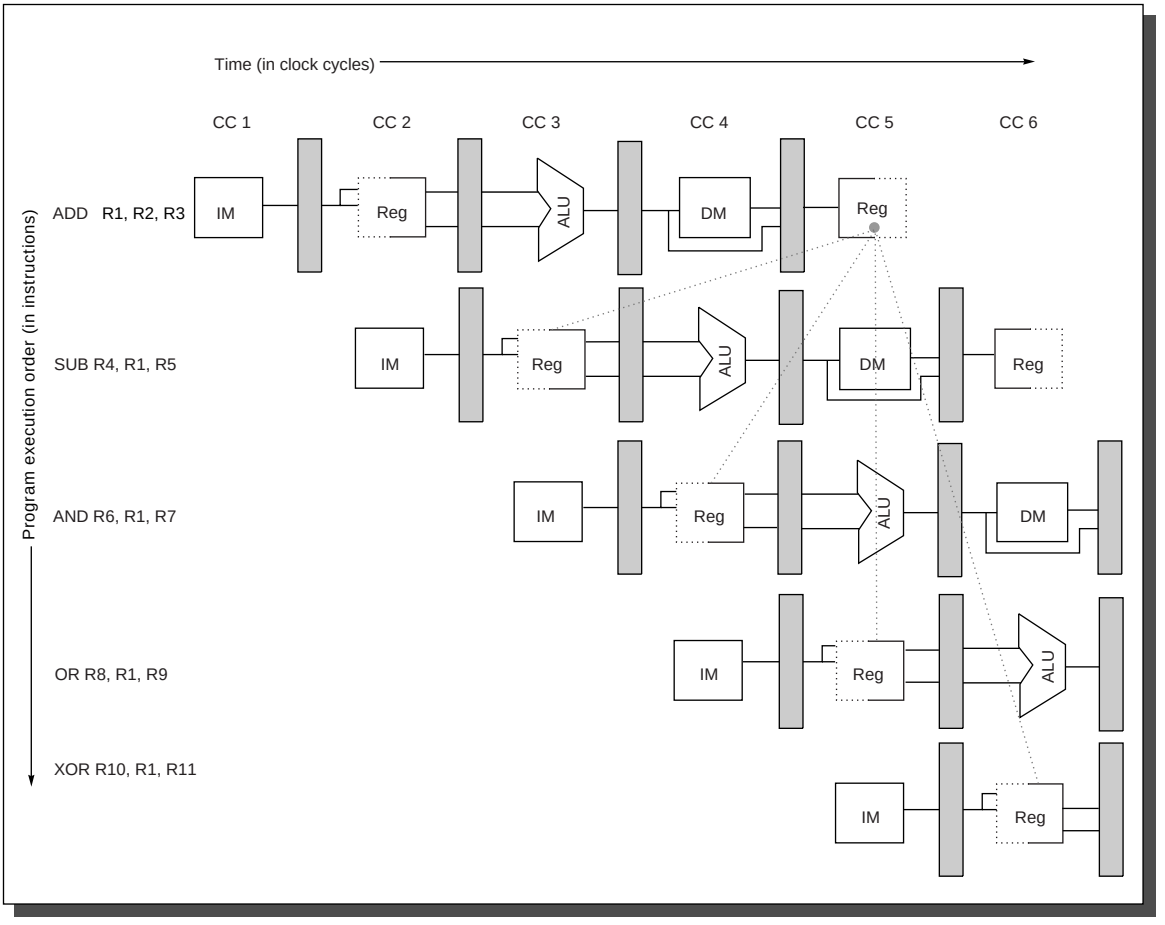

FIGURE 3.9 The use of the result of the ADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

which is hinted at in earlier figures by placing the dashed box around the register file, allows the OR instruction in the example in Figure 3.9 to execute correctly.

The next subsection discusses a technique to eliminate the stalls for the hazard involving the SUB and AND instructions.

Minimizing Data Hazard Stalls By Forwarding

The problem posed in Figure 3.9 can be solved with a simple hardware technique called *forwarding* (also called *bypassing* and sometimes *short-circuiting*). The key insight in forwarding is that the result is not really needed by the SUB until after the ADD actually produces it. If the result can be moved from where the ADD

produces it, the EX/MEM register, to where the SUB needs it, the ALU input latches, then the need for a stall can be avoided. Using this observation, forwarding works as follows:

1. The ALU result from the EX/MEM register is always fed back to the ALU input latches.
2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.

Notice that with forwarding, if the SUB is stalled, the ADD will be completed and the bypass will not be activated. This is also true for the case of an interrupt between the two instructions.

As the example in Figure 3.9 shows, we need to forward results not only from the immediately previous instruction, but possibly from an instruction that started two cycles earlier. Figure 3.10 shows our example with the bypass paths in place and highlighting the timing of the register read and writes. This code sequence can be executed without stalls.

Forwarding can be generalized to include passing a result directly to the functional unit that requires it: A result is forwarded from the output of one unit to the input of another, rather than just from the result of a unit to the input of the same unit. Take, for example, the following sequence:

```
ADD     R1,R2,R3
LW      R4,0(R1)
SW      12(R1),R4
```

To prevent a stall in this sequence, we would need to forward the values of R1 and R4 from the pipeline registers to the ALU and data memory inputs. Figure 3.11 shows all the forwarding paths for this example. In DLX, we may require a forwarding path from any pipeline register to the input of any functional unit. Because the ALU and data memory both accept operands, forwarding paths are needed to their inputs from both the ALU/MEM and MEM/WB registers. In addition, DLX uses a zero detection unit that operates during the EX cycle, and forwarding to that unit will be needed as well. Later in this section we will explore all the necessary forwarding paths and the control of those paths.

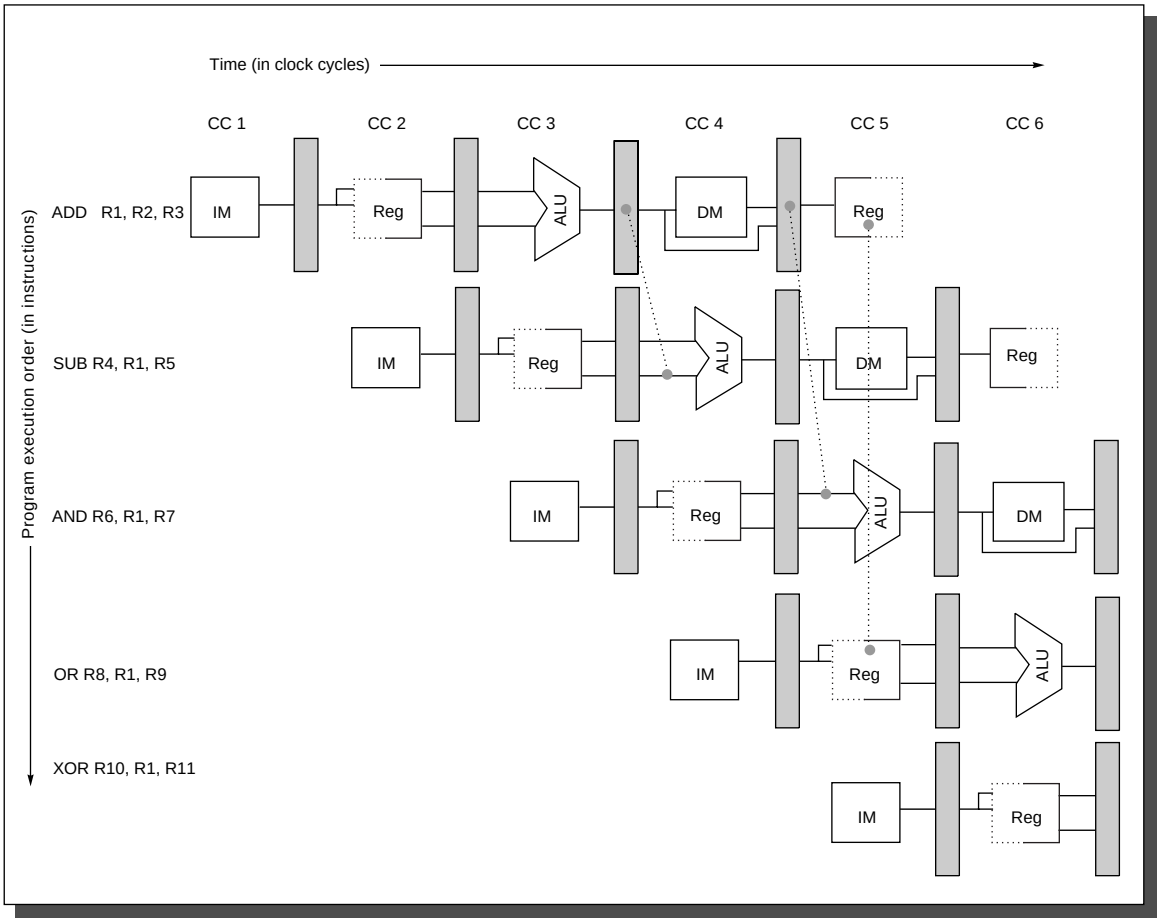

FIGURE 3.10 A set of instructions that depend on the ADD result use forwarding paths to avoid the data hazard. The inputs for the SUB and AND instructions forward from the EX/MEM and the MEM/WB pipeline registers, respectively, to the first ALU input. The OR receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and writing in the first half, as the dashed lines on the registers indicate. Notice that the forwarded result can go to either ALU input; in fact, both ALU inputs could use forwarded inputs from either the same pipeline register or from different pipeline registers. This would occur, for example, if the AND instruction was AND R6, R1, R4.

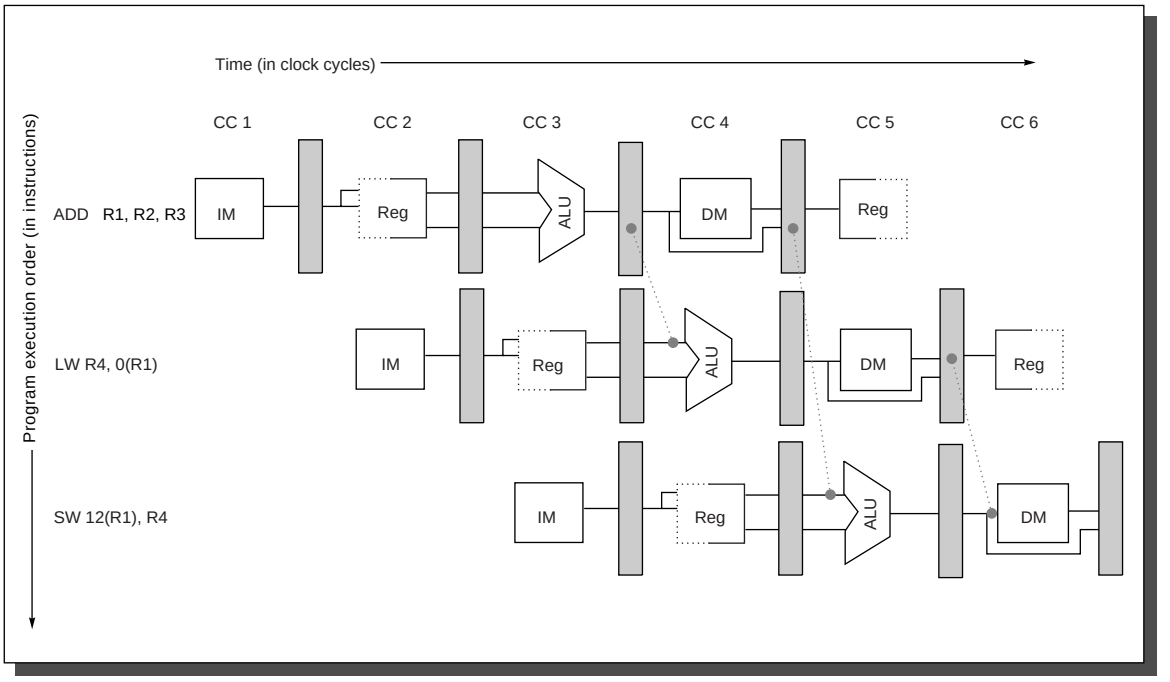

FIGURE 3.11 Stores require an operand during MEM, and forwarding of that operand is shown here. The result of the load is forwarded from the memory output in MEM/WB to the memory input to be stored. In addition, the ALU output is forwarded to the ALU input for the address calculation of both the load and the store (this is no different than forwarding to another ALU operation). If the store depended on an immediately preceding ALU operation (not shown above), the result would need to be forwarded to prevent a stall.

Data Hazard Classification

A hazard is created whenever there is a dependence between instructions, and they are close enough that the overlap caused by pipelining would change the order of access to an operand. Our example hazards have all been with register operands, but it is also possible for a pair of instructions to create a dependence by writing and reading the same memory location. In our DLX pipeline, however, memory references are always kept in order, preventing this type of hazard from arising. Cache misses could cause the memory references to get out of order if we allowed the processor to continue working on later instructions, while an earlier instruction that missed the cache was accessing memory. For the DLX pipeline we stall the entire pipeline on a cache miss, effectively making the instruction

that contained the miss run for multiple clock cycles. In the next chapter, we will discuss machines that allow loads and stores to be executed in an order different from that in the program, which will introduce new problems. All the data hazards discussed in this chapter involve registers within the CPU.

Data hazards may be classified as one of three types, depending on the order of read and write accesses in the instructions. By convention, the hazards are named by the ordering in the program that must be preserved by the pipeline. Consider two instructions *i* and *j*, with *i* occurring before *j*. The possible data hazards are

- RAW *(read after write) — j* tries to read a source before *i* writes it, so *j* incorrectly gets the old value. This is the most common type of hazard and the kind that we used forwarding to overcome in Figures 3.10 and 3.11.
- WAW *(write after write) — j* tries to write an operand before it is written by *i*. The writes end up being performed in the wrong order, leaving the value written by *i* rather than the value written by *j* in the destination. This hazard is present only in pipelines that write in more than one pipe stage (or allow an instruction to proceed even when a previous instruction is stalled). The DLX integer pipeline writes a register only in WB and avoids this class of hazards. If we made two changes to the DLX pipeline, WAW hazards would be possible. First, we could move write back for an ALU operation into the MEM stage, since the data value is available by then. Second, suppose that the data memory access took two pipe stages. Here is a sequence of two instructions showing the execution in this revised pipeline, highlighting the pipe stage that writes the result:

`LW R1,0(R2)`	IF	ID	EX	MEM1	MEM2	**WB**
`ADD R1,R2,R3`		IF	ID	EX	**WB**	

  Unless this hazard is avoided, execution of this sequence on this revised pipeline will leave the result of the first write (the `LW`) in R1, rather than the result of the `ADD`!

  Allowing writes in different pipe stages introduces other problems, since two instructions can try to write during the same clock cycle. When we discuss the DLX FP pipeline (section 3.7), which has both writes in different stages and different pipeline lengths, we will deal with both write conflicts and WAW hazards in detail.

- WAR *(write after read) — j* tries to write a destination before it is read by *i*, so *i* incorrectly gets the new value. This cannot happen in our example pipeline because all reads are early (in ID) and all writes are late (in WB). This hazard occurs when there are some instructions that write results early in the instruction pipeline, and other instructions that read a source late in the pipeline.

Because of the natural structure of a pipeline, which typically reads values before it writes results, such hazards are rare. Pipelines for complex instruction sets that support autoincrement addressing and require operands to be read late in the pipeline could create a WAR hazard. If we modified the DLX pipeline as in the above example and also read some operands late, such as the source value for a store instruction, a WAR hazard could occur. Here is the pipeline timing for such a potential hazard, highlighting the stage where the conflict occurs:

`SW 0(R1),R2`	IF	ID	EX	MEM1	**MEM2**	WB
`ADD R2,R3,R4`		IF	ID	EX	**WB**	

If the `SW` reads R2 during the second half of its MEM2 stage and the `ADD` writes R2 during the first half of its WB stage, the `SW` will incorrectly read and store the value produced by the `ADD`. In the DLX pipeline, reading all operands from the register file during ID avoids this hazard; however, in the next chapter, we will see how these hazards occur more easily when instructions are executed out of order.

Note that the RAR *(read after read)* case is not a hazard.

Data Hazards Requiring Stalls

Unfortunately, not all potential data hazards can be handled by bypassing. Consider the following sequence of instructions:

```
LW      R1,0(R2)
SUB     R4,R1,R5
AND     R6,R1,R7
OR      R8,R1,R9
```

The pipelined datapath with the bypass paths for this example is shown in Figure 3.12. This case is different from the situation with back-to-back ALU operations. The `LW` instruction does not have the data until the end of clock cycle 4 (its MEM cycle), while the `SUB` instruction needs to have the data by the beginning of that clock cycle. Thus, the data hazard from using the result of a load instruction cannot be completely eliminated with simple hardware. As Figure 3.12 shows, such a forwarding path would have to operate backward in time—a capability not yet available to computer designers! We *can* forward the result immediately to the ALU from the MEM/WB registers for use in the `AND` operation, which begins two clock cycles after the load. Likewise, the `OR` instruction has no problem, since it receives the value through the register file. For the `SUB` instruction, the forwarded result arrives too late—at the end of a clock cycle, when it is needed at the beginning.

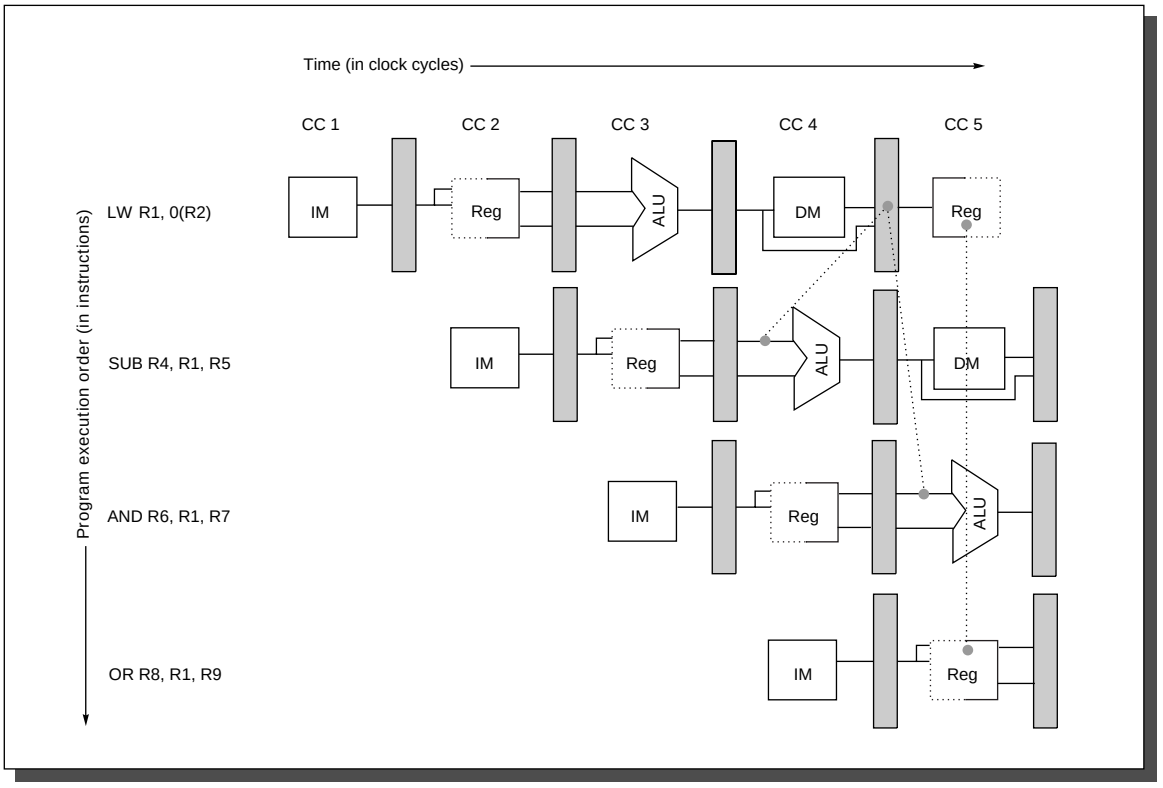

FIGURE 3.12 The load instruction can bypass its results to the `AND` and `OR` instructions, but not to the `SUB`, since that would mean forwarding the result in "negative time."

The load instruction has a delay or latency that cannot be eliminated by forwarding alone. Instead, we need to add hardware, called a *pipeline interlock*, to preserve the correct execution pattern. In general, a *pipeline interlock* detects a hazard and stalls the pipeline until the hazard is cleared. In this case, the interlock stalls the pipeline, beginning with the instruction that wants to use the data until the source instruction produces it. This pipeline interlock introduces a stall or bubble, just as it did for the structural hazard in section 3.3. The CPI for the stalled instruction increases by the length of the stall (one clock cycle in this case). The pipeline with the stall and the legal forwarding is shown in Figure 3.13. Because the stall causes the instructions starting with the `SUB` to move one cycle later in time, the forwarding to the `AND` instruction now goes through the register file, and no forwarding at all is needed for the `OR` instruction. The insertion of the bubble causes the number of cycles to complete this sequence to grow by one. No instruction is started during clock cycle 4 (and none

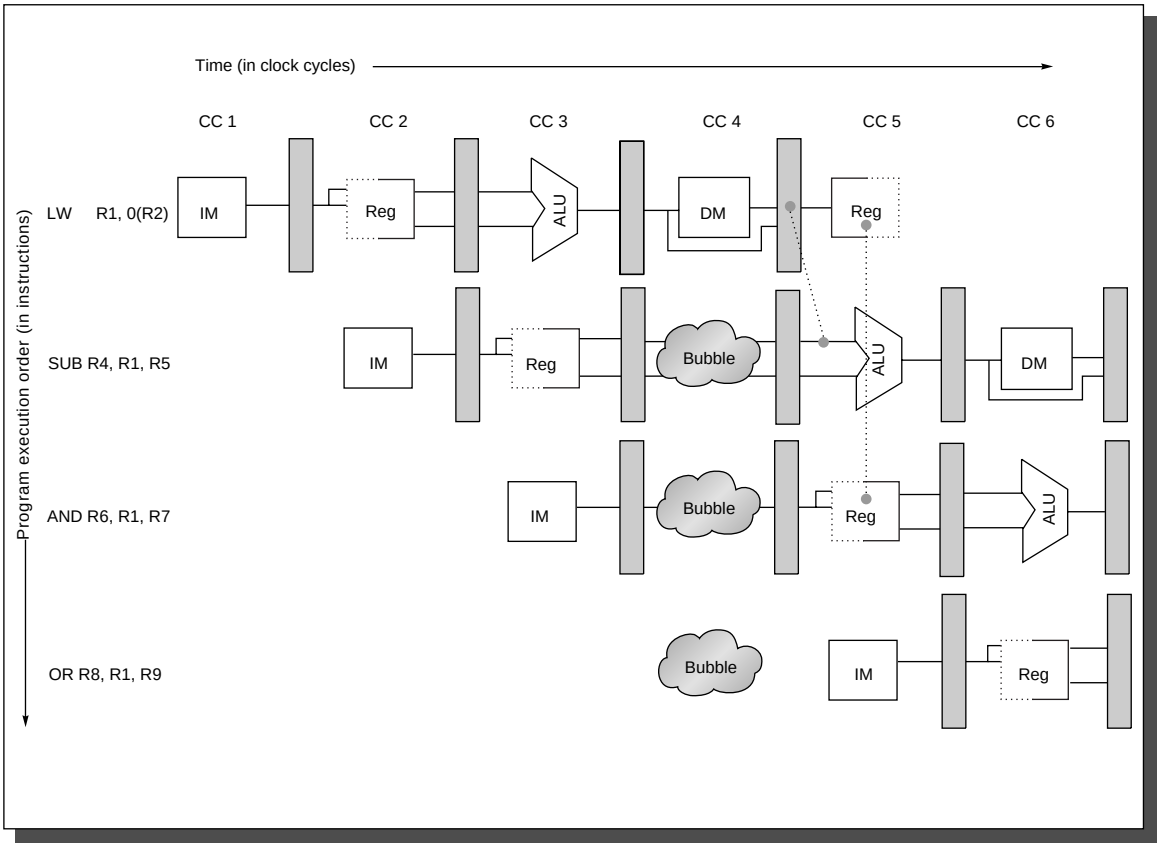

FIGURE 3.13 The load interlock causes a stall to be inserted at clock cycle 4, delaying the SUB instruction and those that follow by one cycle. This delay allows the value to be successfully forwarded on the next clock cycle.

finishes during cycle 6). Figure 3.14 shows the pipeline before and after the stall using a diagram containing only the pipeline stages. We will make extensive use of this more concise form for showing interlocks and stalls in this chapter and the next.

