Pipelining: Basic and Intermediate Concepts

*It is quite a three-pipe problem.*

Sir Arthur Conan Doyle  
*The Adventures of Sherlock Holmes*
A.1 Introduction

Many readers of this text will have covered the basics of pipelining in another text (such as our more basic text Computer Organization and Design) or in another course. Because Chapters 3 and 4 build heavily on this material, readers should ensure that they are familiar with the concepts discussed in this Appendix before proceeding. As you read Chapter 3, you may find it helpful to turn to this material for a quick review.

We begin the appendix with the basics of pipelining, including discussing the datapath implications, introducing hazards, and examining the performance of pipelines. This section describes the basic five-stage RISC pipeline that is the basis for the rest of the Appendix. Section A.2 describes the issue of hazards, why they cause performance problems and how they can be dealt with. Section A.3 discusses how the simple 5-stage pipeline is actually implemented, focusing on control and how hazards are dealt with.

Section A.4 discusses the interaction between pipelining and various aspects of instruction set design, including discussing the important topic of exceptions and their interaction with pipelining. Readers unfamiliar with the concepts of
precise and imprecise interrupts and resumption after exceptions will find this material useful, since they are key to understanding the more advanced approaches in Chapter 3.

Section A.5 discusses how the 5-stage pipeline can be extended to handle longer running floating point instructions. Section A.6 puts these concepts together in a case study of a deeply pipelined processor, the MIPS R4000/4400, including both the 8-stage integer pipeline and the floating-point pipeline. In contrast, Section A.7 discusses the MIPS R4300 series, a popular embedded processor, which uses the simple five-stage pipeline structure.

Section A.8 introduces the concept of dynamic scheduling and the use of scoreboards to implement dynamic scheduling. It is introduced as a cross cutting issue, since it can be used to serve as an introduction to the core concepts in Chapter 3, which focused on dynamically scheduled approaches. Section A.8 is also a gentle introduction to the more complex Tomasulo’s algorithm covered in Chapter 3. Although Tomasulo’s algorithm can be covered and understood without introducing scoreboarding, the scoreboarding approach is simpler and easier to comprehend.

The last three sections of this appendix (A.9–A.11) provide fallacies and pitfalls, summarize the key ideas of this appendix, and provide a brief history of the concepts described in this appendix.

What is pipelining?

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction. 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 processor cycle. Because all stages proceed at the same time, the length of a processor 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 processor cycle is usually
one clock cycle (sometimes it is two, rarely more).

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 processor—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 processor 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 processor 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
processor that takes one (long) clock cycle per instruction, then pipelining de-
creases 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 6), it is not visible to the pro-
grammer. In this appendix we will first cover the concept of pipelining using a
classic five-stage pipeline; other chapters investigate the more sophisticated pipe-
lining techniques in use in modern processors. Before we say more about pipelin-
ing and its use in a processor, we need a simple instruction set, which we
introduce next.

The Basics of a RISC Instruction Set

Throughout this book we use a RISC (Reduced Instruction Set Computer) archi-
tecture or load-store architecture to illustrate the basic concepts, though nearly all
the ideas we introduce in this book are applicable to other processors. In this sec-
tion we introduce the core of a typical RISC architecture. In this appendix, and
throughout the book, our default RISC architecture is MIPS. In many places, the
concepts are significantly similar that we will not need to distinguish the exact ar-
chitecture. RISC architectures are characterized by a few key properties, which
dramatically simplify their implementation:

- All operations on data apply to data in registers and typically change the entire
register (32 or 64 bits per register).

- The only operations that affect memory are load and store operations that move data from memory to a register or to memory from a register, respectively. Load and store operations that load or store less than a full register (e.g., a byte, 16-bits, or 32-bits) are often available.

- The instructions formats are small in number with all instructions typically being one size.

These simple properties lead to dramatic simplifications in the implementation of pipelining, which is why these instruction sets were designed this way.

For consistency with the rest of the text, we use MIPS-64, the 64-bit version of the MIPS instruction set. The extended 64-bit instructions are generally designated by having an extra D on the end of the mnemonic. For example ADDD is the 64-bit version of an add instruction, while LD is the 64-bit version of a load instruction.

Like other RISC architectures, the MIPS instruction set provides 32 registers, although register 0 always has the value 0. Most RISC architectures, like MIPS, have three classes of instructions (see Chapter 2 for more detail):

1. ALU instructions: these instructions take either two registers or a register and a sign-extended immediate (called ALU immediate instructions, they have a 16-bit offset in MIPS), operate on them, and store the result into third register. Typical operations include add (ADDD), subtract (SUBD), and logical operations (such as AND or OR), which do not differentiate between 32-bit and 64-bit versions. Immediate versions of these instructions use the same mnemonics with a suffix of I. In MIPS, there are both signed and unsigned forms of the arithmetic instructions; the unsigned forms, which do not generate overflow exceptions—and thus are the same in 32-bit and 64-bit mode—have a U at the end (e.g. ADDU, SUBU, ADDUI).

2. Load and store instructions: these instructions take a register source, called the base register, and an immediate field (16-bit in MIPS), called the offset, as operand. The sum—called the effective address—of the contents of the base register and the sign-extended offset is used as a memory address. In the case of a load instruction, a second register operand acts as the destination for the data loaded from memory. In the case of a store, the second register operand is the source of the data that is stored into memory. The instructions load word (LD) and store word (SD) load or store the entire 64-bit register contents.