LW R1,0(R2)	IF	ID	EX	MEM	WB				
SUB R4,R1,R5		IF	ID	EX	MEM	WB			
AND R6,R1,R7			IF	ID	EX	MEM	WB		
OR R8,R1,R9				IF	ID	EX	MEM	WB	

LW R1,0(R2)	IF	ID	EX	MEM	WB				
SUB R4,R1,R5		IF	ID	stall	EX	MEM	WB		
AND R6,R1,R7			IF	stall	ID	EX	MEM	WB	
OR R8,R1,R9				stall	IF	ID	EX	MEM	WB

FIGURE 3.14 In the top half, we can see why a stall is needed: the MEM cycle of the load produces a value that is needed in the EX cycle of the SUB, which occurs at the same time. This problem is solved by inserting a stall, as shown in the bottom half.

EXAMPLE Suppose that 30% of the instructions are loads, and half the time the instruction following a load instruction depends on the result of the load. If this hazard creates a single-cycle delay, how much faster is the ideal pipelined machine (with a CPI of 1) that does not delay the pipeline than the real pipeline? Ignore any stalls other than pipeline stalls.

ANSWER The ideal machine will be faster by the ratio of the CPIs. The CPI for an instruction following a load is 1.5, since it stalls half the time. Because loads are 30% of the mix, the effective CPI is $(0.7 \times 1 + 0.3 \times 1.5) = 1.15$. This means that the ideal machine is 1.15 times faster. ■

In the next subsection we consider compiler techniques to reduce these penalties. After that, we look at how to implement hazard detection, forwarding, and interlocks.

Compiler Scheduling for Data Hazards

Many types of stalls are quite frequent. The typical code-generation pattern for a statement such as A = B + C produces a stall for a load of the second data value (C). Figure 3.15 shows that the store of A need not cause another stall, since the result of the addition can be forwarded to the data memory for use by the store.

Rather than just allow the pipeline to stall, the compiler could try to schedule the pipeline to avoid these stalls by rearranging the code sequence to eliminate the hazard. For example, the compiler could try to avoid generating code with a load followed by the immediate use of the load destination register. This technique, called *pipeline scheduling* or *instruction scheduling,* was first used in the 1960s and became an area of major interest in the 1980s, as pipelined machines became more widespread.

`LW R1,B`	IF	ID	EX	MEM	WB				
`LW R2,C`		IF	ID	EX	MEM	WB			
`ADD R3,R1,R2`			IF	ID	stall	EX	MEM	WB	
`SW A,R3`				IF	stall	ID	EX	MEM	WB

FIGURE 3.15 The DLX code sequence for A = B + C. The ADD instruction must be stalled to allow the load of C to complete. The SW need not be delayed further because the forwarding hardware passes the result from the MEM/WB directly to the data memory input for storing.

EXAMPLE Generate DLX code that avoids pipeline stalls for the following sequence:

```
a = b + c;
d = e – f;
```

Assume loads have a latency of one clock cycle.

ANSWER Here is the scheduled code:

```
LW      Rb,b
LW      Rc,c
LW      Re,e    ; swap instructions to avoid stall
ADD     Ra,Rb,Rc
LW      Rf,f
SW      a,Ra    ;store/load exchanged to avoid stall
SUB     Rd,Re,Rf
SW      d,Rd
```

Both load interlocks (`LW Rc, c` to `ADD Ra, Rb, Rc` and `LW Rf, f` to `SUB Rd, Re, Rf`) have been eliminated. There is a dependence between the ALU instruction and the store, but the pipeline structure allows the result to be forwarded. Notice that the use of different registers for the first and second statements was critical for this schedule to be legal. In particular, if the variable e was loaded into the same register as b or c, this schedule would be illegal. In general, pipeline scheduling can increase the register count required. In the next chapter, we will see that this increase can be substantial for machines that can issue multiple instructions in one clock. ■

Many modern compilers try to use instruction scheduling to improve pipeline performance. In the simplest algorithms, the compiler simply schedules using other instructions in the same basic block. A *basic block* is a straight-line code sequence with no transfers in or out, except at the beginning or end. Scheduling such code sequences is easy, since we know that every instruction in the block is executed if the first one is. We can simply make a graph of the dependences among the instructions and order the instructions so as to minimize the stalls. For a simple pipeline like the DLX integer pipeline with only short latencies (the only delay is one cycle on loads), a scheduling strategy focusing on basic blocks is adequate. Figure 3.16 shows the frequency that stalls are required for load results, assuming a single-cycle delay for loads. As you can see, this process is more effective for floating-point programs that have significant amounts of parallelism among instructions. As pipelining becomes more extensive and the effective pipeline latencies grow, more ambitious scheduling schemes are needed; these are discussed in detail in the next chapter.

Implementing the Control for the DLX Pipeline

The process of letting an instruction move from the instruction decode stage (ID) into the execution stage (EX) of this pipeline is usually called *instruction issue*; an instruction that has made this step is said to have *issued*. For the DLX integer pipeline, all the data hazards can be checked during the ID phase of the pipeline.

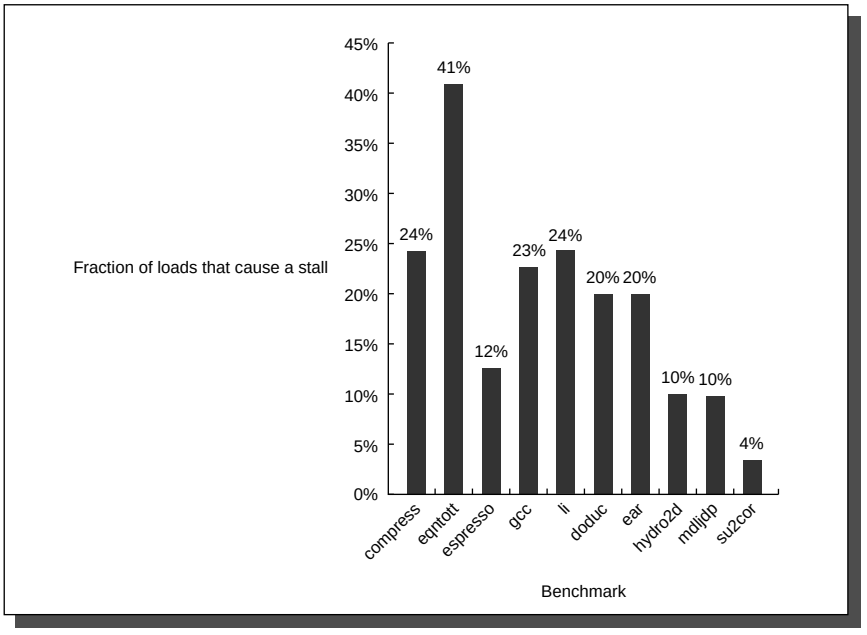

FIGURE 3.16 Percentage of the loads that result in a stall with the DLX pipeline. This chart shows the frequency of stalls remaining in scheduled code that was globally optimized before scheduling. Global optimization actually makes scheduling relatively harder because there are fewer candidates for scheduling into delay slots, as we discuss in *Fallacies and Pitfalls*. The pipeline slot after a load is often called the *load delay* or *delay slot.* In general, it is easier to schedule the delay slots in FP programs, since they are more regular and the analysis is easier. Hence fewer loads stall in the FP programs: an average of 13% of the loads versus 25% on the integer programs. The actual performance impact depends on the load frequency, which varies from 19% to 34% with an average of 24%.The contribution to CPI runs from 0.01 cycles per instruction to 0.15 cycles per instruction.

If a data hazard exists, the instruction is stalled before it is issued. Likewise, we can determine what forwarding will be needed during ID and set the appropriate controls then. Detecting interlocks early in the pipeline reduces the hardware complexity because the hardware never has to suspend an instruction that has updated the state of the machine, unless the entire machine is stalled. Alternatively, we can detect the hazard or forwarding at the beginning of a clock cycle that uses an operand (EX and MEM for this pipeline). To show the differences in these two approaches, we will show how the interlock for a RAW hazard with the source coming from a load instruction (called a *load interlock*) can be implemented by a check in ID, while the implementation of forwarding paths to the ALU inputs can be done during EX. Figure 3.17 lists the variety of circumstances that we must handle.

Situation	Example code sequence	Action

No dependence	`LW R1,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,R6,R7`
	`OR R9,R6,R7`
		No hazard possible because no dependence exists on R1 in the immediately following three instructions.

Dependence requiring stall	`LW R1,45(R2)`
	`ADD R5,R1,R7`
	`SUB R8,R6,R7`
	`OR R9,R6,R7`
		Comparators detect the use of R1 in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX.

Dependence overcome by forwarding	`LW R1,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,R1,R7`
	`OR R9,R6,R7`
		Comparators detect use of R1 in SUB and forward result of load to ALU in time for SUB to begin EX.

Dependence with accesses in order	`LW R1,45(R2)`
	`ADD R5,R6,R7`
	`SUB R8,R6,R7`
	`OR R9,R1,R7`
		No action required because the read of R1 by OR occurs in the second half of the ID phase, while the write of the loaded data occurred in the first half.

FIGURE 3.17 Situations that the pipeline hazard detection hardware can see by comparing the destination and sources of adjacent instructions. This table indicates that the only comparison needed is between the destination and the sources on the two instructions following the instruction that wrote the destination. In the case of a stall, the pipeline dependences will look like the third case once execution continues. Of course hazards that involve R0 can be ignored since the register always contains 0, and the test above could be extended to do this.

Let's start with implementing the load interlock. If there is a RAW hazard with the source instruction being a load, the load instruction will be in the EX stage when an instruction that needs the load data will be in the ID stage. Thus, we can describe all the possible hazard situations with a small table, which can be directly translated to an implementation. Figure 3.18 shows a table that detects all load interlocks when the instruction using the load result is in the ID stage.

Opcode field of ID/EX ($ID/EX.IR_{0..5}$)	Opcode field of IF/ID ($IF/ID.IR_{0..5}$)	Matching operand fields
Load	Register-register ALU	$ID/EX.IR_{11..15} == IF/ID.IR_{6..10}$
Load	Register-register ALU	$ID/EX.IR_{11..15} == IF/ID.IR_{11..15}$
Load	Load, store, ALU immediate, or branch	$ID/EX.IR_{11..15} == IF/ID.IR_{6..10}$

FIGURE 3.18 The logic to detect the need for load interlocks during the ID stage of an instruction requires three comparisons. Lines 1 and 2 of the table test whether the load destination register is one of the source registers for a register-register operation in ID. Line 3 of the table determines if the load destination register is a source for a load or store effective address, an ALU immediate, or a branch test. Remember that the IF/ID register holds the state of the instruction in ID, which potentially uses the load result, while ID/EX holds the state of the instruction in EX, which is the potential load instruction.

Once a hazard has been detected, the control unit must insert the pipeline stall and prevent the instructions in the IF and ID stages from advancing. As we said in section 3.2, all the control information is carried in the pipeline registers. (Carrying the instruction along is enough, since all control is derived from it.) Thus, when we detect a hazard we need only change the control portion of the ID/EX pipeline register to all 0s, which happens to be a no-op (an instruction that does nothing, such as `ADD R0,R0,R0`). In addition, we simply recirculate the contents of the IF/ID registers to hold the stalled instruction. In a pipeline with more complex hazards, the same ideas would apply: We can detect the hazard by comparing some set of pipeline registers and shift in no-ops to prevent erroneous execution.

Implementing the forwarding logic is similar, though there are more cases to consider. The key observation needed to implement the forwarding logic is that the pipeline registers contain both the data to be forwarded as well as the source and destination register fields. All forwarding logically happens from the ALU or data memory output to the ALU input, the data memory input, or the zero detection unit. Thus, we can implement the forwarding by a comparison of the destination registers of the IR contained in the EX/MEM and MEM/WB stages against the source registers of the IR contained in the ID/EX and EX/MEM registers. Figure 3.19 shows the comparisons and possible forwarding operations where the destination of the forwarded result is an ALU input for the instruction currently in EX. The Exercises ask you to add the entries when the result is forwarded to the data memory. The last possible forwarding destination is the zero detect unit, whose forwarding paths look the same as those that are needed when the destination instruction is an ALU immediate.

In addition to the comparators and combinational logic that we need to determine when a forwarding path needs to be enabled, we also need to enlarge the multiplexers at the ALU inputs and add the connections from the pipeline registers that are used to forward the results. Figure 3.20 shows the relevant segments of the pipelined datapath with the additional multiplexers and connections in place.

For DLX, the hazard detection and forwarding hardware is reasonably simple; we will see that things become somewhat more complicated when we extend this pipeline to deal with floating point. Before we do that, we need to handle branches.

Pipeline register containing source instruction	Opcode of source instruction	Pipeline register containing destination instruction	Opcode of destination instruction	Destination of the forwarded result	Comparison (if equal then forward)
EX/MEM	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$EX/MEM.IR_{16..20} == ID/EX.IR_{6..10}$
EX/MEM	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	$EX/MEM.IR_{16..20} == ID/EX.IR_{11..15}$
MEM/WB	Register-register ALU	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$MEM/WB.IR_{16..20} == ID/EX.IR_{6..10}$
MEM/WB	Register-register ALU	ID/EX	Register-register ALU	Bottom ALU input	$MEM/WB.IR_{16..20} == ID/EX.IR_{11..15}$
EX/MEM	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$EX/MEM.IR_{11..15} == ID/EX.IR_{6..10}$
EX/MEM	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	$EX/MEM.IR_{11..15} == ID/EX.IR_{11..15}$
MEM/WB	ALU immediate	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$MEM/WB.IR_{11..15} == ID/EX.IR_{6..10}$
MEM/WB	ALU immediate	ID/EX	Register-register ALU	Bottom ALU input	$MEM/WB.IR_{11..15} == ID/EX.IR_{11..15}$
MEM/WB	Load	ID/EX	Register-register ALU, ALU immediate, load, store, branch	Top ALU input	$MEM/WB.IR_{11..15} == ID/EX.IR_{6..10}$
MEM/WB	Load	ID/EX	Register-register ALU	Bottom ALU input	$MEM/WB.IR_{11..15} == ID/EX.IR_{11..15}$

FIGURE 3.19 Forwarding of data to the two ALU inputs (for the instruction in EX) can occur from the ALU result (in EX/MEM or in MEM/WB) or from the load result in MEM/WB. There are 10 separate comparisons needed to tell whether a forwarding operation should occur. The top and bottom ALU inputs refer to the inputs corresponding to the first and second ALU source operands, respectively, and are shown explicitly in Figure 3.1 on page 130 and in Figure 3.20 on page 161. Remember that the pipeline latch for destination instruction in EX is ID/EX, while the source values come from the ALUOutput portion of EX/MEM or MEM/WB or the LMD portion of MEM/WB. There is one complication not addressed by this logic: dealing with multiple instructions that write the same register. For example, during the code sequence `ADD R1, R2, R3; ADDI R1, R1, #2; SUB R4, R3, R1,` the logic must ensure that the `SUB` instruction uses the result of the `ADDI` instruction rather than the result of the `ADD` instruction. The logic shown above can be extended to handle this case by simply testing that forwarding from MEM/WB is enabled only when forwarding from EX/MEM is not enabled for the same input. Because the `ADDI` result will be in EX/MEM, it will be forwarded, rather than the `ADD` result in MEM/WB.

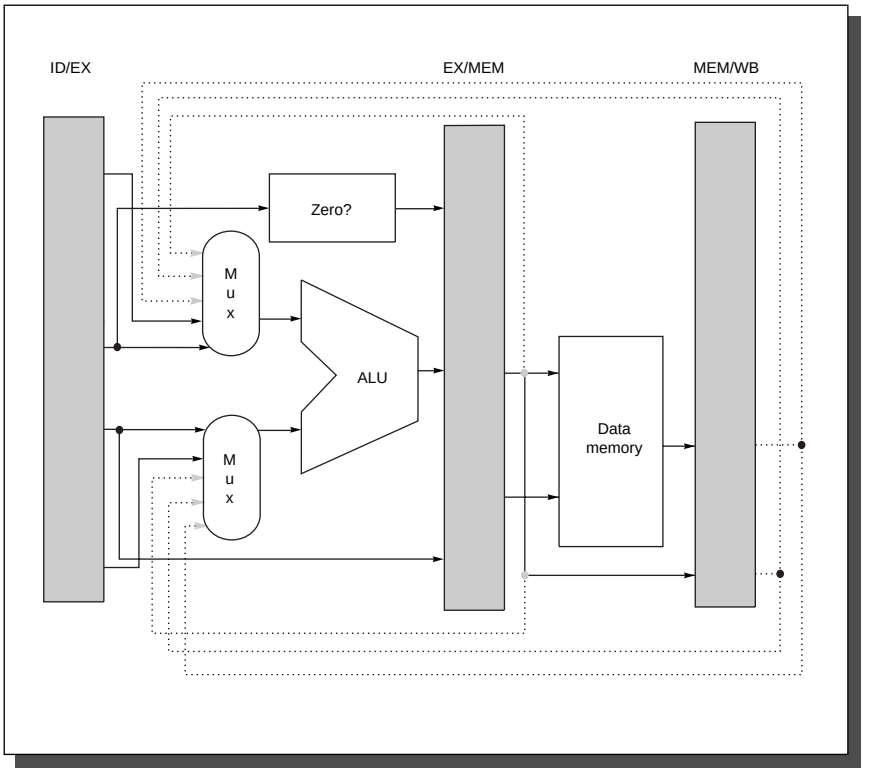

FIGURE 3.20 Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs. The paths correspond to a bypass of (1) the ALU output at the end of the EX, (2) the ALU output at the end of the MEM stage, and (3) the memory output at the end of the MEM stage.

3.5 Control Hazards

Control hazards can cause a greater performance loss for our DLX pipeline than do data hazards. When a branch is executed, it may or may not change the PC to something other than its current value plus 4. Recall that if a branch changes the PC to its target address, it is a *taken* branch; if it falls through, it is *not taken*, or *untaken.* If instruction *i* is a taken branch, then the PC is normally not changed until the end of MEM, after the completion of the address calculation and comparison, as shown in Figure 3.4 (page 134) and Figure 3.5 (page 136).

The simplest method of dealing with branches is to stall the pipeline as soon as we detect the branch until we reach the MEM stage, which determines the new PC. Of course, we do not want to stall the pipeline until we know that the instruction is a branch; thus, the stall does not occur until after the ID stage, and the pipeline behavior looks like that shown in Figure 3.21. This control hazard stall must

be implemented differently from a data hazard stall, since the IF cycle of the instruction following the branch must be repeated as soon as we know the branch outcome. Thus, the first IF cycle is essentially a stall, because it never performs useful work. This stall can be implemented by setting the IF/ID register to zero for the three cycles. You may have noticed that if the branch is untaken, then the repetition of the IF stage is unnecessary since the correct instruction was indeed fetched. We will develop several schemes to take advantage of this fact shortly, but first, let's examine how we could reduce the worst-case branch penalty.

Branch instruction	IF	ID	EX	MEM	WB					
Branch successor		IF	*stall*	*stall*	IF	ID	EX	MEM	WB	
Branch successor + 1						IF	ID	EX	MEM	WB
Branch successor + 2							IF	ID	EX	MEM
Branch successor + 3								IF	ID	EX
Branch successor + 4									IF	ID
Branch successor + 5										IF

FIGURE 3.21 A branch causes a three-cycle stall in the DLX pipeline: One cycle is a repeated IF cycle and two cycles are idle. The instruction after the branch is fetched, but the instruction is ignored, and the fetch is restarted once the branch target is known. It is probably obvious that if the branch is not taken, the second IF for branch successor is redundant. This will be addressed shortly.

Three clock cycles wasted for every branch is a significant loss. With a 30% branch frequency and an ideal CPI of 1, the machine with branch stalls achieves only about *half* the ideal speedup from pipelining! Thus, reducing the branch penalty becomes critical. The number of clock cycles in a branch stall can be reduced by two steps:

1. Find out whether the branch is taken or not taken earlier in the pipeline.
2. Compute the taken PC (i.e., the address of the branch target) earlier.

To optimize the branch behavior, *both* of these must be done—it doesn't help to know the target of the branch without knowing whether the next instruction to execute is the target or the instruction at PC + 4. Both steps should be taken as early in the pipeline as possible.

In DLX, the branches (`BEQZ` and `BNEZ`) require testing a register for equality to zero. Thus, it is possible to complete this decision by the end of the ID cycle by moving the zero test into that cycle. To take advantage of an early decision on whether the branch is taken, both PCs (taken and untaken) must be computed early. Computing the branch target address during ID requires an additional adder because the main ALU, which has been used for this function so far, is not usable until EX. Figure 3.22 shows the revised pipelined datapath. With the separate adder and a branch decision made during ID, there is only a one-clock-cycle stall on branches. Although this reduces the branch delay to one cycle, it means that an ALU instruction followed by a branch on the result of the instruction will in-

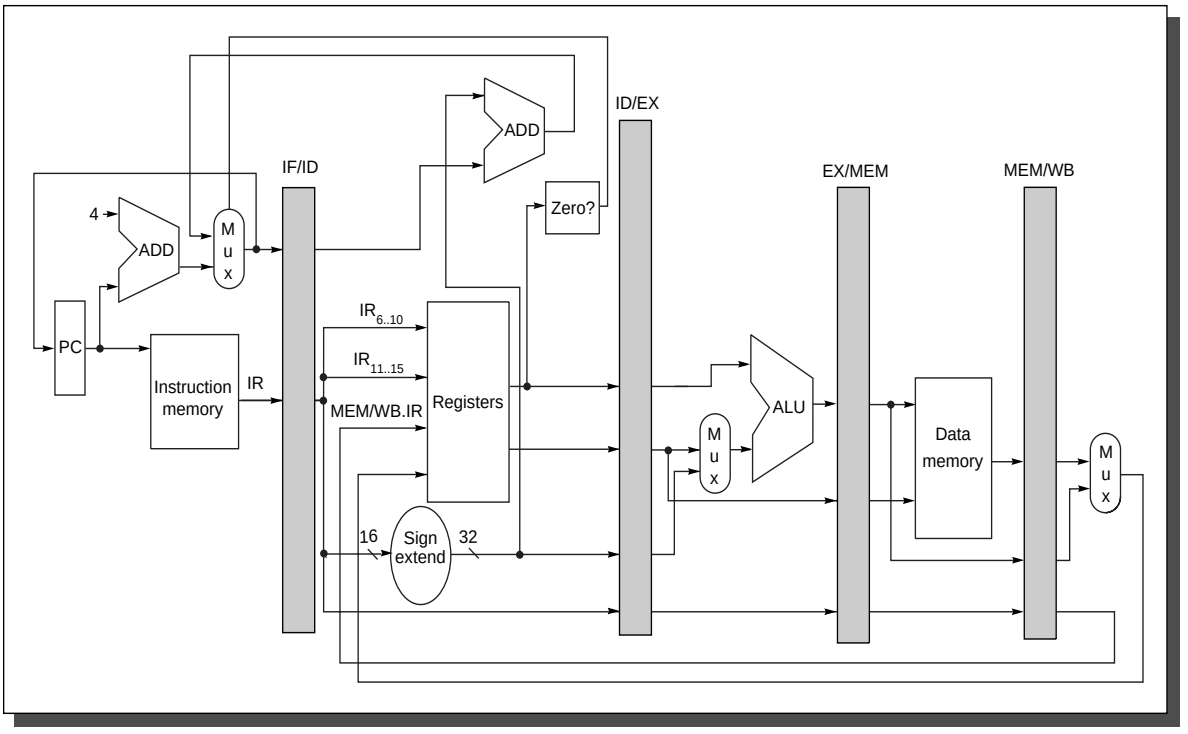

FIGURE 3.22 The stall from branch hazards can be reduced by moving the zero test and branch target calculation into the ID phase of the pipeline. Notice that we have made two important changes, each of which removes one cycle from the three cycle stall for branches. The first change is to move both the branch address target calculation and the branch condition decision to the ID cycle. The second change is to write the PC of the instruction in the IF phase, using either the branch target address computed during ID or the incremented PC computed during IF. In comparison, Figure 3.4 obtained the branch target address from the EX/MEM register and wrote the result during the MEM clock cycle. As mentioned in Figure 3.4, the PC can be thought of as a pipeline register (e.g., as part of ID/IF), which is written with the address of the next instruction at the end of each IF cycle.

cur a data hazard stall. Figure 3.23 shows the branch portion of the revised pipeline table from Figure 3.5 (page 136).

In some machines, branch hazards are even more expensive in clock cycles than in our example, since the time to evaluate the branch condition and compute the destination can be even longer. For example, a machine with separate decode and register fetch stages will probably have a *branch delay*—the length of the control hazard—that is at least one clock cycle longer. The branch delay, unless it is dealt with, turns into a branch penalty. Many older machines that implement more complex instruction sets have branch delays of four clock cycles or more, and large, deeply pipelined machines often have branch penalties of six or seven. In general, the deeper the pipeline, the worse the branch penalty in clock cycles. Of course, the relative performance effect of a longer branch penalty depends on the overall CPI of the machine. A high CPI machine can afford to have more expensive branches because the percentage of the machine's performance that will be lost from branches is less.

Pipe stage	Branch instruction
IF	IF/ID.IR ← Mem[PC]; **IF/ID.NPC,PC ← (if ((IF/ID.opcode == branch) & (Regs[IF/ID.IR$_{6..10}$] op 0)) {IF/ID.NPC + (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$} else {PC+4});**
ID	ID/EX.A ← Regs[IF/ID.IR$_{6..10}$]; ID/EX.B ← Regs[IF/ID.IR$_{11..15}$]; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← (IF/ID.IR$_{16}$)^{16}##IF/ID.IR$_{16..31}$
EX	
MEM	
WB	

FIGURE 3.23 This revised pipeline structure is based on the original in Figure 3.5, page 136. It uses a separate adder, as in Figure 3.22, to compute the branch target address during ID. The operations that are new or have changed are in bold. Because the branch target address addition happens during ID, it will happen for all instructions; the branch condition (Regs[IF/ID.IR$_{6..10}$] *op* 0) will also be done for all instructions. The selection of the sequential PC or the branch target PC still occurs during IF, but it now uses values from the ID/EX register, which correspond to the values set by the previous instruction. This change reduces the branch penalty by two cycles: one from evaluating the branch target and condition earlier and one from controlling the PC selection on the same clock rather than on the next clock. Since the value of cond is set to 0, unless the instruction in ID is a taken branch, the machine must decode the instruction before the end of ID. Because the branch is done by the end of ID, the EX, MEM, and WB stages are unused for branches. An additional complication arises for jumps that have a longer offset than branches. We can resolve this by using an additional adder that sums the PC and lower 26 bits of the IR.

Before talking about methods for reducing the pipeline penalties that can arise from branches, let's take a brief look at the dynamic behavior of branches.

Branch Behavior in Programs

Because branches can dramatically affect pipeline performance, we should look at their behavior to get some ideas about how the penalties of branches and jumps might be reduced. We already know something about branch frequencies from our programs in Chapter 2. Figure 3.24 reviews the overall frequency of control-flow operations for our SPEC subset on DLX and gives the breakdown between branches and jumps. Conditional branches are also broken into forward and backward branches.

The integer benchmarks show conditional branch frequencies of 14% to 16%, with much lower unconditional branch frequencies (though li has a large number because of its high procedure call frequency). For the FP benchmarks, the behavior is much more varied with a conditional branch frequency of 3% up to 12%, but an overall average for both conditional branches and unconditional branches that is lower than for the integer benchmarks. Forward branches dominate backward branches by about 3.7 to 1 on average.

Since the performance of pipelining schemes for branches may depend on whether or not branches are taken, this data becomes critical. Figure 3.25 shows the frequency of forward and backward branches that are taken as a fraction of all conditional branches. Totaling the two columns shows that 67% of the condition-

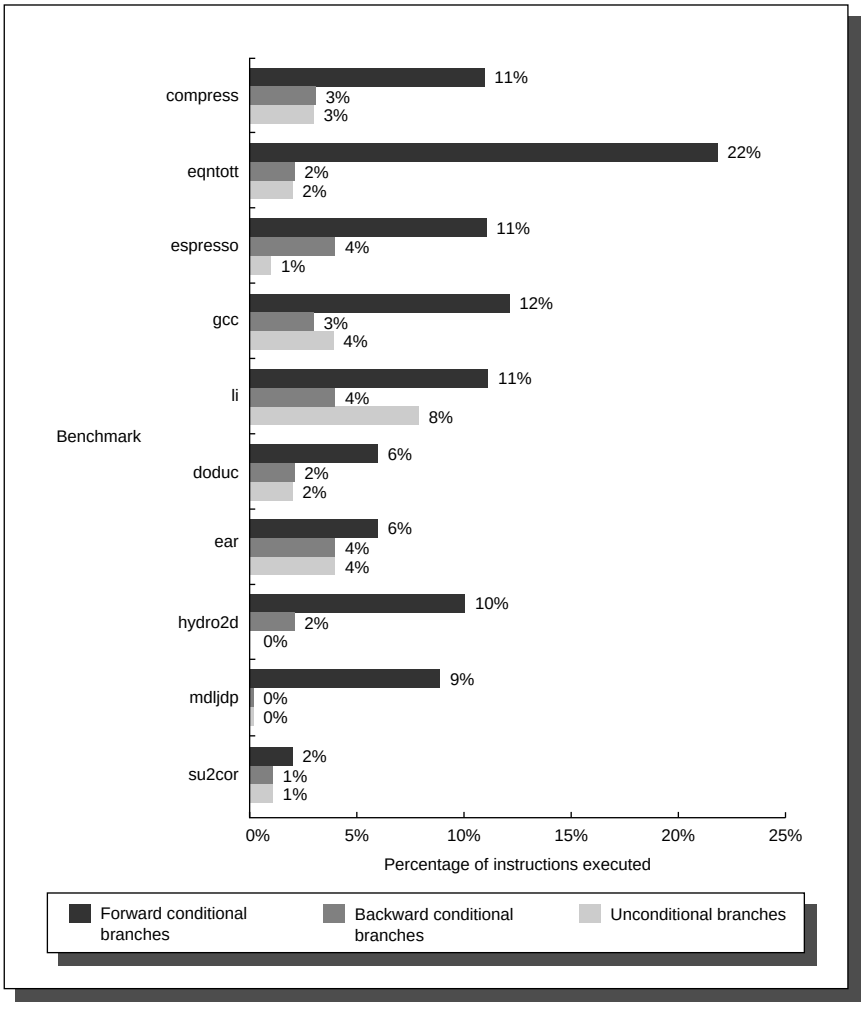

FIGURE 3.24 The frequency of instructions (branches, jumps, calls, and returns) that may change the PC. The unconditional column includes unconditional branches and jumps (these differ in how the target address is specified), procedure calls, and returns. In all the cases except li, the number of unconditional PC changes is roughly equally divided between those that are for calls or returns and those that are unconditional jumps. In li, calls and returns outnumber jumps and unconditional branches by a factor of 3 (6% versus 2%). Since the compiler uses loop unrolling (described in detail in Chapter 4) as an optimization, the backward conditional branch frequency will be lower, especially for the floating-point programs. Overall, the integer programs average 13% forward conditional branches, 3% backward conditional branches, and 4% unconditional branches. The FP programs average 7%, 2%, and 1%, respectively.

al branches are taken on average. By combining the data in Figures 3.24 and 3.25, we can compute the fraction of forward branches that are taken, which is the probability that a forward branch will be taken. Since backward branches

often form loops, we would expect that the probability of a backward branch being taken is higher than the probability of a forward branch being taken. Indeed, the data, when combined, show that 60% of the forward branches are taken on average and 85% of the backward branches are taken.

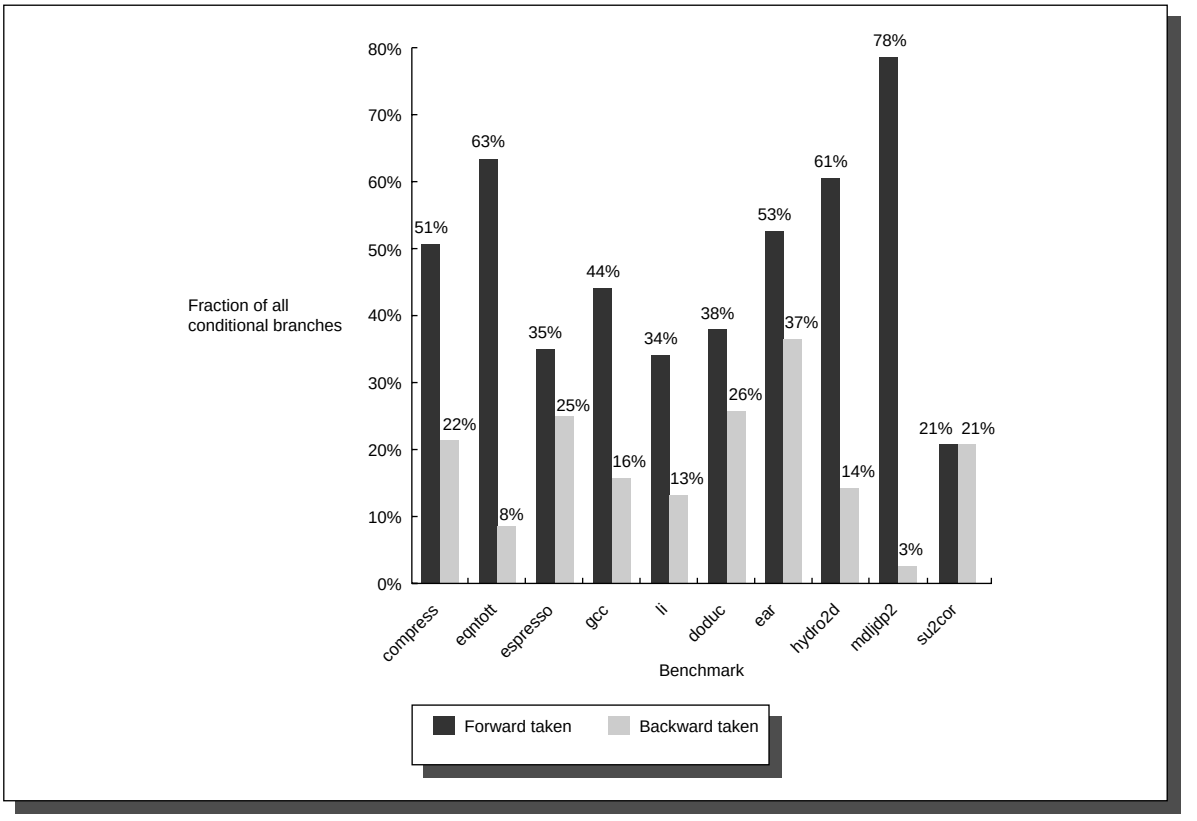

FIGURE 3.25 Together the forward and backward taken branches account for an average of 67% of all conditional branches. Although the backward branches are outnumbered, they are taken with a frequency that is almost 1.5 times higher, contributing substantially to the taken branch frequency. On average, 62% of the branches are taken in the integer programs and 70% in the FP programs. Note the wide disparity in behavior between a program like su2cor and mdljdp2; these variations make it challenging to predict the branch behavior very accurately. As in Figure 3.24, the use of loop unrolling affects this data since it removes backward branches that had a high probability of being taken.

Reducing Pipeline Branch Penalties

There are many methods for dealing with the pipeline stalls caused by branch delay; we discuss four simple compile-time schemes in this subsection. In these four schemes the actions for a branch are static—they are fixed for each branch during the entire execution. The software can try to minimize the branch penalty

using knowledge of the hardware scheme and of branch behavior. After discussing these schemes, we examine compile-time branch prediction, since these branch optimizations all rely on such technology. In the next chapter, we look both at more powerful compile-time schemes (such as loop unrolling) that reduce the frequency of loop branches and at dynamic hardware-based prediction schemes.

The simplest scheme to handle branches is to *freeze* or *flush* the pipeline, holding or deleting any instructions after the branch until the branch destination is known. The attractiveness of this solution lies primarily in its simplicity both for hardware and software. It is the solution used earlier in the pipeline shown in Figure 3.21. In this case the branch penalty is fixed and cannot be reduced by software.

A higher performance, and only slightly more complex, scheme is to treat every branch as not taken, simply allowing the hardware to continue as if the branch were not executed. Here, care must be taken not to change the machine state until the branch outcome is definitely known. The complexity that arises from having to know when the state might be changed by an instruction and how to "back out" a change might cause us to choose the simpler solution of flushing the pipeline in machines with complex pipeline structures.

In the DLX pipeline, this *predict-not-taken* or *predict-untaken* scheme is implemented by continuing to fetch instructions as if the branch were a normal instruction. The pipeline looks as if nothing out of the ordinary is happening. If the branch is taken, however, we need to turn the fetched instruction into a no-op (simply by clearing the IF/ID register) and restart the fetch at the target address. Figure 3.26 shows both situations.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

FIGURE 3.26 The predict-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom). When the branch is untaken, determined during ID, we have fetched the fall-through and just continue. If the branch is taken during ID, we restart the fetch at the branch target. This causes all instructions following the branch to stall one clock cycle.

An alternative scheme is to treat every branch as taken. As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing at the target. Because in our DLX pipeline we don't know the target address any earlier than we know the branch outcome, there is no advantage in this approach for DLX. In some machines—especially those with implicitly set condition codes or more powerful (and hence slower) branch conditions—the branch target is known before the branch outcome, and a predict-taken scheme might make sense. In either a predict-taken or predict-not-taken scheme, the compiler can improve performance by organizing the code so that the most frequent path matches the hardware's choice. Our fourth scheme provides more opportunities for the compiler to improve performance.

A fourth scheme in use in some machines is called *delayed branch*. This technique is also used in many microprogrammed control units. In a delayed branch, the execution cycle with a branch delay of length *n* is

```
branch instruction
sequential successor1
sequential successor2
........
sequential successorn
branch target if taken
```

The sequential successors are in the *branch-delay slots*. These instructions are executed whether or not the branch is taken. The pipeline behavior of the DLX pipeline, which would have one branch-delay slot, is shown in Figure 3.27. In

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch-delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Instruction i + 2			IF	ID	EX	MEM	WB		
Instruction i + 3				IF	ID	EX	MEM	WB	
Instruction i + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch-delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

FIGURE 3.27 The behavior of a delayed branch is the same whether or not the branch is taken. The instructions in the delay slot (there is only one delay slot for DLX) are executed. If the branch is untaken, execution continues with the instruction after the branch-delay instruction; if the branch is taken, execution continues at the branch target. When the instruction in the branch-delay slot is also a branch, the meaning is unclear: if the branch is not taken, what should happen to the branch in the branch-delay slot? Because of this confusion, architectures with delay branches often disallow putting a branch in the delay slot.

practice, all machines with delayed branch have a single instruction delay, and we focus on that case.

The job of the compiler is to make the successor instructions valid and useful. A number of optimizations are used. Figure 3.28 shows the three ways in which the branch delay can be scheduled. Figure 3.29 shows the different constraints for each of these branch-scheduling schemes, as well as situations in which they win.

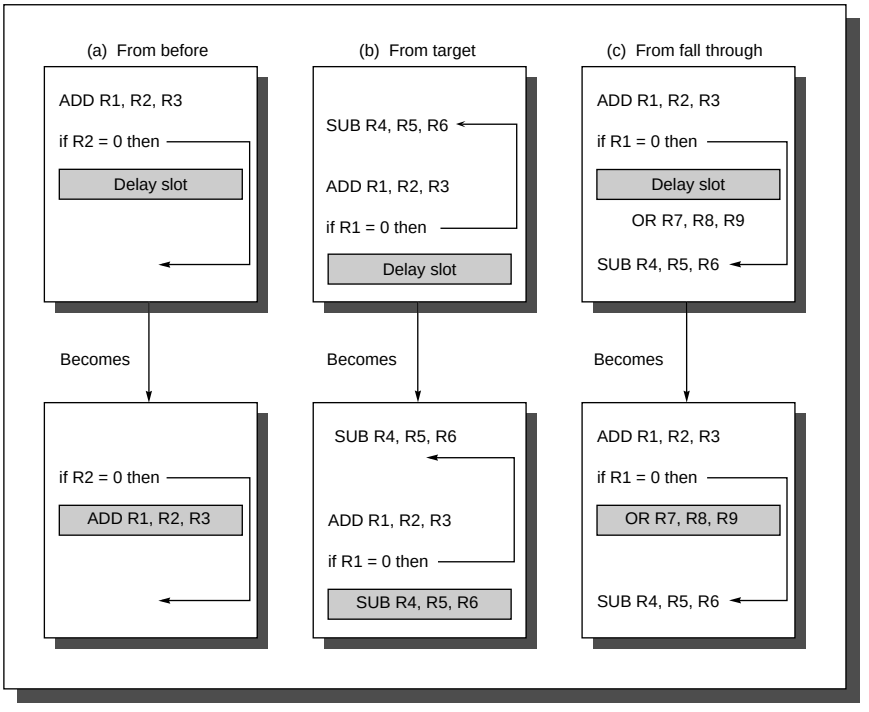

FIGURE 3.28 Scheduling the branch-delay slot. The top box in each pair shows the code before scheduling; the bottom box shows the scheduled code. In (a) the delay slot is scheduled with an independent instruction from before the branch. This is the best choice. Strategies (b) and (c) are used when (a) is not possible. In the code sequences for (b) and (c), the use of R1 in the branch condition prevents the ADD instruction (whose destination is R1) from being moved after the branch. In (b) the branch-delay slot is scheduled from the target of the branch; usually the target instruction will need to be copied because it can be reached by another path. Strategy (b) is preferred when the branch is taken with high probability, such as a loop branch. Finally, the branch may be scheduled from the not-taken fall through as in (c). To make this optimization legal for (b) or (c), it must be OK to execute the moved instruction when the branch goes in the unexpected direction. By OK we mean that the work is wasted, but the program will still execute correctly. This is the case, for example in case (b), if R4 were an unused temporary register when the branch goes in the unexpected direction.

Scheduling strategy	Requirements	Improves performance when?
(a) From before	Branch must not depend on the rescheduled instructions.	Always.
(b) From target	Must be OK to execute rescheduled instructions if branch is not taken. May need to duplicate instructions.	When branch is taken. May enlarge program if instructions are duplicated.
(c) From fall through	Must be OK to execute instructions if branch is taken.	When branch is not taken.

FIGURE 3.29 Delayed-branch scheduling schemes and their requirements. The origin of the instruction being scheduled into the delay slot determines the scheduling strategy. The compiler must enforce the requirements when looking for instructions to schedule the delay slot. When the slots cannot be scheduled, they are filled with no-op instructions. In strategy (b), if the branch target is also accessible from another point in the program—as it would be if it were the head of a loop—the target instructions must be copied and not just moved.

The limitations on delayed-branch scheduling arise from (1) the restrictions on the instructions that are scheduled into the delay slots and (2) our ability to predict at compile time whether a branch is likely to be taken or not. Shortly, we will see how we can better predict branches statically at compile time. To improve the ability of the compiler to fill branch delay slots, most machines with conditional branches have introduced a *cancelling* or *nullifying* branch. In a cancelling branch, the instruction includes the direction that the branch was predicted. When the branch behaves as predicted, the instruction in the branch-delay slot is simply executed as it would normally be with a delayed branch. When the branch is incorrectly predicted, the instruction in the branch-delay slot is simply turned into a no-op. Figure 3.30 shows the behavior of a predicted-taken cancelling branch, both when the branch is taken and untaken.

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch-delay instruction (i + 1)		IF	idle	idle	idle	idle			
Instruction i + 2			IF	ID	EX	MEM	WB		
Instruction i + 3				IF	ID	EX	MEM	WB	
Instruction i + 4					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch-delay instruction (i + 1)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

FIGURE 3.30 The behavior of a predicted-taken cancelling branch depends on whether the branch is taken or not. The instruction in the delay slot is executed only if the branch is taken and is otherwise made into a no-op.

The advantage of cancelling branches is that they eliminate the requirements on the instruction placed in the delay slot, enabling the compiler to use scheduling schemes (b) and (c) of Figure 3.28 without meeting the requirements shown for these schemes in Figure 3.29. Most machines with cancelling branches provide both a noncancelling form (i.e., a regular delayed branch) and a cancelling form, usually cancel if not taken. This combination gains most of the advantages, but does not allow scheduling scheme (c) to be used unless the requirements of Figure 3.29 are met.