3. Branches and Jumps: Branches are conditional transfers of control. There are usually two ways of specifying the branch condition in RISC architectures: with a set of condition bits (sometimes called a condition code) or by a limited set of comparisons between a pair of registers or between a register and zero. MIPS uses the later. For this appendix, we consider only comparisons for
equality between two registers. In all RISC architectures, the branch destination is obtained by adding a sign-extended offset (16-bits in MIPS) to the current PC. Unconditional jumps are provided in many RISC architectures, but we will not cover jumps in this appendix.

A Simple Implementation of a RISC Instruction Set
To understand how a RISC instruction set can be implemented in a pipelined fashion, we need to understand how it is implemented \textit{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. 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. Our implementation will focus only on a pipeline for an integer subset of a RISC architecture that consists of load-store word, branch, and integer ALU operations.

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

1. \textit{Instruction fetch cycle} (IF):

   Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential PC by adding four (since each instruction is four bytes) to the PC.

2. \textit{Instruction decode/register fetch cycle} (ID):

   Decode the instruction and read the registers corresponding to register source specifiers from the register file. Do the equality test on the registers as they are read, for a possible branch. Sign extend the offset field of the instruction in case it is needed. Compute the possible branch target address by adding the sign-extended offset to the incremented PC. In an aggressive implementation, which we assume, the branch can be completed at the end of this stage, by storing the branch target address into the PC, if the condition test yielded true.

   Decoding is done in parallel with reading registers, which is possible because the register specifiers are at a fixed location in a RISC architecture. 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 performance. Because the immediate portion of an instruction is also located in an identical place, the sign-extended immediate is also calculated during this cycle in case it is needed.

3. \textit{Execution/effective address cycle} (EX):
The ALU operates on the operands prepared in the prior cycle, performing one of three functions depending on the instruction type.

- Memory reference:
The ALU adds the base register and the offset to form the effective address.

- Register-Register ALU instruction:
The ALU performs the operation specified by the ALU opcode on the values read from the register file.

- Register-Immediate ALU instruction:
The ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate.

In a load-store architecture 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 and perform an operation on the data.

4. *Memory access* (MEM):

   If the instruction is a load, then read from memory using the effective address computed in the previous cycle. If it is a store, then the data from the second register read from the register file is written into memory using the effective address.

5. *Write-back cycle* (WB):
   - Register-Register ALU instruction or Load instruction:
     Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction).

In this implementation, branch instructions require two cycles, store instructions require four cycles, and all other instructions require five cycles. Assuming the branch frequency of 12% and a store frequency of 10%, a typical instruction distribution, leads to an overall CPI of 4.54. This implementation, however, is not optimal either in achieving the best performance or in using the minimal amount of hardware given the performance level; we leave the improvement of this design as an exercise for the reader and instead focus on pipelining this version.

**The Classic Five-Stage Pipeline for A RISC processor**

We can pipeline the execution described above with almost no changes by simply starting a new instruction on each clock cycle. (See why we chose this 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 A.1, which is the typical way a pipeline structure is drawn. Although 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.

<table>
<thead>
<tr>
<th>Instruction number</th>
<th>Clock number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Instruction $i$</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction $i + 1$</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction $i + 2$</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction $i + 3$</td>
<td>IF</td>
</tr>
<tr>
<td>Instruction $i + 4$</td>
<td>IF</td>
</tr>
</tbody>
</table>

**FIGURE A.1 Simple RISC 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 processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unipipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write back.

You may find it hard to believe that pipelining is as simple as this; it’s not. In this and the following sections, we will make our RISC pipeline “real” by dealing with problems that pipelining introduces.

To start with, we have to determine what happens on every clock cycle of the processor 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 a RISC instruction set makes resource evaluation relatively easy. Figure A.2 shows a simplified version of a RISC 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, we use 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 and data memory access. Notice that if our pipelined processor has a clock cycle that is equal to that of the unipipelined version, the memory system must deliver five times the bandwidth. This increased demand is one cost of higher performance.

Second, the register file is used in the two stages: one for reading in ID and one for writing in WB. These uses are distinct, so we simply show the register file in two places. Hence, we need to perform two reads and one write every clock cy-
To handle reads and a write to the same register (and for another reason, which will become obvious shortly), we perform the register write in the first half of the clock cycle and the read in the second half.

Third, Figure A.2 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.

---

**FIGURE A.2** 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.

This figure is from old Chapter 3, but I have modified it. By adding the path around the register file. The same change is needed in Figure 4. Figures 5, 6, 7, 8, 9 need this addition PLUS the path around the ALU that does appear in 1.18 and 1.19.
during the IF stage in preparation for the next instruction. Furthermore, we must also have an adder to compute the potential branch target during ID. One further problem is that a branch does not change the PC until the ID stage. This causes a problem, which we ignore for now, but will handle shortly.

Although it is critical to ensure that instructions in the pipeline do not attempt to use the hardware resources at the same time, we must also ensure that instructions in different stages of the pipeline do not interfere with one another. This separation is done by introducing pipeline registers between successive stages of the pipeline, so that at the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next clock cycle. Figure 3 shows the pipeline drawn with these pipeline registers.

Although many figures will omit such registers for simplicity, they are required to make the pipeline operate properly and must be present. Of course, similar registers would be needed even in a multicycle datapath that had no pipelining (since only values in registers are preserved across clock boundaries). In the case of a pipelined processor, the pipeline registers also play the key role of carrying intermediate results from one stage to another where the source and destination may be directly adjacent. For example, the register value to be stored during a store instruction is read during RF, but not actually used until MEM; it is passed through two pipeline registers to reach the data memory during the MEM stage. Likewise, the result of an ALU instruction is computed during EX, but not actually stored until WB; it arrives there by passing through two pipeline registers. It is sometimes useful to name the pipeline registers, and we follow the convention of naming them by the pipeline stages they connect, so that the registers are called ID/IF, IF/RF, RF/EX, EX/MEM, MEM/WB.

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. The interested reader should see Kunkel and Smith [1986]. As we will see in Chapter 3, this overhead affected the performance gains achieved by the Pentium 4 versus the Pentium 3.

**Example**

Consider the unpipelined processor in the previous section. Assume that it has a 1 ns clock cycle 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%, respec-
A.2 The Major Hurdle of Pipelining—Pipeline Hazards

Suppose that due to clock skew and setup, pipelining the processor adds 0.2 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 processor is

\[
\text{Average instruction execution time} = \text{Clock cycle} \times \text{Average CPI}
\]

\[
= 1 \text{ ns} \times ((40\% + 20\%) \times 4 + 40\% \times 5)
\]

\[
= 1 \text{ ns} \times 4.4
\]

\[
= 4.4 \text{ ns}
\]

In the pipelined implementation, the clock must run at the speed of the slowest stage plus overhead, which will be 1 + 0.2 or 1.2 ns; this is the average instruction execution time. Thus, the speedup from pipelining is

\[
\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}
\]

\[
= \frac{4.4 \text{ ns}}{1.2 \text{ ns}} = 3.7 \text{ times}
\]

The 0.2 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.

This simple RISC pipeline 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.

### A.2 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 simultaneously in 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. Avoiding 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 appendix, 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. 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.

\[
\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}
\]

\[
= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}
\]

\[
= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}
\]

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 processor is almost always 1. Hence, we can compute the pipelined CPI:

\[
\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}
\]

\[
= 1 + \text{Pipeline stall clock cycles per instruction}
\]

If we ignore the cycle time overhead of pipelining and assume the stages are perfectly balanced, then the cycle time of the two processors 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
Exercises

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 processor, as well as that of the pipelined processor, is 1. This leads to

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

In cases where the pipe stages are perfectly balanced and there is no overhead, the clock cycle on the pipelined processor is smaller than the clock cycle of the unpipelined processor 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:

\[
\text{Speedup from pipelining} = \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}
\]

\[
= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{1}{\text{Pipeline depth}}
\]

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 processor 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 processor 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 processor 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 processors 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 A.4. To resolve this hazard, we stall the pipeline for one clock cycle when the data memory access occurs. 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.

Designers often indicate stall behavior using a simple diagram with only the pipe stage names, as in Figure A.5. The form of Figure A.5 shows the stall by indicating the cycle when no action occurs and simply shifting instruction 3 to the right (which delays its 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.

**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 processor, ignoring the structural hazard, is 1. Assume that the processor with the structural hazard has a clock rate that is 1.05 times higher than the clock rate of the processor 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 processors:

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

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

\[
\text{Average instruction time} = \text{CPI} \times \text{Clock cycle time}_{\text{ideal}}
\]

\[
= (1 + 0.4 \times 1) \times \frac{\text{Clock cycle time}_{\text{ideal}}}{1.05}
\]

\[
= 1.3 \times \text{Clock cycle time}_{\text{ideal}}
\]
A.2 The Major Hurdle of Pipelining—Pipeline Hazards

Clearly, the processor without the structural hazard is faster; we can use the ratio of the average instruction times to conclude that the processor 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. Chapter 5 discusses both the split cache and instruction buffer ideas.

If all other factors are equal, a processor without structural hazards will al-
ways have a lower CPI. Why, then, would a designer allow structural hazards? The primary reason is to reduce cost of the unit, since pipelining all the functional units, or duplicating them, may be too costly. For example, processors 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 is rare, it may not be worth the cost to avoid it.

### Data Hazards

A major effect of pipelining is to change the relative timing of instructions by overlapping their execution. This overlap 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 processor. Consider the pipelined execution of these instructions:

\[
\begin{align*}
  &\text{ADDD} & R1, R2, R3 \\
  &\text{SUBD} & R4, R1, R5 \\
  &\text{AND} & R6, R1, R7 \\
  &\text{OR} & R8, R1, R9 \\
\end{align*}
\]
A.2 The Major Hurdle of Pipelining—Pipeline Hazards

XOR R10, R1, R11

All the instructions after the ADD use the result of the ADD instruction. As shown in Figure A.6, the ADDD instruction writes the value of R1 in the WB pipe stage, but the SUBD 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

FIGURE A.6 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.