Figure 3.31 shows the effectiveness of the branch scheduling in DLX with a single branch-delay slot and both a noncancelling branch and a cancel-if-untaken form. The compiler uses a standard delayed branch whenever possible and then opts for a cancel-if-not-taken branch (also called *branch likely*). The second column shows that almost 20% of the branch delay slots are filled with no-ops. These occur when it is not possible to fill the delay slot, either because the potential candidates are unknown (e.g., for a jump register that will be used in a case statement) or because the successors are also branches. (Branches are not allowed in branch-delay slots because of the confusion in semantics.) The table shows that the

Benchmark	% conditional branches	% conditional branches with empty slots	% conditional branches that are cancelling	% cancelling branches that are cancelled	% branches with cancelled delay slots	Total % branches with empty or cancelled delay slot
compress	14%	18%	31%	43%	13%	31%
eqntott	24%	24%	50%	24%	12%	36%
espresso	15%	29%	19%	21%	4%	33%
gcc	15%	16%	33%	34%	11%	27%
li	15%	20%	55%	48%	26%	46%
Integer average	17%	21%	38%	34%	13%	35%
doduc	8%	33%	12%	62%	7%	40%
ear	10%	37%	36%	14%	5%	42%
hydro2d	12%	0%	69%	24%	17%	17%
mdljdp2	9%	0%	86%	10%	9%	9%
su2cor	3%	7%	17%	57%	10%	17%
FP average	8%	16%	44%	33%	10%	25%
Overall average	12%	18%	41%	34%	12%	30%

FIGURE 3.31 Delayed and cancelling delay branches for DLX allow branch hazards to be hidden 70% of the time on average for these 10 SPEC benchmarks. Empty delay slots cannot be filled at all (most often because the branch target is another branch) in 18% of the branches. Just under half the conditional branches use a cancelling branch, and most of these are not cancelled (65%). The behavior varies widely across benchmarks. When the fraction of conditional branches is added in, the contribution to CPI varies even more widely.

remaining 80% of the branch delay slots are filled nearly equally by standard delayed branches and by cancelling branches. Most of the cancelling branches are not cancelled and hence contribute to useful computation. Figure 3.32 summarizes the performance of the combination of delayed branch and cancelling branch. Overall, 70% of the branch delays are usefully filled, reducing the stall penalty to 0.3 cycles per conditional branch.

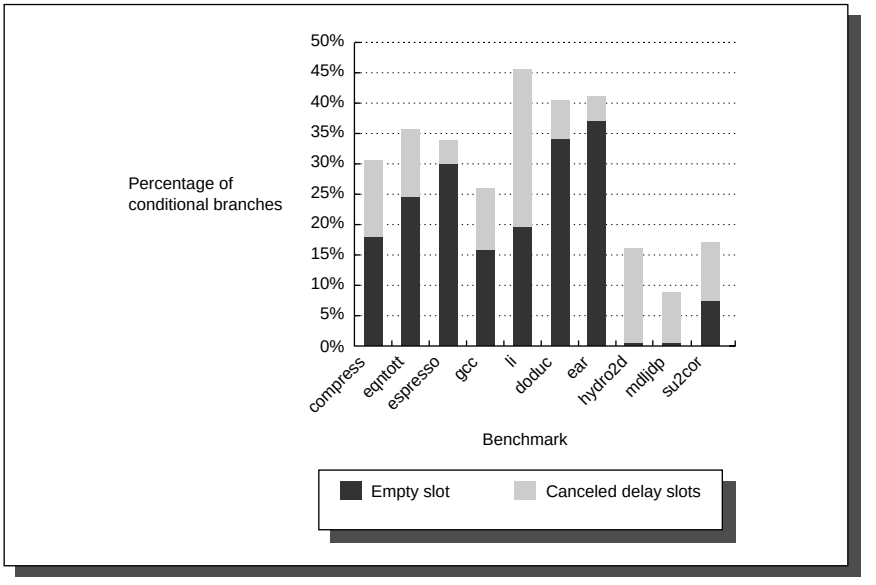

FIGURE 3.32 The performance of delayed and cancelling branches is summarized by showing the fraction of branches either with empty delay slots or with a cancelled delay slot. On average 30% of the branch delay slots are wasted. The integer programs are, on average, worse, wasting an average of 35% of the slots versus 25% for the FP programs. Notice, though, that two of the FP programs waste more branch delay slots than four of the five integer programs.

Delayed branches are an architecturally visible feature of the pipeline. This is the source both of their primary advantage—allowing the use of simple compiler scheduling to reduce branch penalties—and their primary disadvantage—exposing an aspect of the implementation that is likely to change. In the early RISC machines with single-cycle branch delays, the delayed branch approach was attractive, since it yielded good performance with minimal hardware costs. More recently, with deeper pipelines and longer branch delays, a delayed branch approach is less useful since it cannot easily hide the longer delays. With these longer branch delays, most architects have found it necessary to include more powerful hardware schemes for branch prediction (which we will explore in the next chapter), making the delayed branch superfluous.This has led to recent RISC architectures that include both delayed and nondelayed branches or that include only nondelayed branches, relying on hardware prediction.

There is a small additional hardware cost for delayed branches. For a single-cycle delayed branch, the only case that exists in practice, a single extra PC is needed. To understand why an extra PC is needed for the single-cycle delay case, consider when the interrupt occurs for the instruction in the branch-delay slot. If the branch was taken, then the instruction in the delay slot and the branch target have addresses that are not sequential. Thus, we need to save the PCs of both instructions and restore them after the interrupt to restart the pipeline. The two PCs can be kept with the control in the pipeline latches and passed along with the instruction. This makes saving and restoring them easy.

Performance of Branch Schemes

What is the effective performance of each of these schemes? The effective pipeline speedup with branch penalties, assuming an ideal CPI of 1, is

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

Because of the following:

$$\text{Pipeline stall cycles from branches} = \text{Branch frequency} \times \text{Branch penalty}$$

we obtain

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

The branch frequency and branch penalty can have a component from both unconditional and conditional branches. However, the latter dominate since they are more frequent.

Using the DLX measurements in this section, Figure 3.33 shows several hardware options for dealing with branches, along with their performances given as branch penalty and as CPI (assuming a base CPI of 1).

Scheduling scheme	Branch penalty per conditional branch		Penalty per unconditional branch	Average branch penalty per branch		Effective CPI with branch stalls	
	Integer	FP		Integer	FP	Integer	FP
Stall pipeline	1.00	1.00	1.00	1.00	1.00	1.17	1.15
Predict taken	1.00	1.00	1.00	1.00	1.00	1.17	1.15
Predict not taken	0.62	0.70	1.0	0.69	0.74	1.12	1.11
Delayed branch	0.35	0.25	0.0	0.30	0.21	1.06	1.03

FIGURE 3.33 Overall costs of a variety of branch schemes with the DLX pipeline. These data are for our DLX pipeline using the average measured branch frequencies from Figure 3.24 on page 165, the measurements of taken/untaken frequencies from 3.25 on page 166, and the measurements of delay-slot filling from Figure 3.31 on page 171. Shown are both the penalties per branch and the resulting overall CPI including only the effect of branch stalls and assuming a base CPI of 1.

Remember that the numbers in this section are *dramatically* affected by the length of the pipeline delay and the base CPI. A longer pipeline delay will cause an increase in the penalty and a larger percentage of wasted time. A delay of only one clock cycle is small—the R4000 pipeline, which we examine in section 3.9, has a conditional branch delay of three cycles. This results in a much higher penalty.

EXAMPLE For an R4000-style pipeline, it takes three pipeline stages before the branch target address is known and an additional cycle before the branch condition is evaluated, assuming no stalls on the registers in the conditional comparison. This leads to the branch penalties for the three simplest prediction schemes listed in Figure 3.34.

Branch scheme	Penalty unconditional	Penalty untaken	Penalty taken
Flush pipeline	2	3	3
Predict taken	2	3	2
Predict untaken	2	0	3

FIGURE 3.34 Branch penalties for the three simplest prediction schemes for a deeper pipeline.

Find the effective addition to the CPI arising from branches for this pipeline, using the data from the 10 SPEC benchmarks in Figures 3.24 and 3.25.

ANSWER We find the CPIs by multiplying the relative frequency of unconditional, conditional untaken, and conditional taken branches by the respective penalties. These frequencies for the 10 SPEC programs are 4%, 6%, and 10%, respectively. The results are shown in Figure 3.35.

	Addition to the CPI			
Branch scheme	Unconditional branches	Untaken conditional branches	Taken conditional branches	All branches
Frequency of event	4%	6%	10%	20%
Stall pipeline	0.08	0.18	0.30	0.56
Predict taken	0.08	0.18	0.20	0.46
Predict untaken	0.08	0.00	0.30	0.38

FIGURE 3.35 CPI penalties for three branch-prediction schemes and a deeper pipeline.

The differences among the schemes are substantially increased with this longer delay. If the base CPI was 1 and branches were the only source of stalls, the ideal pipeline would be 1.56 times faster than a

pipeline that used the stall-pipeline scheme. The predict-untaken scheme would be 1.13 times better than the stall-pipeline scheme under the same assumptions.

As we will see in section 3.9, the R4000 uses a mixed strategy with a one-cycle, cancelling delayed branch for the first cycle of the branch penalty. For an unconditional branch, a single-cycle stall is always added. For conditional branches, the remaining two cycles of the branch penalty use a predict-not-taken scheme. We will see measurements of the effective branch penalties for this strategy later. ■

Static Branch Prediction: Using Compiler Technology

Delayed branches are a technique that exposes a pipeline hazard so that the compiler can reduce the penalty associated with the hazard. As we saw, the effectiveness of this technique partly depends on whether we correctly guess which way a branch will go. Being able to accurately predict a branch at compile time is also helpful for scheduling data hazards. Consider the following code segment:

```
        LW      R1,0(R2)
        SUB     R1,R1,R3
        BEQZ    R1,L
        OR      R4,R5,R6
        ADD     R10,R4,R3
L:      ADD     R7,R8,R9
```

The dependence of the `SUB` and `BEQZ` on the `LW` instruction means that a stall will be needed after the `LW`. Suppose we knew that this branch was almost always taken and that the value of R7 was not needed on the fall-through path. Then we could increase the speed of the program by moving the instruction `ADD R7,R8,R9` to the position after the `LW`. Correspondingly, if we knew the branch was rarely taken and that the value of R4 was not needed on the taken path, then we could contemplate moving the `OR` instruction after the `LW`. In addition, we can also use the information to better schedule any branch delay, since choosing how to schedule the delay depends on knowing the branch behavior.

To perform these optimizations, we need to predict the branch statically when we compile the program. In the next chapter, we will examine the use of dynamic prediction based on runtime program behavior. We will also look at a variety of compile-time methods for scheduling code; these techniques require static branch prediction and thus the ideas in this section are critical.

There are two basic methods we can use to statically predict branches: by examination of the program behavior and by the use of profile information collected from earlier runs of the program. We saw in Figure 3.25 (page 166) that most branches were taken for both forward and backward branches. Thus, the simplest scheme is to predict a branch as taken. This scheme has an average misprediction

rate for the 10 programs in Figure 3.25 of the untaken branch frequency (34%). Unfortunately, the misprediction rate ranges from not very accurate (59%) to highly accurate (9%).

Another alternative is to predict on the basis of branch direction, choosing backward-going branches to be taken and forward-going branches to be not taken. For some programs and compilation systems, the frequency of forward taken branches may be significantly less than 50%, and this scheme will do better than just predicting all branches as taken. In our SPEC programs, however, more than half of the forward-going branches are taken. Hence, predicting all branches as taken is the better approach. Even for other benchmarks or compilers, direction-based prediction is unlikely to generate an overall misprediction rate of less than 30% to 40%.

A more accurate technique is to predict branches on the basis of profile information collected from earlier runs. The key observation that makes this worthwhile is that the behavior of branches is often bimodally distributed; that is, an individual branch is often highly biased toward taken or untaken. Figure 3.36 shows the success of branch prediction using this strategy. The same input data were used for runs and for collecting the profile; other studies have shown that changing the input so that the profile is for a different run leads to only a small change in the accuracy of profile-based prediction.

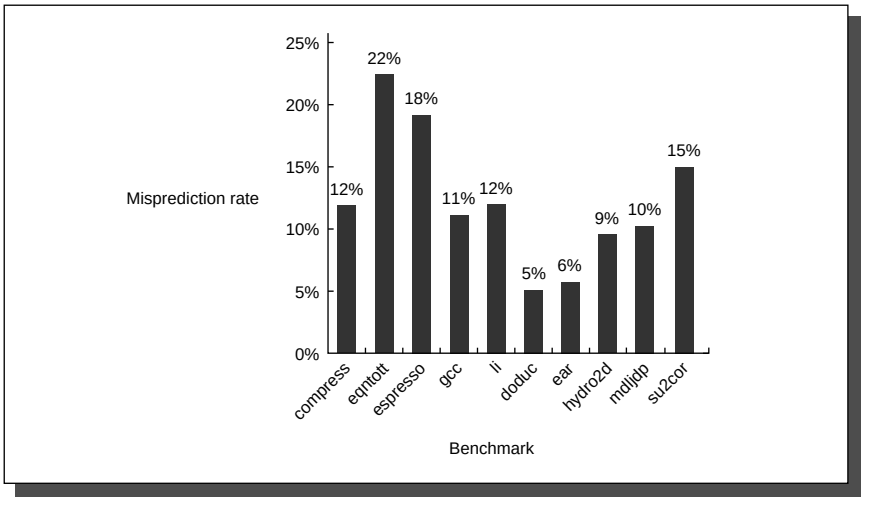

FIGURE 3.36 Misprediction rate for a profile-based predictor varies widely but is generally better for the FP programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%. The actual performance depends on both the prediction accuracy and the branch frequency, which varies from 3% to 24% in Figure 3.31 (page 171); we will examine the combined effect in Figure 3.37.

While we can derive the prediction accuracy of a predict-taken strategy and measure the accuracy of the profile scheme, as in Figure 3.36, the wide range of frequency of conditional branches in these programs, from 3% to 24%, means that the overall frequency of a mispredicted branch varies widely. Figure 3.37 shows the number of instructions executed between mispredicted branches for both a profile-based and a predict-taken strategy. The number varies widely, both because of the variation in accuracy and the variation in branch frequency. On average, the predict-taken strategy has 20 instructions per mispredicted branch and the profile-based strategy has 110. However, these averages are very different for integer and FP programs, as the data in Figure 3.37 show.

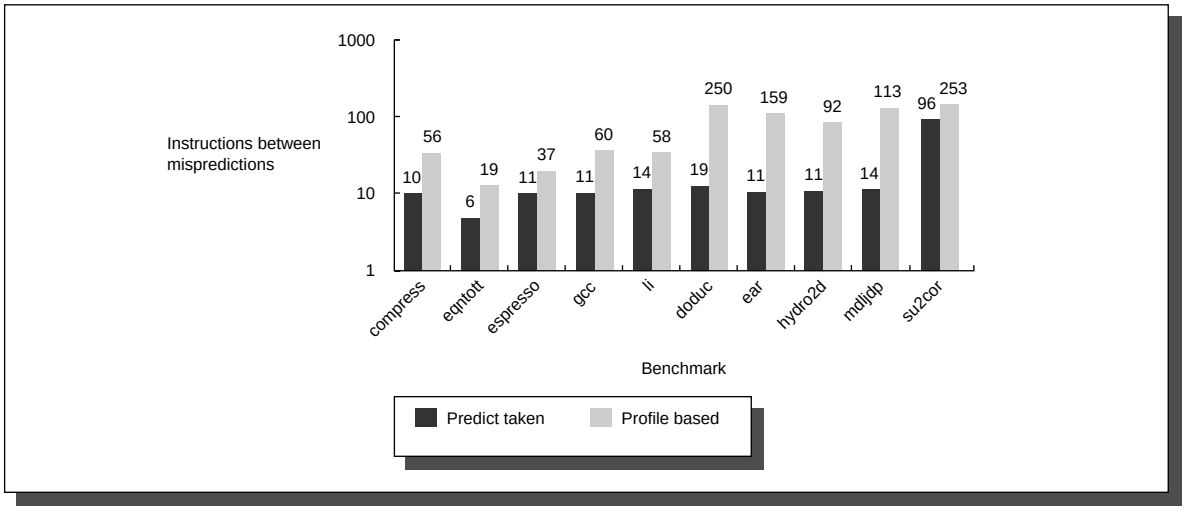

FIGURE 3.37 Accuracy of a predict-taken strategy and a profile-based predictor as measured by the number of instructions executed between mispredicted branches and shown on a log scale. The average number of instructions between mispredictions is 20 for the predict-taken strategy and 110 for the profile-based prediction; however, the standard deviations are large: 27 instructions for the predict-taken strategy and 85 instructions for the profile-based scheme. This wide variation arises because programs such as su2cor have both low conditional branch frequency (3%) and predictable branches (85% accuracy for profiling), while eqntott has eight times the branch frequency with branches that are nearly 1.5 times *less* predictable. The difference between the FP and integer benchmarks as groups is large. For the predict-taken strategy, the average distance between mispredictions for the integer benchmarks is 10 instructions, while it is 30 instructions for the FP programs. With the profile scheme, the distance between mispredictions for the integer benchmarks is 46 instructions, while it is 173 instructions for the FP benchmarks.

Summary: Performance of the DLX Integer Pipeline

We close this section on hazard detection and elimination by showing the total distribution of idle clock cycles for our integer benchmarks when run on the DLX pipeline with software for pipeline scheduling. (After we examine the DLX FP pipeline in section 3.7, we will examine the overall performance of the FP benchmarks.) Figure 3.38 shows the distribution of clock cycles lost to load and branch

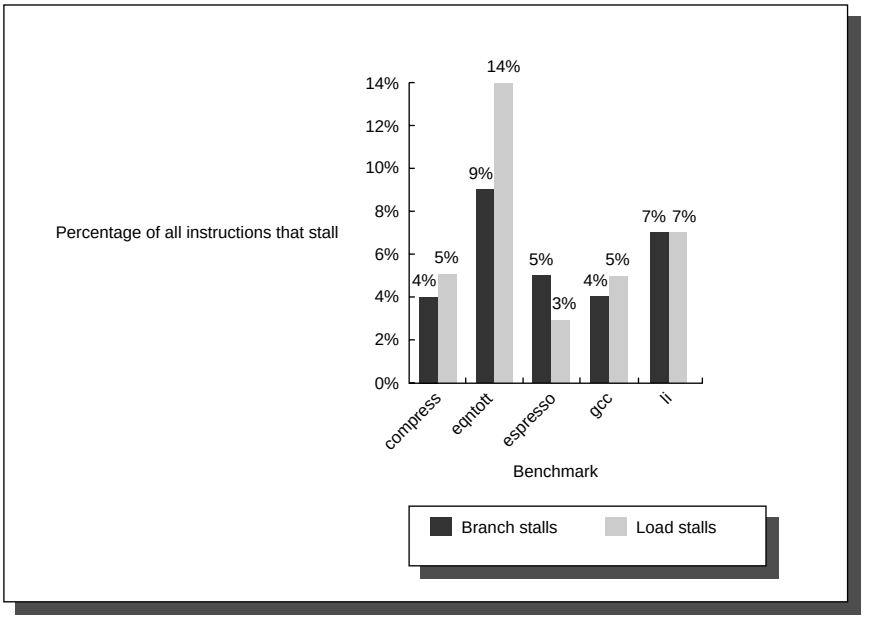

FIGURE 3.38 Percentage of the instructions that cause a stall cycle. This assumes a perfect memory system; the clock-cycle count and instruction count would be identical if there were no integer pipeline stalls. It also assumes the availability of both a basic delayed branch and a cancelling delayed branch, both with one cycle of delay. According to the graph, from 8% to 23% of the instructions cause a stall (or a cancelled instruction), leading to CPIs from pipeline stalls that range from 1.09 to 1.23. The pipeline scheduler fills load delays before branch delays, and this affects the distribution of delay cycles.

delays, which is obtained by combining the separate measurements shown in Figures 3.16 (page 157) and 3.31 (page 171).

Overall the integer programs exhibit an average of 0.06 branch stalls per instruction and 0.05 load stalls per instruction, leading to an average CPI from pipelining (i.e., assuming a perfect memory system) of 1.11. Thus, with a perfect memory system and no clock overhead, pipelining could improve the performance of these five integer SPECint92 benchmarks by 5/1.11 or 4.5 times.

3.6 What Makes Pipelining Hard to Implement?

Now that we understand how to detect and resolve hazards, we can deal with some complications that we have avoided so far. The first part of this section considers the challenges of exceptional situations where the instruction execution order is changed in unexpected ways. In the second part of this section, we discuss some of the challenges raised by different instruction sets.

Dealing with Exceptions

Exceptional situations are harder to handle in a pipelined machine because the overlapping of instructions makes it more difficult to know whether an instruction can safely change the state of the machine. In a pipelined machine, an instruction is executed piece by piece and is not completed for several clock cycles. Unfortunately, other instructions in the pipeline can raise exceptions that may force the machine to abort the instructions in the pipeline before they complete. Before we discuss these problems and their solutions in detail, we need to understand what types of situations can arise and what architectural requirements exist for supporting them.

Types of Exceptions and Requirements

The terminology used to describe exceptional situations where the normal execution order of instruction is changed varies among machines. The terms *interrupt*, *fault*, and *exception* are used, though not in a consistent fashion. We use the term *exception* to cover all these mechanisms, including the following:

I/O device request

Invoking an operating system service from a user program

Tracing instruction execution

Breakpoint (programmer-requested interrupt)

Integer arithmetic overflow

FP arithmetic anomaly (see Appendix A)

Page fault (not in main memory)

Misaligned memory accesses (if alignment is required)

Memory-protection violation

Using an undefined or unimplemented instruction

Hardware malfunctions

Power failure

When we wish to refer to some particular class of such exceptions, we will use a longer name, such as I/O interrupt, floating-point exception, or page fault. Figure 3.39 shows the variety of different names for the common exception events above.

Although we use the name *exception* to cover all of these events, individual events have important characteristics that determine what action is needed in the hardware.The requirements on exceptions can be characterized on five semi-independent axes:

Exception event	IBM 360	VAX	Motorola 680x0	Intel 80x86
I/O device request	Input/output interruption	Device interrupt	Exception (Level 0...7 autovector)	Vectored interrupt
Invoking the operating system service from a user program	Supervisor call interruption	Exception (change mode supervisor trap)	Exception (unimplemented instruction)—on Macintosh	Interrupt (INT instruction)
Tracing instruction execution	Not applicable	Exception (trace fault)	Exception (trace)	Interrupt (single-step trap)
Breakpoint	Not applicable	Exception (breakpoint fault)	Exception (illegal instruction or breakpoint)	Interrupt (breakpoint trap)
Integer arithmetic overflow or underflow; FP trap	Program interruption (overflow or underflow exception)	Exception (integer overflow trap or floating underflow fault)	Exception (floating-point coprocessor errors)	Interrupt (overflow trap or math unit exception)
Page fault (not in main memory)	Not applicable (only in 370)	Exception (translation not valid fault)	Exception (memory-management unit errors)	Interrupt (page fault)
Misaligned memory accesses	Program interruption (specification exception)	Not applicable	Exception (address error)	Not applicable
Memory protection violations	Program interruption (protection exception)	Exception (access control violation fault)	Exception (bus error)	Interrupt (protection exception)
Using undefined instructions	Program interruption (operation exception)	Exception (opcode privileged/reserved fault)	Exception (illegal instruction or breakpoint/unimplemented instruction)	Interrupt (invalid opcode)
Hardware malfunctions	Machine-check interruption	Exception (machine-check abort)	Exception (bus error)	Not applicable
Power failure	Machine-check interruption	Urgent interrupt	Not applicable	Nonmaskable interrupt

FIGURE 3.39 The names of common exceptions vary across four different architectures. Every event on the IBM 360 and 80x86 is called an *interrupt,* while every event on the 680x0 is called an *exception.* VAX divides events into *interrupts* or *exceptions.* Adjectives *device, software,* and *urgent* are used with VAX interrupts, while VAX exceptions are subdivided into *faults, traps,* and *aborts.*

1. *Synchronous versus asynchronous*—If the event occurs at the same place every time the program is executed with the same data and memory allocation, the event is *synchronous*. With the exception of hardware malfunctions, *asynchronous* events are caused by devices external to the processor and memory. Asynchronous events usually can be handled after the completion of the current instruction, which makes them easier to handle.