I made two small text changes, but we also need to Add path around Reg box and path around ALU box as in 1.18.

the SUBD 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 A.6, 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 also operates without incurring a hazard because we perform the register file reads in the second half of the cycle and the writes in the first half.

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 A.6 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 the pipeline register where the ADD stores it to where the SUB needs it, then the need for a stall can be avoided. Using this observation, forwarding works as follows:

1. The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.

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 relationship is also true for the case of an interrupt between the two instructions.

As the example in Figure A.6 shows, we need to forward results not only from the immediately previous instruction, but possibly from an instruction that started two cycles earlier. Figure A.7 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 pipeline register corre-
A.2 The Major Hurdle of Pipelining—Pipeline Hazards

Pipeline Hazards

The inputs for the `SUB` and `AND` instructions forward from the pipeline registers 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`.

I made two small text changes, but we also need to Add path around Reg box and path around ALU box as in 1.18.

I made two small text changes, but we also need to Add path around Reg box and path around ALU box as in 1.18.

FIGURE A.7 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 pipeline registers 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`.

I made two small text changes, but we also need to Add path around Reg box and path around ALU box as in 1.18.

...
ADD R1, R2, R3
LD R4, 0(R1)
SD 12(R1), R4

To prevent a stall in this sequence, we would need to forward the values of the ALU output and memory unit output from the pipeline registers to the ALU and data memory inputs. Figure A.8 shows all the forwarding paths for this example.

I made two small text changes, but we also need to Add path around Reg box and path around ALU box as in 1.18.

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

LD R1, 0(R2)

FIGURE A.8 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 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.
The pipelined datapath with the bypass paths for this example is shown in Figure A.9. This case is different from the situation with back-to-back ALU operations. The LD instruction does not have the data until the end of clock cycle 4 (its MEM cycle), while the SUBD 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 10 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 pipeline 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 SUBD instruction, the forwarded result arrives too late—at the end of a clock cycle, when it is needed at the beginning.

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. The CPI for the stalled instruction increases by the length of the stall (one clock cycle in this case).

Figure A.10 shows the pipeline before and after the stall using the names of the pipeline stages. 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 se-
FIGURE A.9 The load instruction can bypass its results to the \texttt{AND} and \texttt{OR} instructions, but not to the \texttt{SUBD}, since that would mean forwarding the result in "negative time."

I made two small text changes, but we also need to Add path around Reg box and path around ALU box as in 1.18.
sequence to grow by one. No instruction is started during clock cycle 4 (and none finishes during cycle 6).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1,0(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>SUBD R4,R1,R5</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>AND R6,R1,R7</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>OR R8,R1,R9</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
<th>Stage</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD R1,0(R2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>SUBD R4,R1,R5</td>
<td>IF</td>
<td>ID</td>
<td>stall</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>AND R6,R1,R7</td>
<td>IF</td>
<td>stall</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>OR R8,R1,R9</td>
<td>stall</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
</tr>
</tbody>
</table>

**FIGURE A.11** A branch causes a one-cycle stall in the five-stage pipeline. 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.
One stall cycle for every branch will yield a performance loss of 10% to 30% depending on the branch frequency, so we will examine some techniques to deal with this loss.

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. Chapters 3 and 4 look at more powerful hardware and software techniques for both static and dynamic branch prediction.

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 A.11. 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 processor state until the branch outcome is definitely known. The complexity of this scheme arises from having to know when the state might be changed by an instruction and how to “back out” such a change.

In the simple five-stage 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 and restart the fetch at the target address. Figure A.12 shows both situations.

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 five-stage pipeline we don’t know the target address any earlier than we know the branch outcome, there is no advantage in this approach for this pipeline. In some processors—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 processors is called delayed branch. This technique was heavily used in early RISC processors and works reasonably well
A.2 The Major Hurdle of Pipelining—Pipeline Hazards

Pipeline hazards arise in the five-stage pipeline. In a delayed branch, the execution cycle with a branch delay of one is

```
branch instruction
sequential successor₁
branch target if taken
```

The sequential successor is in the \textit{branch-delay slot}. This instruction is executed whether or not the branch is taken. The pipeline behavior of the five-stage pipeline with a branch delay is shown in Figure A.13. Although it is possible to have a branch delay longer than one, in practice, all processors with delayed branch have a single instruction delay; other techniques are used if the pipeline has a longer potential branch penalty.

The job of the compiler is to make the successor instructions valid and useful. A number of optimizations are used. Figure A.14 shows the three ways in which the branch delay can be scheduled.

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. To improve the ability of the compiler to fill branch delay slots, most processors with conditional branches have introduced a \textit{cancelling} or \textit{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

### Table A.13

<table>
<thead>
<tr>
<th>Untaken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction (i + 1)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 2)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 3)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Instruction (i + 4)</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

### Table A.14

<table>
<thead>
<tr>
<th>Taken branch instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction (i + 1)</td>
<td>IF</td>
<td>idle</td>
<td>idle</td>
<td>idle</td>
<td>idle</td>
</tr>
<tr>
<td>Branch target</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 1</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
<tr>
<td>Branch target + 2</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

**FIGURE A.12** 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.
no-op. We explore the performance of cancelling delayed branches in the exercises.

**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.

---

**EXAMPLE**

For a deeper pipeline, such as that in a MIPS R4000, it takes three pipe-
A.2 The Major Hurdle of Pipelining—Pipeline Hazards

Pipeline Hazards

A line 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...
Find the effective addition to the CPI arising from branches for this pipeline, assuming the following frequencies:

<table>
<thead>
<tr>
<th>Unconditional branch</th>
<th>4%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditional branch, untaken</td>
<td>6%</td>
</tr>
<tr>
<td>Conditional branch, taken</td>
<td>10%</td>
</tr>
</tbody>
</table>

**Answer**  
We find the CPIs by multiplying the relative frequency of unconditional, conditional untaken, and conditional taken branches by the respective penalties. The results are shown in Figure 16.

<table>
<thead>
<tr>
<th>Branch scheme</th>
<th>Unconditional branches</th>
<th>Untaken conditional branches</th>
<th>Taken conditional branches</th>
<th>All branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency of event</td>
<td>4%</td>
<td>6%</td>
<td>10%</td>
<td>20%</td>
</tr>
<tr>
<td>Stall pipeline</td>
<td>0.08</td>
<td>0.18</td>
<td>0.30</td>
<td>0.56</td>
</tr>
<tr>
<td>Predict taken</td>
<td>0.08</td>
<td>0.18</td>
<td>0.20</td>
<td>0.46</td>
</tr>
<tr>
<td>Predict untaken</td>
<td>0.08</td>
<td>0.00</td>
<td>0.30</td>
<td>0.38</td>
</tr>
</tbody>
</table>

**Figure A.15** Branch penalties for the three simplest prediction schemes for a deeper pipeline.

**Figure A.16** 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.

**A.3 How is Pipelining Implemented?**
Before we proceed to basic pipelining, we need to review a simple implementation of an unpipelined version of MIPS.

**A Simple Implementation of MIPS**

In this section we follow the style of pages A.29 to A.33, showing first a simple unpipelined implementation and then the pipelined implementation. This time, however, our example is specific to the MIPS architecture.

In this section we focus on a pipeline for an integer subset of MIPS that consists of load-store word, branch equal zero, and integer ALU operations. Later in this appendix, we will incorporate the basic floating-point operations. Although we discuss only a subset of MIPS, the basic principles can be extended to handle all the instructions.

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

1. *Instruction fetch cycle (IF):*

   \[
   \begin{align*}
   \text{IR} & \leftarrow \text{Mem}[PC] ; \\
   \text{NPC} & \leftarrow \text{PC} + 4 ;
   \end{align*}
   \]

   *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):*

   \[
   \begin{align*}
   \text{A} & \leftarrow \text{Regs}[rs] ; \\
   \text{B} & \leftarrow \text{Regs}[rt] ; \\
   \text{Imm} & \leftarrow \text{sign extended immediate field of IR} ;
   \end{align*}
   \]

   *Operation:* Decode the instruction and access the register file to read the registers I(rs and rt are the register specifiers). 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 MIPS instruction format (see Figure 2.21 on page 99). Because the immediate portion of an instruction is located in an identical place in every MIPS 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 MIPS instruction type.

- **Memory reference:**

  \[ \text{ALUOutput} \leftarrow A + \text{Imm}; \]

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

- **Register-Register ALU instruction:**

  \[ \text{ALUOutput} \leftarrow A \text{ func } B; \]

  *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:**

  \[ \text{ALUOutput} \leftarrow A \text{ op } \text{Imm}; \]

  *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:**

  \[ \text{ALUOutput} \leftarrow \text{NPC } + \text{Imm}; \]
  \[ \text{Cond } \leftarrow (A == 0) \]

  *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. Since we are considering only one form of branch (BEQZ), the comparison is against 0.

The load-store architecture of MIPS 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: \( \text{PC } \leftarrow \text{NPC}; \)
A.3 How is Pipelining Implemented?

- Memory reference:

\[
\text{LMD} \leftarrow \text{Mem[ALUOutput]} \text{ or } \\
\text{Mem[ALUOutput]} \leftarrow \text{B};
\]

*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:

\[
\text{if (cond) PC} \leftarrow \text{ALUOutput}
\]

*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:

\[
\text{Regs[rd]} \leftarrow \text{ALUOutput};
\]

- Register-Immediate ALU instruction:

\[
\text{Regs[rt]} \leftarrow \text{ALUOutput};
\]

- Load instruction:

\[
\text{Regs[rt]} \leftarrow \text{LMD};
\]

*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 (rd or rt) depending on the effective opcode.

Figure A.17 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.
Although all processors today are pipelined, this multicycle implementation is a reasonable approximation of how most processors 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 processor, microcode control could be used. In either event, an instruction sequence like that above would determine the structure of the control.

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 A.17, 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 CPU 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 processor. The clock cycle, however, would be roughly equal to five times the clock cycle of the multicycle processor, 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 CPUs 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 processor.

**A Basic Pipeline for MIPS**

As before, we can pipeline the datapath of Figure A.17 with almost no changes by starting a new instruction on each clock cycle. 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 A.18 shows the MIPS pipeline with the appropriate registers, called *pipeline registers* or *pipeline latches*, between each pipeline stage. The registers are labeled with the names of the stag-
es they connect. Figure A.18 is drawn so that connections through the pipeline registers from one stage to another are clear.