2. *User requested versus coerced*—If the user task directly asks for it, it is a *user-request* event. In some sense, user-requested exceptions are not really exceptions, since they are predictable. They are treated as exceptions, however, because the same mechanisms that are used to save and restore the state are used for these user-requested events. Because the only function of an instruction that triggers this exception is to cause the exception, user-requested exceptions can always be handled after the instruction has completed. *Coerced* exceptions are caused by some hardware event that is not under the control of the user program. Coerced exceptions are harder to implement because they are not predictable.

3. *User maskable versus user nonmaskable*—If an event can be masked or disabled by a user task, it is *user maskable*. This mask simply controls whether the hardware responds to the exception or not.

4. *Within versus between instructions*—This classification depends on whether the event prevents instruction completion by occurring in the middle of execution—no matter how short—or whether it is recognized *between* instructions. Exceptions that occur *within* instructions are usually synchronous, since the instruction triggers the exception. It's harder to implement exceptions that occur within instructions than those between instructions, since the instruction must be stopped and restarted. Asynchronous exceptions that occur within instructions arise from catastrophic situations (e.g., hardware malfunction) and always cause program termination.

5. *Resume versus terminate*—If the program's execution always stops after the interrupt, it is a *terminating* event. If the program's execution continues after the interrupt, it is a *resuming* event. It is easier to implement exceptions that terminate execution, since the machine need not be able to restart execution of the same program after handling the exception.

Figure 3.40 classifies the examples from Figure 3.39 according to these five categories. The difficult task is implementing interrupts occurring within instructions where the instruction must be resumed. Implementing such exceptions requires that another program must be invoked to save the state of the executing program, correct the cause of the exception, and then restore the state of the program before the instruction that caused the exception can be tried again. This process must be effectively invisible to the executing program. If a pipeline provides the ability for the machine to handle the exception, save the state, and restart without affecting the execution of the program, the pipeline or machine is said to be *restartable*. While early supercomputers and microprocessors often lacked this property, almost all machines today support it, at least for the integer pipeline, because it is needed to implement virtual memory (see Chapter 5).

Exception type	Synchronous vs. asynchronous	User request vs. coerced	User maskable vs. nonmaskable	Within vs. between instructions	Resume vs. terminate
I/O device request	Asynchronous	Coerced	Nonmaskable	Between	Resume
Invoke operating system	Synchronous	User request	Nonmaskable	Between	Resume
Tracing instruction execution	Synchronous	User request	User maskable	Between	Resume
Breakpoint	Synchronous	User request	User maskable	Between	Resume
Integer arithmetic overflow	Synchronous	Coerced	User maskable	Within	Resume
Floating-point arithmetic overflow or underflow	Synchronous	Coerced	User maskable	Within	Resume
Page fault	Synchronous	Coerced	Nonmaskable	Within	Resume
Misaligned memory accesses	Synchronous	Coerced	User maskable	Within	Resume
Memory-protection violations	Synchronous	Coerced	Nonmaskable	Within	Resume
Using undefined instructions	Synchronous	Coerced	Nonmaskable	Within	Terminate
Hardware malfunctions	Asynchronous	Coerced	Nonmaskable	Within	Terminate
Power failure	Asynchronous	Coerced	Nonmaskable	Within	Terminate

FIGURE 3.40 Five categories are used to define what actions are needed for the different exception types shown in Figure 3.39. Exceptions that must allow resumption are marked as resume, although the software may often choose to terminate the program. Synchronous, coerced exceptions occurring within instructions that can be resumed are the most difficult to implement. We might expect that memory protection access violations would always result in termination; however, modern operating systems use memory protection to detect events such as the first attempt to use a page or the first write to a page. Thus, processors should be able to resume after such exceptions.

Stopping and Restarting Execution

As in unpipelined implementations, the most difficult exceptions have two properties: (1) they occur within instructions (that is, in the middle of the instruction execution corresponding to EX or MEM pipe stages), and (2) they must be restartable. In our DLX pipeline, for example, a virtual memory page fault resulting from a data fetch cannot occur until sometime in the MEM stage of the instruction. By the time that fault is seen, several other instructions will be in execution. A page fault must be restartable and requires the intervention of another process, such as the operating system. Thus, the pipeline must be safely shut down and the state saved so that the instruction can be restarted in the correct state. Restarting is usually implemented by saving the PC of the instruction at which to restart. If the restarted instruction is not a branch, then we will continue to fetch the sequential successors and begin their execution in the normal fashion. If the restarted instruction is a branch, then we will reevaluate the branch condition and begin fetching from either the target or the fall through. When an exception occurs, the pipeline control can take the following steps to save the pipeline state safely:

1. Force a trap instruction into the pipeline on the next IF.
2. Until the trap is taken, turn off all writes for the faulting instruction and for all instructions that follow in the pipeline; this can be done by placing zeros into the pipeline latches of all instructions in the pipeline, starting with the instruction that generates the exception, but not those that precede that instruction. This prevents any state changes for instructions that will not be completed before the exception is handled.
3. After the exception-handling routine in the operating system receives control, it immediately saves the PC of the faulting instruction. This value will be used to return from the exception later.

When we use delayed branches, as mentioned in the last section, it is no longer possible to re-create the state of the machine with a single PC because the instructions in the pipeline may not be sequentially related. So we need to save and restore as many PCs as the length of the branch delay plus one. This is done in the third step above.

After the exception has been handled, special instructions return the machine from the exception by reloading the PCs and restarting the instruction stream (using the instruction RFE in DLX). If the pipeline can be stopped so that the instructions just before the faulting instruction are completed and those after it can be restarted from scratch, the pipeline is said to have *precise exceptions.* Ideally, the faulting instruction would not have changed the state, and correctly handling some exceptions requires that the faulting instruction have no effects. For other exceptions, such as floating-point exceptions, the faulting instruction on some machines writes its result before the exception can be handled. In such cases, the hardware must be prepared to retrieve the source operands, even if the destination is identical to one of the source operands. Because floating-point operations may run for many cycles, it is highly likely that some other instruction may have written the source operands (as we will see in the next section, floating-point operations often complete out of order). To overcome this, many recent high-performance machines have introduced two modes of operation. One mode has precise exceptions and the other (fast or performance mode) does not. Of course, the precise exception mode is slower, since it allows less overlap among floating-point instructions. In some high-performance machines, including Alpha 21064, Power-2, and MIPS R8000, the precise mode is often much slower (>10 times) and thus useful only for debugging of codes.

Supporting precise exceptions is a requirement in many systems, while in others it is "just" valuable because it simplifies the operating system interface. At a minimum, any machine with demand paging or IEEE arithmetic trap handlers must make its exceptions precise, either in the hardware or with some software support. For integer pipelines, the task of creating precise exceptions is easier, and accommodating virtual memory strongly motivates the support of precise

exceptions for memory references. In practice, these reasons have led designers and architects to always provide precise exceptions for the integer pipeline. In this section we describe how to implement precise exceptions for the DLX integer pipeline. We will describe techniques for handling the more complex challenges arising in the FP pipeline in section 3.7.

Exceptions in DLX

Figure 3.41 shows the DLX pipeline stages and which "problem" exceptions might occur in each stage. With pipelining, multiple exceptions may occur in the same clock cycle because there are multiple instructions in execution. For example, consider this instruction sequence:

LW	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

This pair of instructions can cause a data page fault and an arithmetic exception at the same time, since the LW is in the MEM stage while the ADD is in the EX stage. This case can be handled by dealing with only the data page fault and then restarting the execution. The second exception will reoccur (but not the first, if the software is correct), and when the second exception occurs, it can be handled independently.

In reality, the situation is not as straightforward as this simple example. Exceptions may occur out of order; that is, an instruction may cause an exception before an earlier instruction causes one. Consider again the above sequence of instructions, LW followed by ADD. The LW can get a data page fault, seen when the instruction is in MEM, and the ADD can get an instruction page fault, seen when

Pipeline stage	Problem exceptions occurring
IF	Page fault on instruction fetch; misaligned memory access; memory-protection violation
ID	Undefined or illegal opcode
EX	Arithmetic exception
MEM	Page fault on data fetch; misaligned memory access; memory-protection violation
WB	None

FIGURE 3.41 Exceptions that may occur in the DLX pipeline. Exceptions raised from instruction or data-memory access account for six out of eight cases.

the ADD instruction is in IF. The instruction page fault will actually occur first, even though it is caused by a later instruction!

Since we are implementing precise exceptions, the pipeline is required to handle the exception caused by the LW instruction first. To explain how this works, let's call the instruction in the position of the LW instruction *i*, and the instruction in the position of the ADD instruction $i + 1$. The pipeline cannot simply handle an exception when it occurs in time, since that will lead to exceptions occurring out of the unpipelined order. Instead, the hardware posts all exceptions caused by a given instruction in a status vector associated with that instruction. The exception status vector is carried along as the instruction goes down the pipeline. Once an exception indication is set in the exception status vector, any control signal that may cause a data value to be written is turned off (this includes both register writes and memory writes). Because a store can cause an exception during MEM, the hardware must be prepared to prevent the store from completing if it raises an exception.

When an instruction enters WB (or is about to leave MEM), the exception status vector is checked. If any exceptions are posted, they are handled in the order in which they would occur in time on an unpipelined machine—the exception corresponding to the earliest instruction (and usually the earliest pipe stage for that instruction) is handled first. This guarantees that all exceptions will be seen on instruction *i* before any are seen on $i + 1$. Of course, any action taken in earlier pipe stages on behalf of instruction *i* may be invalid, but since writes to the register file and memory were disabled, no state could have been changed. As we will see in section 3.7, maintaining this precise model for FP operations is much harder.

In the next subsection we describe problems that arise in implementing exceptions in the pipelines of machines with more powerful, longer-running instructions.

Instruction Set Complications

No DLX instruction has more than one result, and our DLX pipeline writes that result only at the end of an instruction's execution. When an instruction is guaranteed to complete it is called *committed*. In the DLX integer pipeline, all instructions are committed when they reach the end of the MEM stage (or beginning of WB) and no instruction updates the state before that stage. Thus, precise exceptions are straightforward. Some machines have instructions that change the state in the middle of the instruction execution, before the instruction and its predecessors are guaranteed to complete. For example, autoincrement addressing modes on the VAX cause the update of registers in the middle of an instruction execution. In such a case, if the instruction is aborted because of an exception, it will leave the machine state altered. Although we know which instruction caused the exception, without additional hardware support the exception will be imprecise because the instruction will be half finished. Restarting the instruction stream after such an imprecise exception is difficult. Alternatively, we could avoid updating the state before the instruction commits, but this may be difficult or costly,

since there may be dependences on the updated state: Consider a VAX instruction that autoincrements the same register multiple times. Thus, to maintain a precise exception model, most machines with such instructions have the ability to back out any state changes made before the instruction is committed. If an exception occurs, the machine uses this ability to reset the state of the machine to its value before the interrupted instruction started. In the next section, we will see that a more powerful DLX floating-point pipeline can introduce similar problems, and the next chapter introduces techniques that substantially complicate exception handling.

A related source of difficulties arises from instructions that update memory state during execution, such as the string copy operations on the VAX or 360. To make it possible to interrupt and restart these instructions, the instructions are defined to use the general-purpose registers as working registers. Thus the state of the partially completed instruction is always in the registers, which are saved on an exception and restored after the exception, allowing the instruction to continue. In the VAX an additional bit of state records when an instruction has started updating the memory state, so that when the pipeline is restarted, the machine knows whether to restart the instruction from the beginning or from the middle of the instruction. The 80x86 string instructions also use the registers as working storage, so that saving and restoring the registers saves and restores the state of such instructions.

A different set of difficulties arises from odd bits of state that may create additional pipeline hazards or may require extra hardware to save and restore. Condition codes are a good example of this. Many machines set the condition codes implicitly as part of the instruction. This approach has advantages, since condition codes decouple the evaluation of the condition from the actual branch. However, implicitly set condition codes can cause difficulties in scheduling any pipeline delays between setting the condition code and the branch, since most instructions set the condition code and cannot be used in the delay slots between the condition evaluation and the branch.

Additionally, in machines with condition codes, the processor must decide when the branch condition is fixed. This involves finding out when the condition code has been set for the last time before the branch. In most machines with implicitly set condition codes, this is done by delaying the branch condition evaluation until all previous instructions have had a chance to set the condition code.

Of course, architectures with explicitly set condition codes allow the delay between condition test and the branch to be scheduled; however, pipeline control must still track the last instruction that sets the condition code to know when the branch condition is decided. In effect, the condition code must be treated as an operand that requires hazard detection for RAW hazards with branches, just as DLX must do on the registers.

A final thorny area in pipelining is multicycle operations. Imagine trying to pipeline a sequence of VAX instructions such as this:

```
MOVL   R1,R2
ADDL3  42(R1),56(R1)+,@(R1)
SUBL2  R2,R3
MOVC3  @(R1)[R2],74(R2),R3
```

These instructions differ radically in the number of clock cycles they will require, from as low as one up to hundreds of clock cycles. They also require different numbers of data memory accesses, from zero to possibly hundreds. The data hazards are very complex and occur both between and within instructions. The simple solution of making all instructions execute for the same number of clock cycles is unacceptable, because it introduces an enormous number of hazards and bypass conditions and makes an immensely long pipeline. Pipelining the VAX at the instruction level is difficult, but a clever solution was found by the VAX 8800 designers. They pipeline the *microinstruction* execution: a microinstruction is a simple instruction used in sequences to implement a more complex instruction set. Because the microinstructions are simple (they look a lot like DLX), the pipeline control is much easier. While it is not clear that this approach can achieve quite as low a CPI as an instruction-level pipeline for the VAX, it is much simpler, possibly leading to a shorter clock cycle.

In comparison, load-store machines have simple operations with similar amounts of work and pipeline more easily. If architects realize the relationship between instruction set design and pipelining, they can design architectures for more efficient pipelining. In the next section we will see how the DLX pipeline deals with long-running instructions, specifically floating-point operations.

3.7 Extending the DLX Pipeline to Handle Multicycle Operations

We now want to explore how our DLX pipeline can be extended to handle floating-point operations. This section concentrates on the basic approach and the design alternatives, closing with some performance measurements of a DLX floating-point pipeline.

It is impractical to require that all DLX floating-point operations complete in one clock cycle, or even in two. Doing so would mean accepting a slow clock, or using enormous amounts of logic in the floating-point units, or both. Instead, the floating-point pipeline will allow for a longer latency for operations. This is easier to grasp if we imagine the floating-point instructions as having the same pipeline as the integer instructions, with two important changes. First, the EX cycle may be repeated as many times as needed to complete the operation—the number of repetitions can vary for different operations. Second, there may be multiple floating-point functional units. A stall will occur if the instruction to be issued will either cause a structural hazard for the functional unit it uses or cause a data hazard.

For this section, let's assume that there are four separate functional units in our DLX implementation:

1. The main integer unit that handles loads and stores, integer ALU operations, and branches.
2. FP and integer multiplier.
3. FP adder that handles FP add, subtract, and conversion.
4. FP and integer divider.

If we also assume that the execution stages of these functional units are not pipelined, then Figure 3.42 shows the resulting pipeline structure. Because EX is not pipelined, no other instruction using that functional unit may issue until the previous instruction leaves EX. Moreover, if an instruction cannot proceed to the EX stage, the entire pipeline behind that instruction will be stalled.

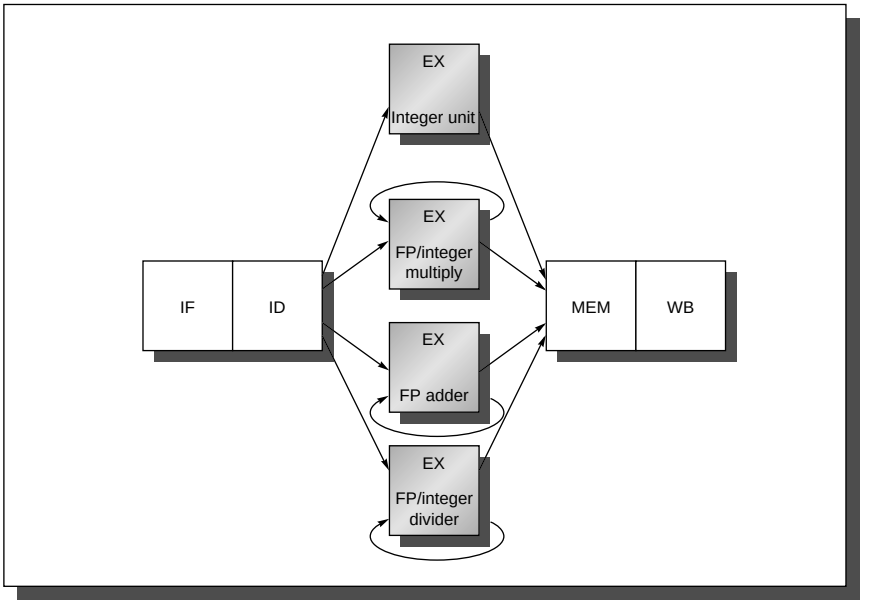

FIGURE 3.42 The DLX pipeline with three additional unpipelined, floating-point, functional units. Because only one instruction issues on every clock cycle, all instructions go through the standard pipeline for integer operations. The floating-point operations simply loop when they reach the EX stage. After they have finished the EX stage, they proceed to MEM and WB to complete execution.

In reality, the intermediate results are probably not cycled around the EX unit as Figure 3.42 suggests; instead, the EX pipeline stage has some number of clock delays larger than 1. We can generalize the structure of the FP pipeline shown in

Figure 3.42 to allow pipelining of some stages and multiple ongoing operations. To describe such a pipeline, we must define both the latency of the functional units and also the *initiation interval* or *repeat interval.* We define latency the same way we defined it earlier: the number of intervening cycles between an instruction that produces a result and an instruction that uses the result. The initiation or repeat interval is the number of cycles that must elapse between issuing two operations of a given type. For example, we will use the latencies and initiation intervals shown in Figure 3.43.

Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

FIGURE 3.43 Latencies and initiation intervals for functional units.

With this definition of latency, integer ALU operations have a latency of 0, since the results can be used on the next clock cycle, and loads have a latency of 1, since their results can be used after one intervening cycle. Since most operations consume their operands at the beginning of EX, the latency is usually the number of stages after EX that an instruction produces a result—for example, zero stages for ALU operations and one stage for loads. The primary exception is stores, which consume the value being stored one cycle later. Hence the latency to a store for the value being stored, but not for the base address register, will be one cycle less. Pipeline latency is essentially equal to one cycle less than the depth of the execution pipeline, which is the number of stages from the EX stage to the stage that produces the result. Thus, for the example pipeline just above, the number of stages in an FP add is four, while the number of stages in an FP multiply is seven. To achieve a higher clock rate, designers need to put fewer logic levels in each pipe stage, which makes the number of pipe stages required for more complex operations larger. The penalty for the faster clock rate is thus longer latency for operations.

The example pipeline structure in Figure 3.43 allows up to four outstanding FP adds, seven outstanding FP/integer multiplies, and one FP divide. Figure 3.44 shows how this pipeline can be drawn by extending Figure 3.42. The repeat interval is implemented in Figure 3.44 by adding additional pipeline stages, which will be separated by additional pipeline registers. Because the units are independent, we name the stages differently. The pipeline stages that take multiple clock cycles, such as the divide unit, are further subdivided to show the latency of those stages. Because they are not complete stages, only one operation may be active.

The pipeline structure can also be shown using the familiar diagrams from earlier in the chapter, as Figure 3.45 shows for a set of independent FP operations and FP loads and stores. Naturally, the longer latency of the FP operations increases the frequency of RAW hazards and resultant stalls, as we will see later in this section.

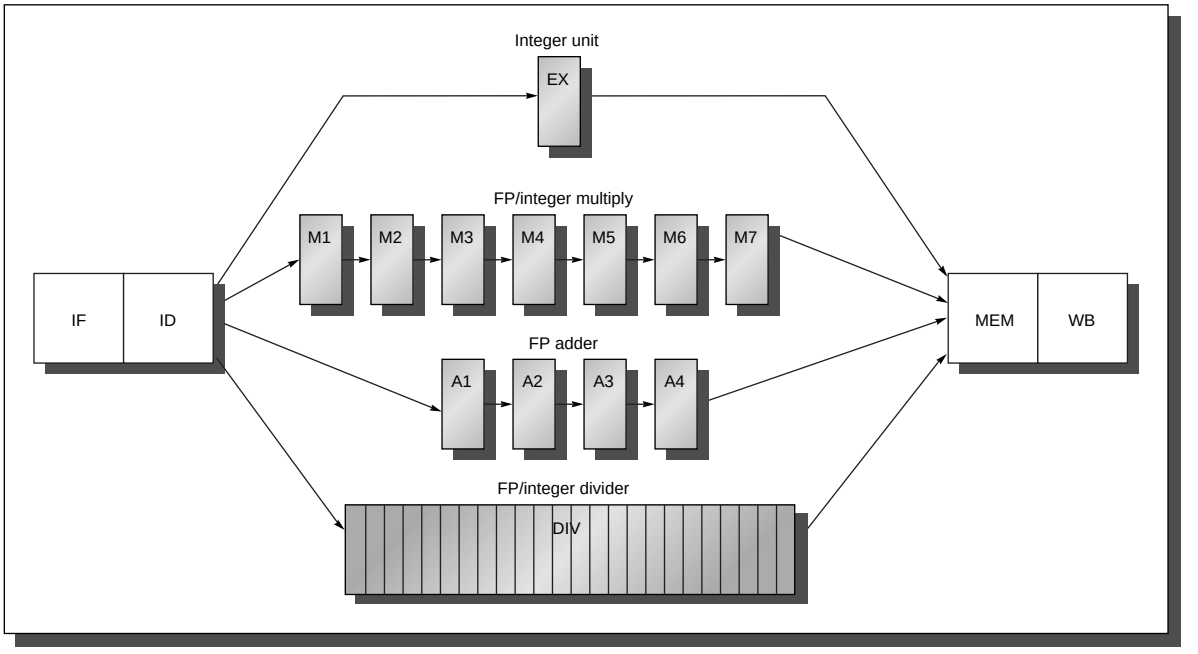

FIGURE 3.44 A pipeline that supports multiple outstanding FP operations. The FP multiplier and adder are fully pipelined and have a depth of seven and four stages, respectively. The FP divider is not pipelined, but requires 24 clock cycles to complete. The latency in instructions between the issue of an FP operation and the use of the result of that operation without incurring a RAW stall is determined by the number of cycles spent in the execution stages. For example, the fourth instruction after an FP add can use the result of the FP add. For integer ALU operations, the depth of the execution pipeline is always one and the next instruction can use the results. Both FP loads and integer loads complete during MEM, which means that the memory system must provide either 32 or 64 bits in a single clock.