![Diagram of a pipelined datapath](image)

**FIGURE A.18** 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.

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
A.3 How is Pipelining Implemented?

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 A.19 shows this view. Fields of the pipeline registers are named so as to show

<table>
<thead>
<tr>
<th>Stage</th>
<th>Any instruction</th>
<th>ALU instruction</th>
<th>Load or store instruction</th>
<th>Branch instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>IF/ID.IR ← Mem[PC]; [rs]</td>
<td>EX/MEM.IR ← ID/EX.IR; EX/MEM.ALUOutput ← 0;</td>
<td>MEM/WB.IR ← EX/MEM.IR; MEM/WB.LMD ← Mem[EX/MEM.ALUOutput];</td>
<td>For load only: Regs [MEM/WB.IR[rd]] ← MEM/WB.LMD;</td>
</tr>
<tr>
<td></td>
<td>IF/ID.NPC,PC ← (if ((EX/MEM.opcode == branch) &amp; EX/MEM.cond) {EX/MEM.ALUOutput} else {PC+4})</td>
<td>ID/EX.A + ID/EX.Imm;</td>
<td>ID/EX.NPC+ID/EX.Imm;</td>
<td>(ID/EX.A == 0);</td>
</tr>
<tr>
<td></td>
<td>ID/EX.B ← Regs[IF/ID.IR[rt]];</td>
<td>ID/EX.A func ID/EX.B;</td>
<td>ID/EX.A + ID/EX.Imm;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ID/EX.NPC ← IF/ID.NPC; ID/EX.IR ← IF/ID.IR;</td>
<td>ID/EX.A op ID/EX.Imm;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ID/EX.Imm ← sign extend(IF/ID.IR[immediate field]);</td>
<td>EX/MEM.cond ← 0; EX/MEM.B ← ID/EX.B;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FIGURE A.19** Events on every pipe stage of the MIPS 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 A.18, 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 (the immediate field), 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.
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 A.18. 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 A.18, 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.

Implementing the Control for the MIPS 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 MIPS integer pipeline, all the data hazards can be checked during the ID phase of the pipeline. 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 processor, unless the entire processor 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 A.20 lists the variety of circumstances that we must handle.

<table>
<thead>
<tr>
<th>Situation</th>
<th>Example code sequence</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>No dependence</td>
<td>LD R1, 45 (R2) ADDD R5, R6, R7 SUBD R8, R6, R7 OR R9, R6, R7</td>
<td>No hazard possible because no dependence exists on R1 in the immediately following three instructions.</td>
</tr>
<tr>
<td>Dependence requiring stall</td>
<td>LD R1, 45 (R2) ADDD R5, R1, R7 SUBD R8, R6, R7 OR R9, R6, R7</td>
<td>Comparators detect the use of R1 in the ADD and stall the ADD (and SUB and OR) before the ADD begins EX.</td>
</tr>
<tr>
<td>Dependence overcome by forwarding</td>
<td>LD R1, 45 (R2) ADDD R5, R6, R7 SUBD R8, R1, R7 OR R9, R6, R7</td>
<td>Comparators detect use of R1 in SUB and forward result of load to ALU in time for SUB to begin EX.</td>
</tr>
<tr>
<td>Dependence with accesses in order</td>
<td>LD R1, 45 (R2) ADDD R5, R6, R7 SUBD R8, R6, R7 OR R9, R1, R7</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 A.21 shows a table that detects all load interlocks when the instruction using the load result is in the ID stage.

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 E.E.3.1, 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 A.22 shows the comparisons and possible forwarding operations where the destination of the forwarded result is an ALU input for the instruction currently in EX.

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 A.23 shows the relevant segments of the pipelined datapath with the additional multiplexers and connections in place.

For MIPS, 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.

### Dealing with Branches in the Pipeline

In MIPS, the branches (BEQZ and BNEZ) require testing a register for equality to zero or to another register for equality. 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
### A.3 How is Pipelining Implemented?

Function so far, is not usable until EX. Figure A.24 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.

<table>
<thead>
<tr>
<th>Pipeline register containing source instruction</th>
<th>Opcode of source instruction</th>
<th>Pipeline register containing destination instruction</th>
<th>Opcode of destination instruction</th>
<th>Destination of the forwarded result</th>
<th>Comparison (if equal then forward)</th>
</tr>
</thead>
<tbody>
<tr>
<td>EX/MEM</td>
<td>Register-register ALU</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>EX/MEM.IR [rd] == ID/EX.IR [rs]</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>EX/MEM.IR [rt] == ID/EX.IR [rt]</td>
</tr>
<tr>
<td>EX/MEM</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>EX/MEM.IR [rt] == ID/EX.IR [rt]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>MEM/WB.IR [rt] == ID/EX.IR [rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>ALU immediate</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>MEM/WB.IR [rt] == ID/EX.IR [rt]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Load</td>
<td>ID/EX</td>
<td>Register-register ALU, ALU immediate, load, store, branch</td>
<td>Top ALU input</td>
<td>MEM/WB.IR [rt] == ID/EX.IR [rs]</td>
</tr>
<tr>
<td>MEM/WB</td>
<td>Load</td>
<td>ID/EX</td>
<td>Register-register ALU</td>
<td>Bottom ALU input</td>
<td>MEM/WB.IR [rt] == ID/EX.IR [rt]</td>
</tr>
</tbody>
</table>

**FIGURE A.22** 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 E.17 on page E-32 and in Figure E.23 on page E-40. 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.
to one cycle, it means that an ALU instruction followed by a branch on the result of the instruction will incur a data hazard stall. Figure A.25 shows the branch portion of the revised pipeline table from Figure 19 (page 35).