MULTD	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
ADDD		IF	ID	*A1*	A2	A3	**A4**	MEM	WB		
LD			IF	ID	*EX*	**MEM**	WB				
SD				IF	ID	*EX*	*MEM*	WB			

FIGURE 3.45 The pipeline timing of a set of independent FP operations. The stages in italics show where data is needed, while the stages in bold show where a result is available. FP loads and stores use a 64-bit path to memory so that the pipelining timing is just like an integer load or store.

The structure of the pipeline in Figure 3.44 requires the introduction of the additional pipeline registers (e.g., A1/A2, A2/A3, A3/A4) and the modification of the connections to those registers. The ID/EX register must be expanded to connect ID to EX, DIV, M1, and A1; we can refer to the portion of the register associated with one of the next stages with the notation ID/EX, ID/DIV, ID/M1, or ID/A1. The pipeline register between ID and all the other stages may be thought of as logically separate registers and may, in fact, be implemented as separate registers. Because only one operation can be in a pipe stage at a time, the control information can be associated with the register at the head of the stage.

Hazards and Forwarding in Longer Latency Pipelines

There are a number of different aspects to the hazard detection and forwarding for a pipeline like that in Figure 3.44:

1. Because the divide unit is not fully pipelined, structural hazards can occur. These will need to be detected and issuing instructions will need to be stalled.
2. Because the instructions have varying running times, the number of register writes required in a cycle can be larger than 1.
3. WAW hazards are possible, since instructions no longer reach WB in order. Note that WAR hazards are not possible, since the register reads always occur in ID.
4. Instructions can complete in a different order than they were issued, causing problems with exceptions; we deal with this in the next subsection.
5. Because of longer latency of operations, stalls for RAW hazards will be more frequent.

The increase in stalls arising from longer operation latencies is fundamentally the same as that for the integer pipeline. Before describing the new problems that arise in this FP pipeline and looking at solutions, let's examine the potential impact of RAW hazards. Figure 3.46 shows a typical FP code sequence and the resultant stalls. At the end of this section, we'll examine the performance of this FP pipeline for our SPEC subset.

Now look at the problems arising from writes, described as (2) and (3) in the list above. If we assume the FP register file has one write port, sequences of FP operations, as well as an FP load together with FP operations, can cause conflicts for the register write port. Consider the pipeline sequence shown in Figure 3.47: In clock cycle 11, all three instructions will reach WB and want to write the register file. With only a single register file write port, the machine must serialize the instruction completion. This single register port represents a structural hazard. We could increase the number of write ports to solve this, but that solution may be unattractive since the additional write ports would be used only rarely. This is because the maximum steady state number of write ports needed is 1. Instead, we choose to detect and enforce access to the write port as a structural hazard.

	Clock cycle number															
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
LD F4,0 (R2)	IF	ID	EX	MEM	WB												
MULTD F0, F4,F6		IF	ID	stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
ADDD F2, F0,F8			IF	stall	ID	stall	stall	stall	stall	stall	stall	A1	A2	A3	A4	MEM	
SD 0(R2), F2					IF	stall	stall	stall	stall	stall	stall	ID	EX	stall	stall	stall	MEM

FIGURE 3.46 A typical FP code sequence showing the stalls arising from RAW hazards. The longer pipeline substantially raises the frequency of stalls versus the shallower integer pipeline. Each instruction in this sequence is dependent on the previous and proceeds as soon as data are available, which assumes the pipeline has full bypassing and forwarding. The SD must be stalled an extra cycle so that its MEM does not conflict with the ADDD. Extra hardware could easily handle this case.

	Clock cycle number										
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
MULTD F0,F4,F6	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
ADDD F2,F4,F6				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
LD F2,0(R2)							IF	ID	EX	MEM	WB

FIGURE 3.47 Three instructions want to perform a write back to the FP register file simultaneously, as shown in clock cycle 11. This is *not* the worst case, since an earlier divide in the FP unit could also finish on the same clock. Note that although the MULTD, ADDD, and LD all are in the MEM stage in clock cycle 10, only the LD actually uses the memory, so no structural hazard exists for MEM.

There are two different ways to implement this interlock. The first is to track the use of the write port in the ID stage and to stall an instruction before it issues, just as we would for any other structural hazard. Tracking the use of the write port can be done with a shift register that indicates when already-issued instructions will use the register file. If the instruction in ID needs to use the register file at the same time as an instruction already issued, the instruction in ID is stalled for a cycle. On each clock the reservation register is shifted one bit. This implementation has an advantage: It maintains the property that all interlock detection and stall insertion occurs in the ID stage. The cost is the addition of the shift register and write conflict logic. We will assume this scheme throughout this section.

An alternative scheme is to stall a conflicting instruction when it tries to enter either the MEM or WB stage. If we wait to stall the conflicting instructions until

they want to enter the MEM or WB stage, we can choose to stall either instruction. A simple, though sometimes suboptimal, heuristic is to give priority to the unit with the longest latency, since that is the one most likely to have caused another instruction to be stalled for a RAW hazard. The advantage of this scheme is that it does not require us to detect the conflict until the entrance of the MEM or WB stage, where it is easy to see. The disadvantage is that it complicates pipeline control, as stalls can now arise from two places. Notice that stalling before entering MEM will cause the EX, A4, or M7 stage to be occupied, possibly forcing the stall to trickle back in the pipeline. Likewise, stalling before WB would cause MEM to back up.

Our other problem is the possibility of WAW hazards. To see that these exist, consider the example in Figure 3.47. If the LD instruction were issued one cycle earlier and had a destination of F2, then it would create a WAW hazard, because it would write F2 one cycle earlier than the ADDD. Note that this hazard only occurs when the result of the ADDD is overwritten *without* any instruction ever using it! If there were a use of F2 between the ADDD and the LD, the pipeline would need to be stalled for a RAW hazard, and the LD would not issue until the ADDD was completed. We could argue that, for our pipeline, WAW hazards only occur when a useless instruction is executed, but we must still detect them and make sure that the result of the LD appears in F2 when we are done. (As we will see in section 3.10, such sequences sometimes *do* occur in reasonable code.)

There are two possible ways to handle this WAW hazard. The first approach is to delay the issue of the load instruction until the ADDD enters MEM. The second approach is to stamp out the result of the ADDD by detecting the hazard and changing the control so that the ADDD does not write its result. Then, the LD can issue right away. Because this hazard is rare, either scheme will work fine—you can pick whatever is simpler to implement. In either case, the hazard can be detected during ID when the LD is issuing. Then stalling the LD or making the ADDD a no-op is easy. The difficult situation is to detect that the LD might finish before the ADDD, because that requires knowing the length of the pipeline and the current position of the ADDD. Luckily, this code sequence (two writes with no intervening read) will be very rare, so we can use a simple solution: If an instruction in ID wants to write the same register as an instruction already issued, do not issue the instruction to EX. In the next chapter, we will see how additional hardware can eliminate stalls for such hazards. First, let's put together the pieces for implementing the hazard and issue logic in our FP pipeline.

In detecting the possible hazards, we must consider hazards among FP instructions, as well as hazards between an FP instruction and an integer instruction. Except for FP loads-stores and FP-integer register moves, the FP and integer registers are distinct. All integer instructions operate on the integer registers, while the floating-point operations operate only on their own registers. Thus, we need only consider FP loads-stores and FP register moves in detecting hazards between FP and integer instructions. This simplification of pipeline control is an additional advantage of having separate register files for integer and floating-point data. (The main advantages are a doubling of the number of registers, with-

out making either set larger, and an increase in bandwidth without adding more ports to either set. The main disadvantage, beyond the need for an extra register file, is the small cost of occasional moves needed between the two register sets.) Assuming that the pipeline does all hazard detection in ID, there are three checks that must be performed before an instruction can issue:

1. *Check for structural hazards*—Wait until the required functional unit is not busy (this is only needed for divides in this pipeline) and make sure the register write port is available when it will be needed.
2. *Check for a RAW data hazard*—Wait until the source registers are not listed as pending destinations in a pipeline register that will not be available when this instruction needs the result. A number of checks must be made here, depending on both the source instruction, which determines when the result will be available, and the destination instruction, which determines when the value is needed. For example, if the instruction in ID is an FP operation with source register F2, then F2 cannot be listed as a destination in ID/A1, A1/A2, or A2/A3, which correspond to FP add instructions that will not be finished when the instruction in ID needs a result. (ID/A1 is the portion of the output register of ID that is sent to A1.) Divide is somewhat more tricky, if we want to allow the last few cycles of a divide to be overlapped, since we need to handle the case when a divide is close to finishing as special. In practice, designers might ignore this optimization in favor of a simpler issue test.
3. *Check for a WAW data hazard*—Determine if any instruction in A1,..., A4, D, M1,..., M7 has the same register destination as this instruction. If so, stall the issue of the instruction in ID.

Although the hazard detection is more complex with the multicycle FP operations, the concepts are the same as for the DLX integer pipeline. The same is true for the forwarding logic. The forwarding can be implemented by checking if the destination register in any of EX/MEM, A4/MEM, M7/MEM, D/MEM, or MEM/WB registers is one of the source registers of a floating-point instruction. If so, the appropriate input multiplexer will have to be enabled so as to choose the forwarded data. In the Exercises, you will have the opportunity to specify the logic for the RAW and WAW hazard detection as well as for forwarding.

Multicycle FP operations also introduce problems for our exception mechanisms, which we deal with next.

Maintaining Precise Exceptions

Another problem caused by these long-running instructions can be illustrated with the following sequence of code:

```
DIVF    F0,F2,F4
ADDF    F10,F10,F8
```

```
SUBF    F12,F12,F14
```

This code sequence looks straightforward; there are no dependences. A problem arises, however, because an instruction issued early may complete after an instruction issued later. In this example, we can expect ADDF and SUBF to complete *before* the DIVF completes. This is called *out-of-order completion* and is common in pipelines with long-running operations. Because hazard detection will prevent any dependence among instructions from being violated, why is out-of-order completion a problem? Suppose that the SUBF causes a floating-point arithmetic exception at a point where the ADDF has completed but the DIVF has not. The result will be an imprecise exception, something we are trying to avoid. It may appear that this could be handled by letting the floating-point pipeline drain, as we do for the integer pipeline. But the exception may be in a position where this is not possible. For example, if the DIVF decided to take a floating-point-arithmetic exception after the add completed, we could not have a precise exception at the hardware level. In fact, because the ADDF destroys one of its operands, we could not restore the state to what it was before the DIVF, even with software help.

This problem arises because instructions are completing in a different order than they were issued. There are four possible approaches to dealing with out-of-order completion. The first is to ignore the problem and settle for imprecise exceptions. This approach was used in the 1960s and early 1970s. It is still used in some supercomputers, where certain classes of exceptions are not allowed or are handled by the hardware without stopping the pipeline. It is difficult to use this approach in most machines built today because of features such as virtual memory and the IEEE floating-point standard, which essentially require precise exceptions through a combination of hardware and software. As mentioned earlier, some recent machines have solved this problem by introducing two modes of execution: a fast, but possibly imprecise mode and a slower, precise mode. The slower precise mode is implemented either with a mode switch or by insertion of explicit instructions that test for FP exceptions. In either case the amount of overlap and reordering permitted in the FP pipeline is significantly restricted so that effectively only one FP instruction is active at a time. This solution is used in the DEC Alpha 21064 and 21164, in the IBM Power-1 and Power-2, and in the MIPS R8000.

A second approach is to buffer the results of an operation until all the operations that were issued earlier are complete. Some machines actually use this solution, but it becomes expensive when the difference in running times among operations is large, since the number of results to buffer can become large. Furthermore, results from the queue must be bypassed to continue issuing instructions while waiting for the longer instruction. This requires a large number of comparators and a very large multiplexer.

There are two viable variations on this basic approach. The first is a *history file*, used in the CYBER 180/990. The history file keeps track of the original values of registers. When an exception occurs and the state must be rolled back ear-

lier than some instruction that completed out of order, the original value of the register can be restored from the history file. A similar technique is used for autoincrement and autodecrement addressing on machines like VAXes. Another approach, the *future file*, proposed by J. Smith and A. Pleszkun [1988], keeps the newer value of a register; when all earlier instructions have completed, the main register file is updated from the future file. On an exception, the main register file has the precise values for the interrupted state. In the next chapter (section 4.6), we will see extensions of this idea, which are used in processors such as the PowerPC 620 and MIPS R10000 to allow overlap and reordering while preserving precise exceptions.

A third technique in use is to allow the exceptions to become somewhat imprecise, but to keep enough information so that the trap-handling routines can create a precise sequence for the exception. This means knowing what operations were in the pipeline and their PCs. Then, after handling the exception, the software finishes any instructions that precede the latest instruction completed, and the sequence can restart. Consider the following worst-case code sequence:

Instruction$_1$—A long-running instruction that eventually interrupts execution.

Instruction$_2$, ..., Instruction$_{n-1}$—A series of instructions that are not completed.

Instruction$_n$—An instruction that is finished.

Given the PCs of all the instructions in the pipeline and the exception return PC, the software can find the state of instruction$_1$ and instruction$_n$. Because instruction$_n$ has completed, we will want to restart execution at instruction$_{n+1}$. After handling the exception, the software must simulate the execution of instruction$_1$, ..., instruction$_{n-1}$. Then we can return from the exception and restart at instruction$_{n+1}$. The complexity of executing these instructions properly by the handler is the major difficulty of this scheme. There is an important simplification for simple DLX-like pipelines: If instruction$_2$, ..., instruction$_n$ are all integer instructions, then we know that if instruction$_n$ has completed, all of instruction$_2$, ..., instruction$_{n-1}$ have also completed. Thus, only floating-point operations need to be handled. To make this scheme tractable, the number of floating-point instructions that can be overlapped in execution can be limited. For example, if we only overlap two instructions, then only the interrupting instruction need be completed by software. This restriction may reduce the potential throughput if the FP pipelines are deep or if there is a significant number of FP functional units. This approach is used in the SPARC architecture to allow overlap of floating-point and integer operations.

The final technique is a hybrid scheme that allows the instruction issue to continue only if it is certain that all the instructions before the issuing instruction will complete without causing an exception. This guarantees that when an exception occurs, no instructions after the interrupting one will be completed and all of the instructions before the interrupting one can be completed. This sometimes means stalling the machine to maintain precise exceptions. To make this scheme work,

the floating-point functional units must determine if an exception is possible early in the EX stage (in the first three clock cycles in the DLX pipeline), so as to prevent further instructions from completing. This scheme is used in the MIPS R2000/3000, the R4000, and the Intel Pentium. It is discussed further in Appendix A.

Performance of a DLX FP Pipeline

The DLX FP pipeline of Figure 3.44 on page 190 can generate both structural stalls for the divide unit and stalls for RAW hazards (it also can have WAW hazards, but this rarely occurs in practice). Figure 3.48 shows the number of stall cycles for each type of floating-point operation on a per instance basis (i.e., the first bar for each FP benchmark shows the number of FP result stalls for each FP add, subtract, or compare). As we might expect, the stall cycles per operation track the latency of the FP operations, varying from 46% to 59% of the latency of the functional unit.

Figure 3.49 gives the complete breakdown of integer and floating-point stalls for the five FP SPEC benchmarks we are using. There are four classes of stalls shown: FP result stalls, FP compare stalls, load and branch delays, and floating-point structural delays. The compiler tries to schedule both load and FP delays before it schedules branch delays. The total number of stalls per instruction varies from 0.65 to 1.21.

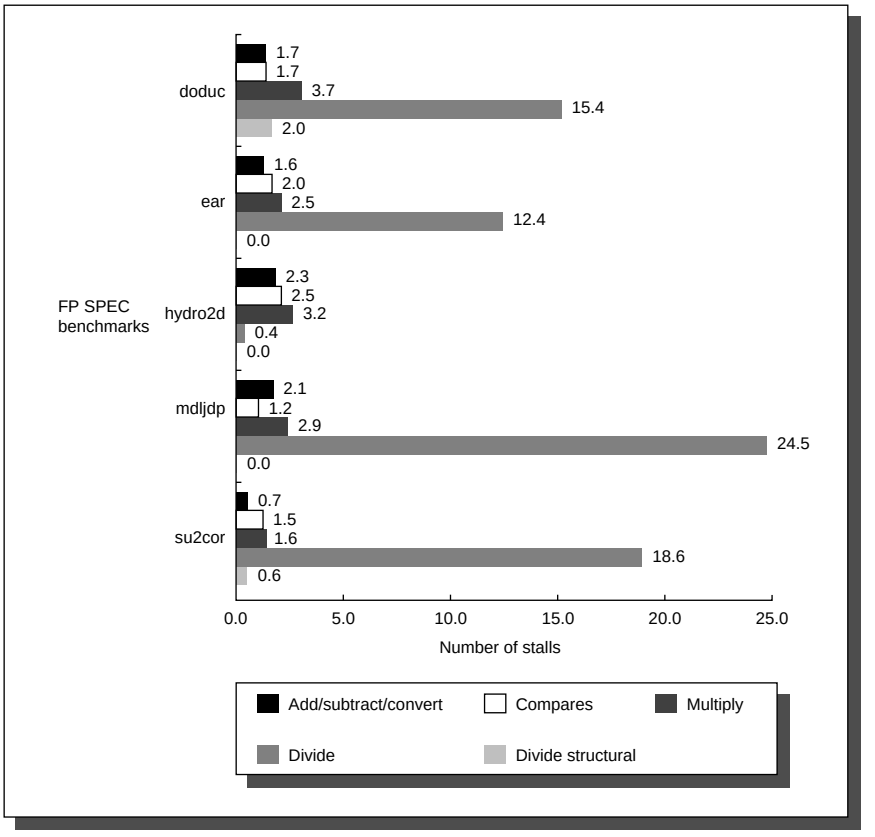

FIGURE 3.48 Stalls per FP operation for each major type of FP operation. Except for the divide structural hazards, these data do not depend on the frequency of an operation, only on its latency and the number of cycles before the result is used. The number of stalls from RAW hazards roughly tracks the latency of the FP unit. For example, the average number of stalls per FP add, subtract, or convert is 1.7 cycles, or 56% of the latency (3 cycles). Likewise, the average number of stalls for multiplies and divides are 2.8 and 14.2, respectively, or 46% and 59% of the corresponding latency. Structural hazards for divides are rare, since the divide frequency is low.

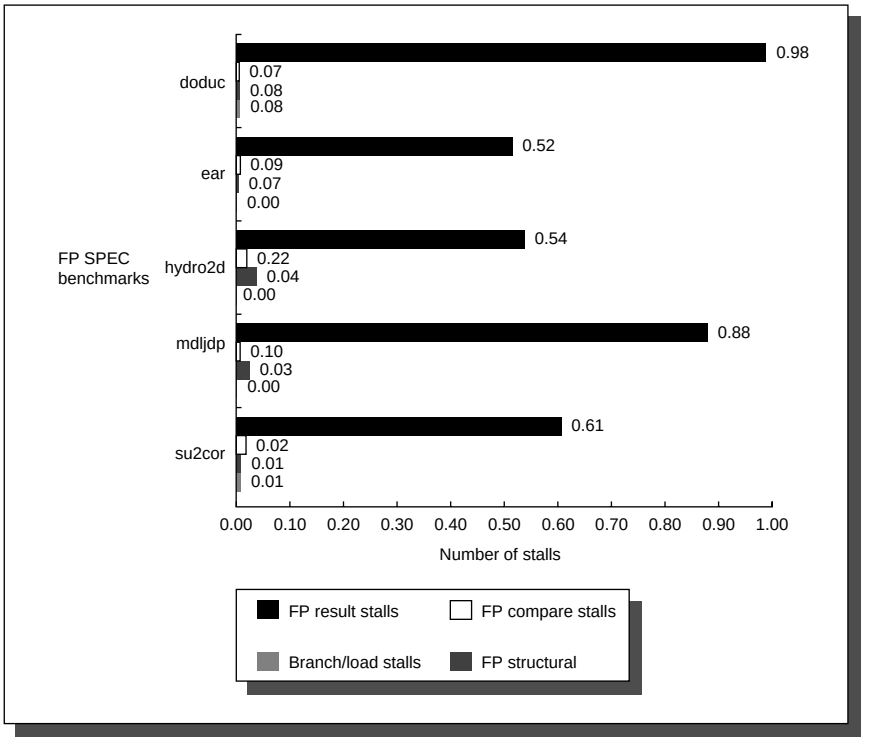

FIGURE 3.49 The stalls occurring for the DLX FP pipeline for the five FP SPEC benchmarks. The total number of stalls per instruction ranges from 0.65 for su2cor to 1.21 for doduc, with an average of 0.87. FP result stalls dominate in all cases, with an average of 0.71 stalls per instruction or 82% of the stalled cycles. Compares generate an average of 0.1 stalls per instruction and are the second largest source. The divide structural hazard is only significant for doduc.

3.8 Crosscutting Issues: Instruction Set Design and Pipelining

For many years the interaction between instruction sets and implementations was believed to be small, and implementation issues were not a major focus in designing instruction sets. In the 1980s it became clear that the difficulty and inefficiency of pipelining could both be increased by instruction set complications. Here are some examples, many of which are mentioned earlier in the chapter:

- Variable instruction lengths and running times can lead to imbalance among pipeline stages, causing other stages to back up. They also severely complicate hazard detection and the maintenance of precise exceptions. Of course, some-

times the advantages justify the added complexity. For example, caches cause instruction running times to vary when they miss; however, the performance advantages of caches make the added complexity acceptable. To minimize the complexity, most machines freeze the pipeline on a cache miss. Other machines try to continue running parts of the pipeline; though this is complex, it may overcome some of the performance losses from cache misses.

- Sophisticated addressing modes can lead to different sorts of problems. Addressing modes that update registers, such as post-autoincrement, complicate hazard detection. They also slightly increase the complexity of instruction restart. Other addressing modes that require multiple memory accesses substantially complicate pipeline control and make it difficult to keep the pipeline flowing smoothly.
- Architectures that allow writes into the instruction space (self-modifying code), such as the 80x86, can cause trouble for pipelining (as well as for cache designs). For example, if an instruction in the pipeline can modify another instruction, we must constantly check if the address being written by an instruction corresponds to the address of an instruction following the instruction that writes in the pipeline. If so, the pipeline must be flushed or the instruction in the pipeline somehow updated.
- Implicitly set condition codes increase the difficulty of finding when a branch has been decided and the difficulty of scheduling branch delays. The former problem occurs when the condition-code setting is not uniform, making it difficult to decide which instruction assigns the condition code last. The latter problem occurs when the condition code is unconditionally set by almost every instruction. This makes it hard to find instructions that can be scheduled between the condition evaluation and the branch. Most older architectures (the IBM 360, the DEC VAX, and the Intel 80x86, for example) have one or both of these problems. Many newer architectures avoid condition codes or set them explicitly under the control of a bit in the instruction. Either approach dramatically reduces pipelining difficulties.

As a simple example, suppose the DLX instruction format were more complex, so that a separate, decode pipe stage were required before register fetch. This would increase the branch delay to two clock cycles. At best, the second branch-delay slot would be wasted at least as often as the first. Gross [1983] found that a second delay slot was only used half as often as the first. This would lead to a performance penalty for the second delay slot of more than 0.1 clock cycles per instruction. Another example comes from a comparison of the pipeline efficiencies of a VAX 8800 and a MIPS R3000. Although these two machines have many similarities in organization, the VAX instruction set was not designed with pipelining in mind. As a result, on the SPEC89 benchmarks, the MIPS R3000 is faster by between two times and four times, with a mean performance advantage of 2.7 times.

3.9 Putting It All Together: The MIPS R4000 Pipeline

In this section we look at the pipeline structure and performance of the MIPS R4000 processor family. The MIPS-3 instruction set, which the R4000 implements, is a 64-bit instruction set similar to DLX. The R4000 uses a deeper pipeline than that of our DLX model both for integer and FP programs. This deeper pipeline allows it to achieve higher clock rates (100–200 MHz) by decomposing the five-stage integer pipeline into eight stages. Because cache access is particularly time critical, the extra pipeline stages come from decomposing the memory access. This type of deeper pipelining is sometimes called *superpipelining*.

Figure 3.50 shows the eight-stage pipeline structure using an abstracted version of the datapath. Figure 3.51 shows the overlap of successive instructions in the pipeline. Notice that although the instruction and data memory occupy multiple cycles, they are fully pipelined, so that a new instruction can start on every clock. In fact, the pipeline uses the data before the cache hit detection is complete; Chapter 5 discusses how this can be done in more detail.

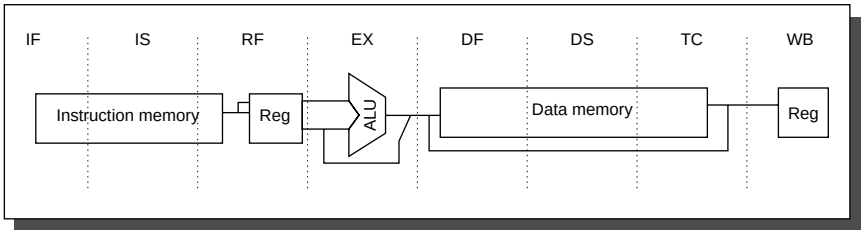

FIGURE 3.50 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating through RF. The TC stage is needed for data memory access, since we cannot write the data into the register until we know whether the cache access was a hit or not.

The function of each stage is as follows:

- IF—First half of instruction fetch; PC selection actually happens here, together with initiation of instruction cache access.
- IS—Second half of instruction fetch, complete instruction cache access.
- RF—Instruction decode and register fetch, hazard checking, and also instruction cache hit detection.

- EX—Execution, which includes effective address calculation, ALU operation, and branch target computation and condition evaluation.
- DF—Data fetch, first half of data cache access.
- DS—Second half of data fetch, completion of data cache access.
- TC—Tag check, determine whether the data cache access hit.
- WB—Write back for loads and register-register operations.

In addition to substantially increasing the amount of forwarding required, this longer latency pipeline increases both the load and branch delays. Figure 3.51 shows that load delays are two cycles, since the data value is available at the end of DS. Figure 3.52 shows the shorthand pipeline schedule when a use immediately follows a load. It shows that forwarding is required for the result of a load instruction to a destination that is three or four cycles later.

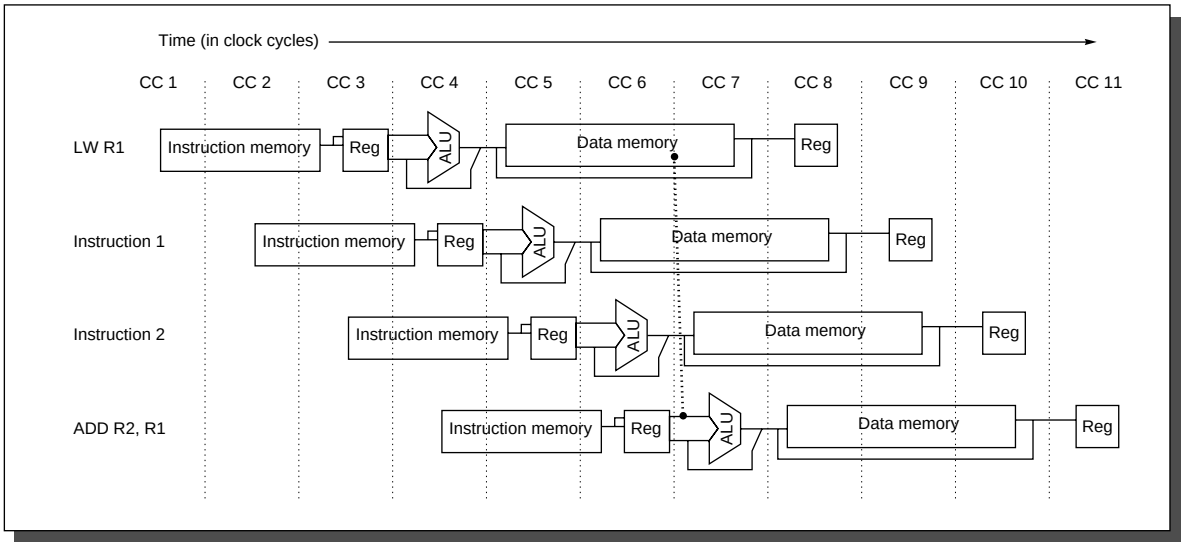

FIGURE 3.51 The structure of the R4000 integer pipeline leads to a two-cycle load delay. A two-cycle delay is possible because the data value is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up a cycle, when the correct data are available.

Figure 3.53 shows that the basic branch delay is three cycles, since the branch condition is computed during EX. The MIPS architecture has a single-cycle delayed branch. The R4000 uses a predict-not-taken strategy for the remaining two cycles of the branch delay. As Figure 3.54 shows, untaken branches are simply one-cycle delayed branches, while taken branches have a one-cycle delay slot

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
`LW R1, . . .`	IF	IS	RF	EX	DF	DS	TC	WB	
`ADD R2,R1, . . .`		IF	IS	RF	stall	stall	EX	DF	DS
`SUB R3,R1, . . .`			IF	IS	stall	stall	RF	EX	DF
`OR R4,R1, . . .`				IF	stall	stall	IS	RF	EX

FIGURE 3.52 A load instruction followed by an immediate use results in a two-cycle stall. Normal forwarding paths can be used after two cycles, so the ADD and SUB get the value by forwarding after the stall. The OR instruction gets the value from the register file. Since the two instructions after the load could be independent and hence not stall, the bypass can be to instructions that are three or four cycles after the load.

followed by two idle cycles. The instruction set provides a branch likely instruction, which we described earlier and which helps in filling the branch delay slot. Pipeline interlocks enforce both the two-cycle branch stall penalty on a taken branch and any data hazard stall that arises from use of a load result.

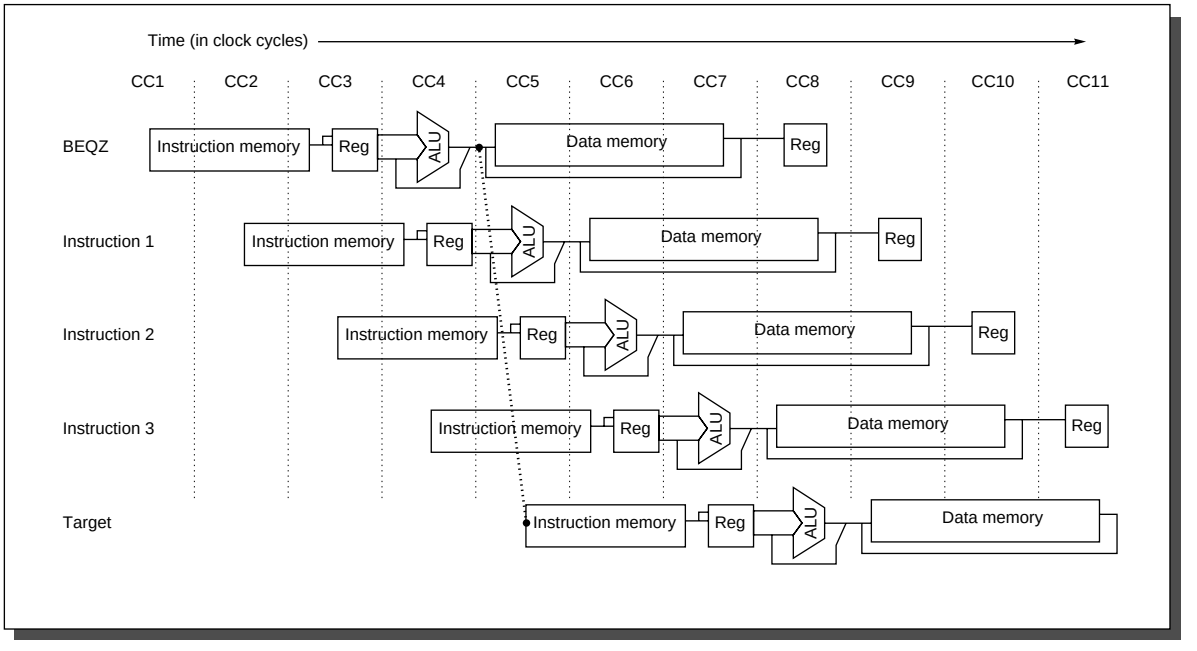

FIGURE 3.53 The basic branch delay is three cycles, since the condition evaluation is performed during EX.

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
Delay slot		IF	IS	RF	EX	DF	DS	TC	WB
Stall			stall	stall	stall	stall	stall	stall	stall
Stall				stall	stall	stall	stall	stall	stall
Branch target					IF	IS	RF	EX	DF

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Branch instruction	IF	IS	RF	EX	DF	DS	TC	WB	
Delay slot		IF	IS	RF	EX	DF	DS	TC	WB
Branch instruction + 2			IF	IS	RF	EX	DF	DS	TC
Branch instruction + 3				IF	IS	RF	EX	DF	DS

FIGURE 3.54 A taken branch, shown in the top portion of the figure, has a one-cycle delay slot followed by a two-cycle stall, while an untaken branch, shown in the bottom portion, has simply a one-cycle delay slot. The branch instruction can be an ordinary delayed branch or a branch-likely, which cancels the effect of the instruction in the delay slot if the branch is untaken.

In addition to the increase in stalls for loads and branches, the deeper pipeline increases the number of levels of forwarding for ALU operations. In our DLX five-stage pipeline, forwarding between two register-register ALU instructions could happen from the ALU/MEM or the MEM/WB registers. In the R4000 pipeline, there are four possible sources for an ALU bypass: EX/DF, DF/DS, DS/TC, and TC/WB. The Exercises ask you to explore all the possible forwarding conditions for the DLX instruction set using an R4000-style pipeline.

The Floating-Point Pipeline

The R4000 floating-point unit consists of three functional units: a floating-point divider, a floating-point multiplier, and a floating-point adder. As in the R3000, the adder logic is used on the final step of a multiply or divide. Double-precision FP operations can take from two cycles (for a negate) up to 112 cycles for a square root. In addition, the various units have different initiation rates. The floating-point functional unit can be thought of as having eight different stages, listed in Figure 3.55.

Stage	Functional unit	Description
A	FP adder	Mantissa ADD stage
D	FP divider	Divide pipeline stage
E	FP multiplier	Exception test stage
M	FP multiplier	First stage of multiplier
N	FP multiplier	Second stage of multiplier
R	FP adder	Rounding stage
S	FP adder	Operand shift stage
U		Unpack FP numbers

FIGURE 3.55 The eight stages used in the R4000 floating-point pipelines.

There is a single copy of each of these stages, and various instructions may use a stage zero or more times and in different orders. Figure 3.56 shows the latency, initiation rate, and pipeline stages used by the most common double-precision FP operations.

FP instruction	Latency	Initiation interval	Pipe stages
Add, subtract	4	3	U,S+A,A+R,R+S
Multiply	8	4	U,E+M,M,M,M,N,N+A,R
Divide	36	35	U,A,R,D^{27},D+A,D+R,D+A,D+R,A,R
Square root	112	111	U,E,$(A+R)^{108}$,A,R
Negate	2	1	U,S
Absolute value	2	1	U,S
FP compare	3	2	U,A,R

FIGURE 3.56 The latencies and initiation intervals for the FP operations both depend on the FP unit stages that a given operation must use. The latency values assume that the destination instruction is an FP operation; the latencies are one cycle less when the destination is a store. The pipe stages are shown in the order in which they are used for any operation. The notation S+A indicates a clock cycle in which both the S and A stages are used. The notation D^{28} indicates that the D stage is used 28 times in a row.

From the information in Figure 3.56, we can determine whether a sequence of different, independent FP operations can issue without stalling. If the timing of the sequence is such that a conflict occurs for a shared pipeline stage, then a stall will be needed. Figures 3.57, 3.58, 3.59, and 3.60 show four common possible two-instruction sequences: a multiply followed by an add, an add followed by a multiply, a divide followed by an add, and an add followed by a divide. The figures show all the interesting starting positions for the second instruction and

		Clock cycle												
Operation	**Issue/stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Multiply	Issue	U	M	M	M	M	N	**N+A**	**R**					
Add	Issue		U	S+A	A+R	R+S								
	Issue			U	S+A	A+R	R+S							
	Issue				U	S+A	A+R	R+S						
	Stall					U	S+A	**A+R**	**R+S**					
	Stall						U	**S+A**	**A+R**	R+S				
	Issue							U	S+A	A+R	R+S			
	Issue								U	S+A	A+R	R+S		

FIGURE 3.57 An FP multiply issued at clock 0 is followed by a *single* FP add issued between clocks 1 and 7. The second column indicates whether an instruction of the specified type stalls when it is issued *n* cycles later, where *n* is the clock cycle number in which the U stage of the second instruction occurs. The stage or stages that cause a stall are highlighted. Note that this table deals with only the interaction between the multiply and *one* add issued between clocks 1 and 7. In this case, the add will stall if it is issued four or five cycles after the multiply; otherwise, it issues without stalling. Notice that the add will be stalled for two cycles if it issues in cycle 4 since on the next clock cycle it will still conflict with the multiply; if, however, the add issues in cycle 5, it will stall for only one clock cycle, since that will eliminate the conflicts.

		Clock cycle												
Operation	**Issue/stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Add	Issue	U	S+A	A+R	R+S									
Multiply	Issue		U	M	M	M	M	N	N+A	R				
	Issue			U	M	M	M	M	N	N+A	R			

FIGURE 3.58 A multiply issuing after an add can always proceed without stalling, since the shorter instruction clears the shared pipeline stages before the longer instruction reaches them.

whether that second instruction will issue or stall for each position. Of course, there could be three instructions active, in which case the possibilities for stalls are much higher and the figures more complex.

		Clock cycle											
Operation	**Issue/stall**	**25**	**26**	**27**	**28**	**29**	**30**	**31**	**32**	**33**	**34**	**35**	**36**
Divide	issued in cycle 0...	D	D	D	D	D	**D+A**	**D+R**	**D+A**	**D+R**	**A**	**R**	
Add	Issue		U	S+A	A+R	R+S							
	Issue			U	S+A	A+R	R+S						
	Stall				U	S+A	**A+R**	**R+S**					
	Stall					U	**S+A**	**A+R**	R+S				
	Stall						U	S+A	**A+R**	**R+S**			
	Stall							U	**S+A**	**A+R**	R+S		
	Stall								U	S+A	**A+R**	**R+S**	
	Stall									U	**S+A**	**A+R**	R+S
	Issue										U	S+A	A+R
	Issue											U	S+A
	Issue												U

FIGURE 3.59 An FP divide can cause a stall for an add that starts near the end of the divide. The divide starts at cycle 0 and completes at cycle 35; the last 10 cycles of the divide are shown. Since the divide makes heavy use of the rounding hardware needed by the add, it stalls an add that starts in any of cycles 28 to 33. Notice the add starting in cycle 28 will be stalled until cycle 34. If the add started right after the divide it would not conflict, since the add could complete before the divide needed the shared stages, just as we saw in Figure 3.58 for a multiply and add. As in the earlier figure, this example assumes *exactly* one add that reaches the U stage between clock cycles 26 and 35.

		Clock cycle												
Operation	**Issue/stall**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
Add	Issue	U	S+A	**A+R**	**R+S**									
Divide	Stall		U	**A**	**R**	D	D	D	D	D	D	D	D	D
	Issue			U	A	R	D	D	D	D	D	D	D	D
	Issue				U	A	R	D	D	D	D	D	D	D

FIGURE 3.60 A double-precision add is followed by a double-precision divide. If the divide starts one cycle after the add, the divide stalls, but after that there is no conflict.

Performance of the R4000 Pipeline

In this section we examine the stalls that occur for the SPEC92 benchmarks when running on the R4000 pipeline structure. There are four major causes of pipeline stalls or losses:

1. Load stalls—Delays arising from the use of a load result one or two cycles after the load.

2. Branch stalls—Two-cycle stall on every taken branch plus unfilled or cancelled branch delay slots.
3. FP result stalls—Stalls because of RAW hazards for an FP operand.
4. FP structural stalls—Delays because of issue restrictions arising from conflicts for functional units in the FP pipeline.

Figure 3.61 shows the pipeline CPI breakdown for the R4000 pipeline for the 10 SPEC92 benchmarks. Figure 3.62 shows the same data but in tabular form.

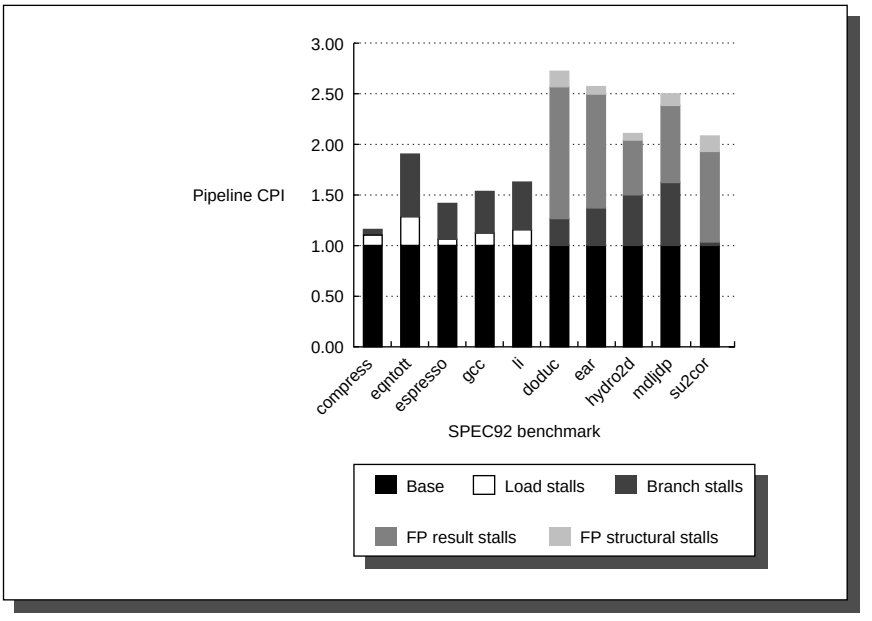

FIGURE 3.61 The pipeline CPI for 10 of the SPEC92 benchmarks, assuming a perfect cache. The pipeline CPI varies from 1.2 to 2.8. The leftmost five programs are integer programs, and branch delays are the major CPI contributor for these. The rightmost five programs are FP, and FP result stalls are the major contributor for these.

Benchmark	Pipeline CPI	Load stalls	Branch stalls	FP result stalls	FP structural stalls
compress	1.20	0.14	0.06	0.00	0.00
eqntott	1.88	0.27	0.61	0.00	0.00
espresso	1.42	0.07	0.35	0.00	0.00
gcc	1.56	0.13	0.43	0.00	0.00
li	1.64	0.18	0.46	0.00	0.00
Integer average	1.54	0.16	0.38	0.00	0.00
doduc	2.84	0.01	0.22	1.39	0.22
mdljdp2	2.66	0.01	0.31	1.20	0.15
ear	2.17	0.00	0.46	0.59	0.12
hydro2d	2.53	0.00	0.62	0.75	0.17
su2cor	2.18	0.02	0.07	0.84	0.26
FP average	2.48	0.01	0.33	0.95	0.18
Overall average	2.00	0.10	0.36	0.46	0.09

FIGURE 3.62 The total pipeline CPI and the contributions of the four major sources of stalls are shown. The major contributors are FP result stalls (both for branches and for FP inputs) and branch stalls, with loads and FP structural stalls adding less.

From the data in Figures 3.61 and 3.62, we can see the penalty of the deeper pipelining. The R4000's pipeline has much longer branch delays than the five-stage DLX-style pipeline. The longer branch delay substantially increases the cycles spent on branches, especially for the integer programs with a higher branch frequency. An interesting effect for the FP programs is that the latency of the FP functional units leads to more stalls than the structural hazards, which arise both from the initiation interval limitations and from conflicts for functional units from different FP instructions. Thus, reducing the latency of FP operations should be the first target, rather than more pipelining or replication of the functional units. Of course, reducing the latency would probably increase the structural stalls, since many potential structural stalls are hidden behind data hazards.

3.10 Fallacies and Pitfalls

Pitfall: Unexpected execution sequences may cause unexpected hazards.

At first glance, WAW hazards look like they should never occur because no compiler would ever generate two writes to the same register without an intervening read. But they can occur when the sequence is unexpected. For example, the first write might be in the delay slot of a taken branch when the scheduler thought the branch would not be taken. Here is the code sequence that could cause this:

```
        BNEZ   R1,foo
        DIVD   F0,F2,F4 ; moved into delay slot
               ; from fall through
        .....
        .....
foo:    LD     F0,qrs
```

If the branch is taken, then before the DIVD can complete, the LD will reach WB, causing a WAW hazard. The hardware must detect this and may stall the issue of the LD. Another way this can happen is if the second write is in a trap routine. This occurs when an instruction that traps and is writing results continues and completes after an instruction that writes the same register in the trap handler. The hardware must detect and prevent this as well.

Pitfall: Extensive pipelining can impact other aspects of a design, leading to overall worse cost/performance.

The best example of this phenomenon comes from two implementations of the VAX, the 8600 and the 8700. When the 8600 was initially delivered, it had a cycle time of 80 ns. Subsequently, a redesigned version, called the 8650, with a 55-ns clock was introduced. The 8700 has a much simpler pipeline that operates at the microinstruction level, yielding a smaller CPU with a faster clock cycle of 45 ns. The overall outcome is that the 8650 has a CPI advantage of about 20%, but the 8700 has a clock rate that is about 20% faster. Thus, the 8700 achieves the same performance with much less hardware.

Fallacy: Increasing the number of pipeline stages always increases performance.