In some processors, 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 processor 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 CPUs that implement more complex instruction sets have branch delays of four clock cycles or more, and large, deeply pipelined processors 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 processor. A high CPI processor can afford to have more ex-
A.3 How is Pipelining Implemented?

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.

FIGURE A.24 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, C.18 obtained the branch target address from the EX/MEM register and wrote the result during the MEM clock cycle. As mentioned in C.18, 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.

pensive branches because the percentage of the processor’s performance that will be lost from branches is less.

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.
Appendix A   Pipelining: Basic and Intermediate Concepts

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 CPU because the overlapping of instructions makes it more difficult to know whether an instruction can safely change the state of the CPU. In a pipelined CPU, 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 CPU to abort the instructions in the pipeline before they complete. Before we discuss these

<table>
<thead>
<tr>
<th>Pipe stage</th>
<th>Branch instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>IF/ID.IR ← Mem[PC]; IF/ID.NPC,PC ← (if ((IF/ID.opcode == branch) &amp; (Regs[IF/ID.IR6..10] op 0)) {IF/ID.NPC + (IF/ID.IR16)16##IF/ID.IR16..31} else {PC+4});</td>
</tr>
<tr>
<td>ID</td>
<td>ID/EX.A ← Regs[IF/ID.IR6..10]; ID/EX.B ← Regs[IF/ID.IR11..15]; ID/EX.IR ← IF/ID.IR; ID/EX.Imm ← (IF/ID.IR16)16##IF/ID.IR16..31</td>
</tr>
<tr>
<td>EX</td>
<td>MEM</td>
</tr>
<tr>
<td>WB</td>
<td>FIGURE A.25 This revised pipeline structure is based on the original in Figure 19, page 35. It uses a separate adder, as in Figure A.24, 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.IR6..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 processor 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.</td>
</tr>
</tbody>
</table>

A.4  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.
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 CPUs. 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
- 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 A.26 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:

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 CPU 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-
<table>
<thead>
<tr>
<th>Exception event</th>
<th>IBM 360</th>
<th>VAX</th>
<th>Motorola 680x0</th>
<th>Intel 80x86</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O device request</td>
<td>Input/output interruption</td>
<td>Device interrupt</td>
<td>Exception (Level0...7 autovector)</td>
<td>Vectored interrupt</td>
</tr>
<tr>
<td>Invoking the operating system service</td>
<td>Supervisor call interruption</td>
<td>Exception (change mode supervisor trap)</td>
<td>Exception (unimplemented instruction)— on Macintosh</td>
<td>Interrupt (INT instruction)</td>
</tr>
<tr>
<td>from a user program</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tracing instruction execution</td>
<td>Not applicable</td>
<td>Exception (trace fault)</td>
<td>Exception (trace)</td>
<td>Interrupt (single-step trap)</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>Not applicable</td>
<td>Exception (breakpoint fault)</td>
<td>Exception (illegal instruction or breakpoint)</td>
<td>Interrupt (breakpoint trap)</td>
</tr>
<tr>
<td>Integer arithmetic overflow or underflow; FP trap</td>
<td>Program interruption (overflow or underflow exception)</td>
<td>Exception (integer overflow trap or floating underflow fault)</td>
<td>Exception (floating-point coprocessor errors)</td>
<td>Interrupt (overflow trap or math unit exception)</td>
</tr>
<tr>
<td>Page fault (not in main memory)</td>
<td>Not applicable (only in 370)</td>
<td>Exception (translation not valid fault)</td>
<td>Exception (memory-management unit errors)</td>
<td>Interrupt (page fault)</td>
</tr>
<tr>
<td>Misaligned memory accesses</td>
<td>Program interruption (specification exception)</td>
<td>Not applicable</td>
<td>Exception (address error)</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Memory protection violations</td>
<td>Program interruption (protection exception)</td>
<td>Exception (access control violation fault)</td>
<td>Exception (bus error)</td>
<td>Interrupt (protection exception)</td>
</tr>
<tr>
<td>Using undefined instructions</td>
<td>Program interruption (operation exception)</td>
<td>Exception (opcode privileged/reserved fault)</td>
<td>Exception (illegal instruction or breakpoint/unimplemented instruction)</td>
<td>Interrupt (invalid opcode)</td>
</tr>
<tr>
<td>Hardware malfunctions</td>
<td>Machine-check interruption</td>
<td>Exception (machine-check abort)</td>
<td>Exception (bus error)</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Power failure</td>
<td>Machine-check interruption</td>
<td>Urgent interrupt</td>
<td>Not applicable</td>
<td>Nonmaskable interrupt</td>
</tr>
</tbody>
</table>

**FIGURE A.26** 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.

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
A.4 What Makes Pipelining Hard to Implement?