Two factors combine to limit the performance improvement gained by pipelining. Limited parallelism in the instruction stream means that increasing the number of pipeline stages, called the pipeline depth, will eventually increase the CPI, due to dependences that require stalls. Second, clock skew and latch overhead combine to limit the decrease in clock period obtained by further pipelining. Figure 3.63 shows the trade-off between the number of pipeline stages and performance for the first 14 of the Livermore Loops. The performance flattens out when the number of pipeline stages reaches 4 and actually drops when the execution portion is pipelined 16 deep. Although this study is limited to a small set of FP programs, the trade-off of increasing CPI versus increasing clock rate by more pipelining arises constantly.

Pitfall: Evaluating a compile-time scheduler on the basis of unoptimized code.

Unoptimized code—containing redundant loads, stores, and other operations that might be eliminated by an optimizer—is much easier to schedule than “tight” optimized code. This holds for scheduling both control delays (with delayed

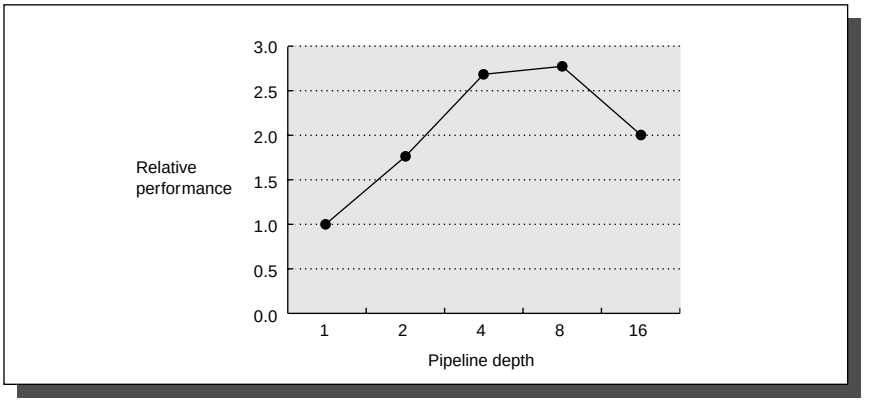

FIGURE 3.63 The depth of pipelining versus the speedup obtained. The x-axis shows the number of stages in the EX portion of the floating-point pipeline. A single-stage pipeline corresponds to 32 levels of logic, which might be appropriate for a single FP operation. Data based on Table 2 in Kunkel and Smith [1986].

branches) and delays arising from RAW hazards. In gcc running on an R3000, which has a pipeline almost identical to that of DLX, the frequency of idle clock cycles increases by 18% from the unoptimized and scheduled code to the optimized and scheduled code. Of course, the optimized program is much faster, since it has fewer instructions. To fairly evaluate a scheduler you must use optimized code, since in the real system you will derive good performance from other optimizations in addition to scheduling.

3.11 Concluding Remarks

Pipelining has been and is likely to continue to be one of the most important techniques for enhancing the performance of processors. Improving performance via pipelining was the key focus of many early computer designers in the late 1950s through the mid 1960s. In the late 1960s through the late 1970s, the attention of computer architects was focused on other things, including the dramatic improvements in cost, size, and reliability that were achieved by the introduction of integrated circuit technology. In this period pipelining played a secondary role in many designs. Since pipelining was not a primary focus, many instruction sets designed in this period made pipelining overly difficult and reduced its payoff. The VAX architecture is perhaps the best example.

In the late 1970s and early 1980s several researchers realized that instruction set complexity and implementation ease, particularly ease of pipelining, were related. The RISC movement led to a dramatic simplification in instruction sets that allowed rapid progress in the development of pipelining techniques. As we will

see in the next chapter, these techniques have become extremely sophisticated. The sophisticated implementation techniques now in use in many designs would have been extremely difficult with the more complex architectures of the 1970s.

In this chapter, we introduced the basic ideas in pipelining and looked at some simple compiler strategies for enhancing performance. The pipelined microprocessors of the 1980s relied on these strategies, with the R4000-style machine representing one of the most advanced of the "simple" pipeline organizations. To further improve performance in this decade most microprocessors have introduced schemes such as hardware-based pipeline scheduling, dynamic branch prediction, the ability to issue more than one instruction in a cycle, and the use of more powerful compiler technology. These more advanced techniques are the subject of the next chapter.

3.12 Historical Perspective and References

This section describes some of the major advances in pipelining and ends with some of the recent literature on high-performance pipelining.

The first general-purpose pipelined machine is considered to be Stretch, the IBM 7030. Stretch followed the IBM 704 and had a goal of being 100 times faster than the 704. The goal was a stretch from the state of the art at that time—hence the nickname. The plan was to obtain a factor of 1.6 from overlapping fetch, decode, and execute, using a four-stage pipeline. Bloch [1959] and Bucholtz [1962] describe the design and engineering trade-offs, including the use of ALU bypasses. The CDC 6600, developed in the early 1960s, also introduced several enhancements in pipelining; these innovations and the history of that design are discussed in the next chapter.

A series of general pipelining descriptions that appeared in the late 1970s and early 1980s provided most of the terminology and described most of the basic techniques used in simple pipelines. These surveys include Keller [1975], Ramamoorthy and Li [1977], Chen [1980], and Kogge's book [1981], devoted entirely to pipelining. Davidson and his colleagues [1971, 1975] developed the concept of pipeline reservation tables as a design methodology for multicycle pipelines with feedback (also described in Kogge [1981]). Many designers use a variation of these concepts, as we did in sections 3.2 and 3.3.

The RISC machines were originally designed with ease of implementation and pipelining in mind. Several of the early RISC papers, published in the early 1980s, attempt to quantify the performance advantages of the simplification in instruction set. The best analysis, however, is a comparison of a VAX and a MIPS implementation published by Bhandarkar and Clark in 1991, 10 years after the first published RISC papers. After 10 years of arguments about the implementation benefits of RISC, this paper convinced even the most skeptical designers of the advantages of a RISC instruction set architecture.

The RISC machines refined the notion of compiler-scheduled pipelines in the early 1980s, though earlier work on this topic is described at the end of the next chapter. The concepts of delayed branches and delayed loads—common in microprogramming—were extended into the high-level architecture. The Stanford MIPS architecture made the pipeline structure purposely visible to the compiler and allowed multiple operations per instruction. Simple schemes for scheduling the pipeline in the compiler were described by Sites [1979] for the Cray, by Hennessy and Gross [1983] (and in Gross's thesis [1983]), and by Gibbons and Muchnik [1986]. More advanced techniques will be described in the next chapter. Rymarczyk [1982] describes the interlock conditions that programmers should be aware of for a 360-like machine; this paper also shows the complex interaction between pipelining and an instruction set not designed to be pipelined. Static branch prediction by profiling has been explored by McFarling and Hennessy [1986] and by Fisher and Freudenberger [1992].

J. E. Smith and his colleagues have written a number of papers examining instruction issue, exception handling, and pipeline depth for high-speed scalar machines. Kunkel and Smith [1986] evaluate the impact of pipeline overhead and dependences on the choice of optimal pipeline depth; they also have an excellent discussion of latch design and its impact on pipelining. Smith and Pleszkun [1988] evaluate a variety of techniques for preserving precise exceptions. Weiss and Smith [1984] evaluate a variety of hardware pipeline scheduling and instruction-issue techniques.

The MIPS R4000, in addition to being one of the first deeply pipelined microprocessors, was the first true 64-bit architecture. It is described by Killian [1991] and by Heinrich [1993]. The initial Alpha implementation (the 21064) has a similar instruction set and similar integer pipeline structure, with more pipelining in the floating-point unit.

References

BHANDARKAR, D. AND D. W. CLARK [1991]. "Performance from architecture: Comparing a RISC and a CISC with similar hardware organizations," *Proc. Fourth Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (April), Palo Alto, Calif., 310–319.

BLOCH, E. [1959]. "The engineering design of the Stretch computer," *Proc. Fall Joint Computer Conf.*, 48–59.

BUCHOLTZ, W. [1962]. *Planning a Computer System: Project Stretch*, McGraw-Hill, New York.

CHEN, T. C. [1980]. "Overlap and parallel processing," in *Introduction to Computer Architecture*, H. Stone, ed., Science Research Associates, Chicago, 427–486.

CLARK, D. W. [1987]. "Pipelining and performance in the VAX 8800 processor," *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 173–177.

DAVIDSON, E. S. [1971]. "The design and control of pipelined function generators," *Proc. Conf. on Systems, Networks, and Computers*, IEEE (January), Oaxtepec, Mexico, 19–21.

DAVIDSON, E. S., A. T. THOMAS, L. E. SHAR, AND J. H. PATEL [1975]. "Effective control for pipelined processors," *COMPCON, IEEE* (March), San Francisco, 181–184.

EARLE, J. G. [1965]. "Latched carry-save adder," *IBM Technical Disclosure Bull.* 7 (March), 909–910.

EMER, J. S. AND D. W. CLARK [1984]. "A characterization of processor performance in the VAX-11/780," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 301–310.

FISHER, J. AND FREUDENBERGER, S. [1992]. "Predicting conditional branch directions from previous runs of a program," *Proc. Fifth Conf. on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (October), Boston, 85–95.

GIBBONS, P. B. AND S. S. MUCHNIK [1986]. "Efficient instruction scheduling for a pipelined processor," *SIGPLAN '86 Symposium on Compiler Construction*, ACM (June), Palo Alto, Calif., 11–16.

GROSS, T. R. [1983]. *Code Optimization of Pipeline Constraints*, Ph.D. Thesis (December), Computer Systems Lab., Stanford Univ.

HEINRICH, J. [1993]. *MIPS R4000 User's Manual*, Prentice Hall, Englewood Cliffs, N.J.

HENNESSY, J. L. AND T. R. GROSS [1983]. "Postpass code optimization of pipeline constraints," *ACM Trans. on Programming Languages and Systems* 5:3 (July), 422–448.

IBM [1990]. "The IBM RISC System/6000 processor" (collection of papers), *IBM J. of Research and Development* 34:1 (January).

KELLER R. M. [1975]. "Look-ahead processors," *ACM Computing Surveys* 7:4 (December), 177–195.

KILLIAN, E. [1991]. "MIPS R4000 technical overview–64 bits/100 MHz or bust," *Hot Chips III Symposium Record* (August), Stanford University, 1.6–1.19.

KOGGE, P. M. [1981]. *The Architecture of Pipelined Computers*, McGraw-Hill, New York.

KUNKEL, S. R. AND J. E. SMITH [1986]. "Optimal pipelining in supercomputers," *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 404–414.

MCFARLING, S. AND J. L. HENNESSY [1986]. "Reducing the cost of branches," *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 396-403.

RAMAMOORTHY, C. V. AND H. F. LI [1977]. "Pipeline architecture," *ACM Computing Surveys* 9:1 (March), 61–102.

RYMARCZYK, J. [1982]. "Coding guidelines for pipelined processors," *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems*, IEEE/ACM (March), Palo Alto, Calif., 12–19.

SITES, R. [1979]. *Instruction Ordering for the CRAY-1 Computer*, Tech. Rep. 78-CS-023 (July), Dept. of Computer Science, Univ. of Calif., San Diego.

SMITH, J. E. AND A. R. PLESZKUN [1988]. "Implementing precise interrupts in pipelined processors," *IEEE Trans. on Computers* 37:5 (May), 562–573.

WEISS, S. AND J. E. SMITH [1984]. "Instruction issue logic for pipelined supercomputers," *Proc. 11th Symposium on Computer Architecture* (June), Ann Arbor, Mich., 110–118.

EXERCISES

3.1 [15/15/15] <3.4,3.5> Use the following code fragment:

```
loop:   LW      R1,0(R2)
        ADDI    R1,R1,#1
        SW      0(R2),R1
        ADDI    R2,R2,#4
        SUB     R4,R3,R2
        BNEZ    R4,Loop
```

Assume that the initial value of R3 is R2 + 396.

Throughout this exercise use the DLX integer pipeline and assume all memory accesses are cache hits.

a. [15] <3.4,3.5> Show the timing of this instruction sequence for the DLX pipeline *without* any forwarding or bypassing hardware but assuming a register read and a write in the same clock cycle "forwards" through the register file, as in Figure 3.10. Use a pipeline timing chart like Figure 3.14 or 3.15. Assume that the branch is handled by flushing the pipeline. If all memory references hit in the cache, how many cycles does this loop take to execute?

b. [15] <3.4,3.5> Show the timing of this instruction sequence for the DLX pipeline with normal forwarding and bypassing hardware. Use a pipeline timing chart like Figure 3.14 or 3.15. Assume that the branch is handled by predicting it as not taken. If all memory references hit in the cache, how many cycles does this loop take to execute?

c. [15] <3.4,3.5> Assuming the DLX pipeline with a single-cycle delayed branch and normal forwarding and bypassing hardware, schedule the instructions in the loop including the branch-delay slot. You may reorder instructions and modify the individual instruction operands, but do not undertake other loop transformations that change the number or opcode of the instructions in the loop (that's for the next chapter!). Show a pipeline timing diagram and compute the number of cycles needed to execute the entire loop.

3.2 [15/15/15] <3.4,3.5,3.7> Use the following code fragment:

```
Loop:   LD      F0,0(R2)
        LD      F4,0(R3)
        MULTD   F0,F0,F4
        ADDD    F2,F0,F2
        ADDI    R2,R2,#8
        ADDI    R3,R3,#8
        SUB     R5,R4,R2
        BNEZ    R5,Loop
```

Assume that the initial value of R4 is R2 + 792.

For this exercise assume the standard DLX integer pipeline (as shown in Figure 3.10) and the standard DLX FP pipeline as described in Figures 3.43 and 3.44. If structural hazards are due to write-back contention, assume the earliest instruction gets priority and other instructions are stalled.

a. [15] <3.4,3.5,3.7> Show the timing of this instruction sequence for the DLX FP pipeline *without* any forwarding or bypassing hardware but assuming a register read and a write in the same clock cycle "forwards" through the register file, as in Figure 3.10. Use a pipeline timing chart like Figure 3.14 or 3.15. Assume that the branch is handled by flushing the pipeline. If all memory references hit in the cache, how many cycles does this loop take to execute?

b. [15] <3.4,3.5,3.7> Show the timing of this instruction sequence for the DLX FP pipeline with normal forwarding and bypassing hardware. Use a pipeline timing chart like Figure 3.14 or 3.15. Assume that the branch is handled by predicting it as not taken. If all memory references hit in the cache, how many cycles does this loop take to execute?

c. [15] <3.4,3.5,3.7> Assuming the DLX FP pipeline with a single-cycle delayed branch and full bypassing and forwarding hardware, schedule the instructions in the loop including the branch-delay slot. You may reorder instructions and modify the individual instruction operands, but do not undertake other loop transformations that change the number or opcode of the instructions in the loop (that's for the next chapter!). Show a pipeline timing diagram and compute the time needed in cycles to execute the entire loop.

3.3 [12/13/20/20/15/15] <3.2,3.4,3.5> For these problems, we will explore a pipeline for a register-memory architecture. The architecture has two instruction formats: a register-register format and a register-memory format. There is a single-memory addressing mode (offset + base register).

There is a set of ALU operations with format:

```
ALUop Rdest, Rsrc1, Rsrc2
```

or

```
ALUop Rdest, Rsrc1, MEM
```

where the ALUop is one of the following: Add, Subtract, And, Or, Load ($Rsrc_1$ ignored), Store. Rsrc or Rdest are registers. MEM is a base register and offset pair.

Branches use a full compare of two registers and are PC-relative. Assume that this machine is pipelined so that a new instruction is started every clock cycle. The following pipeline structure—similar to that used in the VAX 8700 micropipeline (Clark [1987])—is

IF	RF	ALU1	MEM	ALU2	WB					
	IF	RF	ALU1	MEM	ALU2	WB				
		IF	RF	ALU1	MEM	ALU2	WB			
			IF	RF	ALU1	MEM	ALU2	WB		
				IF	RF	ALU1	MEM	ALU2	WB	
					IF	RF	ALU1	MEM	ALU2	WB

The first ALU stage is used for effective address calculation for memory references and branches. The second ALU cycle is used for operations and branch comparison. RF is both a decode and register-fetch cycle. Assume that when a register read and a register write of the same register occur in the same clock the write data is forwarded.

a. [12] <3.2> Find the number of adders needed, counting any adder or incrementer; show a combination of instructions and pipe stages that justify this answer. You need only give one combination that maximizes the adder count.

b. [13] <3.2> Find the number of register read and write ports and memory read and write ports required. Show that your answer is correct by showing a combination of instructions and pipeline stage indicating the instruction and the number of read ports and write ports required for that instruction.

c. [20] <3.4> Determine any *data forwarding* for any ALUs that will be needed. Assume that there are separate ALUs for the ALU1 and ALU2 pipe stages. Put in all forwarding among ALUs needed to avoid or reduce stalls. Show the relationship between the two instructions involved in forwarding using the format of the table in Figure 3.19 but ignoring the last two columns. Be careful to consider forwarding across an intervening instruction, e.g.,

```
ADD     R1, ...
any instruction
ADD     ..., R1, ...
```

d. [20] <3.4> Show all data forwarding requirements needed to avoid or reduce stalls when either the source or destination unit is not an ALU. Use the same format as Figure 3.19, again ignoring the last two columns. Remember to forward to and from memory references.

e. [15] <3.4> Show all the remaining hazards that involve at least one unit other than an ALU as the source or destination unit. Use a table like that in Figure 3.18, but listing the length of hazard in place of the last column.

f. [15] <3.5> Show all control hazard types by example and state the length of the stall. Use a format like Figure 3.21, labeling each example.

3.4 [10] <3.2> Consider the example on page 137 that compares the unpipelined and pipelined machine. Assume that 1 ns overhead is fixed and that each pipe stage is balanced and takes 10 ns in the five-stage pipeline. Plot the speedup of the pipelined machine versus the unpipelined machine as the number of pipeline stages is increased from five stages to 20 stages, considering only the impact of the pipelining overhead and assuming that the work can be evenly divided as the stages are increased (which is not generally true). Also plot the "perfect" speedup that would be obtained if there was no overhead.

3.5 [12] <3.1–3.5> A machine is called "underpipelined" if additional levels of pipelining can be added without changing the pipeline-stall behavior appreciably. Suppose that the DLX integer pipeline was changed to four stages by merging EX and MEM and lengthening the clock cycle by 50%. How much faster would the conventional DLX pipeline be versus the underpipelined DLX on integer code only? Make sure you include the effect of any change in pipeline stalls using the data for gcc in Figure 3.38 (page 178).

3.6 [20] <3.4> Add the forwarding entries for stores and for the zero detect unit (for branches) to the table in Figure 3.19. *Hint:* Remember the tricky case:

```
ADD     R1, ...
any instruction
SW      ..., R1
```

How is the forwarding handled for this case?

3.7 [20] <3.4,3.9> Create a table showing the forwarding logic for the R4000 integer pipeline using the same format as that in Figure 3.19. Include only the DLX instructions we considered in Figure 3.19.

3.8 [15] <3.4,3.9> Create a table showing the R4000 integer hazard detection using the same format as that in Figure 3.18. Include only the instructions in the DLX subset that we considered in section 3.4.

3.9 [15] <3.5> Suppose the branch frequencies (as percentages of all instructions) are as follows:

Conditional branches	20%
Jumps and calls	5%
Conditional branches	60% are taken

We are examining a four-deep pipeline where the branch is resolved at the end of the second cycle for unconditional branches and at the end of the third cycle for conditional branches. Assuming that only the first pipe stage can always be done independent of whether the branch goes and ignoring other pipeline stalls, how much faster would the machine be without any branch hazards?

3.10 [20/20] <3.4> Suppose that we have the pipeline layout shown in Figure 3.64.

Stage	Function
1	Instruction fetch
2	Operand decode
3	Execution or memory access (branch resolution)

FIGURE 3.64 Pipeline stages.

All data dependences are between the register written in stage 3 of instruction *i* and a register read in stage 2 of instruction $i + 1$, before instruction *i* has completed. The probability of such an interlock occurring is $1/p$.

We are considering a change in the machine organization that would write back the result of an instruction during an effective fourth pipe stage. This would decrease the length of the clock cycle by *d* (i.e., if the length of the clock cycle was T, it is now $T - d$). The probability of a dependence between instruction *i* and instruction $i + 2$ is p^{-2}. (Assume that the value of p^{-1} excludes instructions that would interlock on $i + 2$.) The branch would also be resolved during the fourth stage.

a. [20] <3.4> Assume that we add no additional forwarding hardware for the four-stage pipeline. Considering only the data hazard, find the lower bound on *d* that makes this a profitable change. Assume that each result has exactly one use and that the basic clock cycle has length T.

b. [20] <3.4> Now assume that we have used forwarding to eliminate the extra hazard introduced by the change. That is, for all *data* hazards the pipeline length is *effectively* 3. This design may still not be worthwhile because of the impact of control hazards coming from a four-stage versus a three-stage pipeline. Assume that only stage 1 of the pipeline can be safely executed before we decide whether a branch goes or not. We want to know the impact of branch hazards before this longer pipeline does not yield high performance. Find an upper bound on the percentages of conditional branches in

programs in terms of the ratio of *d* to the original clock-cycle time, so that the longer pipeline has better performance. If *d* is a 10% reduction, what is the maximum percentage of conditional branches before we lose with this longer pipeline? Assume the taken-branch frequency for conditional branches is 60%.

3.11 [20] <3.4,3.7> Construct a table like Figure 3.18 that shows the data hazard stalls for the DLX FP pipeline as shown in Figure 3.44. Consider both integer-FP and FP-FP interactions but ignore divides (FP and integer).

3.12 [20] <3.4,3.7> Construct the forwarding table for the DLX FP pipeline of Figure 3.44 as we did in Figure 3.19. Consider both FP to FP forwarding and forwarding of FP loads to the FP units but ignore FP and integer divides.

3.13 [25] <3.4,3.7> Suppose DLX had only one register set. Construct the forwarding table for the FP and integer instructions using the format of Figure 3.19. Assume the DLX pipeline in Figure 3.44. Ignore FP and integer divides.

3.14 [15] <3.4,3.7> Construct a table like Figure 3.18 to check for WAW stalls in the DLX FP pipeline of Figure 3.44. Do not consider integer or FP divides.

3.15 [20] <3.4,3.7> Construct a table like Figure 3.18 that shows the structural stalls for the R4000 FP pipeline.

3.16 [35] <3.2–3.7> Change the DLX instruction simulator to be pipelined. Measure the frequency of empty branch-delay slots, the frequency of load delays, and the frequency of FP stalls for a variety of integer and FP programs. Also, measure the frequency of forwarding operations. Determine the performance impact of eliminating forwarding and stalling.

3.17 [35] <3.7> Using a DLX simulator, create a DLX pipeline simulator. Explore the impact of lengthening the FP pipelines, assuming both fully pipelined and unpipelined FP units. How does clustering of FP operations affect the results? Which FP units are most susceptible to changes in the FP pipeline length?

3.18 [40] <3.3–3.5> Write an instruction scheduler for DLX that works on DLX assembly language. Evaluate your scheduler using either profiles of programs or a pipeline simulator. If the DLX C compiler does optimization, evaluate your scheduler's performance both with and without optimization.