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 CPU need not be able to restart execution of the same program after handling the exception.

Figure A.27 classifies the examples from Figure A.26 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 processor to handle the exception, save the state, and restart without affecting the execution of the program, the pipeline or processor is said to be **restartable**. While early supercomputers and microprocessors often lacked this property, almost all processors today support it, at least for the integer pipeline, because it is needed to implement virtual memory (see Chapter 5).

**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 MIPS 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 exe-
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

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

**FIGURE A.27** Five categories are used to define what actions are needed for the different exception types shown in Figure A.26. 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, CPUs should be able to resume after such exceptions.
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 processor 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 processor from the exception by reloading the PCs and restarting the instruction stream (using the instruction RFE in MIPS). 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 processors 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 CPUs 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 CPUs, 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 processor 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 MIPS inte-
ger pipeline. We will describe techniques for handling the more complex challenges arising in the FP pipeline in section A.5A.

**Exceptions in MIPS**

Figure A.28 shows the MIPS 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:

<table>
<thead>
<tr>
<th>LD</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDD</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
</tr>
</tbody>
</table>

This pair of instructions can cause a data page fault and an arithmetic exception at the same time, since the LD is in the MEM stage while the ADDD 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, LD followed by ADDD. The LD can get a data page fault, seen when the instruction is in MEM, and the ADDD can get an instruction page fault, seen when the ADDD 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 LD instruction first. To explain how this works,
let’s call the instruction in the position of the \texttt{LW} instruction $i$, and the instruction in the position of the \texttt{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 processor—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 A.5A., 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 processors with more powerful, longer-running instructions.

**Instruction Set Complications**

No MIPS instruction has more than one result, and our MIPS pipeline writes that result only at the end of an instruction’s execution. When an instruction is guaranteed to complete it is called \textit{committed}. In the MIPS 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 processors 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 in the IA-32 architecture 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 processor 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 IA-32 instruction that autoincrements the same register multiple times. Thus, to maintain a precise exception model, most processors with such
instructions have the ability to back out any state changes made before the instruction is committed. If an exception occurs, the processor uses this ability to reset the state of the processor to its value before the interrupted instruction started. In the next section, we will see that a more powerful MIPS floating-point pipeline can introduce similar problems, and Section A.8 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 IBM 360 (see Appendix B online). 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 CPU knows whether to restart the instruction from the beginning or from the middle of the instruction. The IA-32 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 processors 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 processors 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 processors 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 MIPS 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 ; moves between registers
ADDL3 42(R1),56(R1)+,@(R1) ; adds memory locations
SUBL2 R2,R3 ; subtracts registers
```
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 MIPS), the pipeline control is much easier. Since 1995, all Intel IA-32 microprocessors have used this strategy of converting the IA-32 instructions into microoperations, and then pipelining the microoperations.

In comparison, load-store processors 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 MIPS pipeline deals with long-running instructions, specifically floating-point operations. 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. In the 1990s, all companies moved to simpler instructions sets with the goal of reducing the complexity of aggressive implementations.

We now want to explore how our MIPS 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 MIPS floating-point pipeline. It is impractical to require that all MIPS 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 MIPS 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 A.29 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.

In reality, the intermediate results are probably not cycled around the EX unit as Figure A.29 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 A.29 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 \textit{initiation interval} or \textit{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 A.30.

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
A.5 Extending the MIPS Pipeline to Handle Multicycle Operations

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 A.30 allows up to four outstanding FP adds, seven outstanding FP/integer multiplies, and one FP divide. Figure A.31 shows how this pipeline can be drawn by extending Figure A.29. The repeat interval is implemented in Figure A.31 by adding additional pipeline stages, which

---

**FIGURE A.29** The MIPS 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.

<table>
<thead>
<tr>
<th>Functional unit</th>
<th>Latency</th>
<th>Initiation interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer ALU</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Data memory (integer and FP loads)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>FP add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP multiply (also integer multiply)</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>FP divide (also integer divide)</td>
<td>24</td>
<td>25</td>
</tr>
</tbody>
</table>

**FIGURE A.30** Latencies and initiation intervals for functional units.
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 appendix, as Figure A.32 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.

The structure of the pipeline in Figure A.31 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 asso-

**FIGURE A.31** 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.
A.5 Extending the MIPS Pipeline to Handle Multicycle Operations

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 A.31:

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 A.33 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:

<table>
<thead>
<tr>
<th>MULT_D</th>
<th>IF</th>
<th>ID</th>
<th>M1</th>
<th>M2</th>
<th>M3</th>
<th>M4</th>
<th>M5</th>
<th>M6</th>
<th>M7</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_D</td>
<td>IF</td>
<td>ID</td>
<td>A1</td>
<td>A2</td>
<td>A3</td>
<td>A4</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L.D</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S.D</td>
<td>IF</td>
<td>ID</td>
<td>EX</td>
<td>MEM</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIGURE A.32 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. The “.D” extension on the instruction mnemonic indicates double-precision (64-bit) floating point operations. FP loads and stores use a 64-bit path to memory so that the pipelining timing is just like an integer load or store.
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 processor 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.

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 processor 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.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle number</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.D F4,0(R2)</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>MULT.D F0,F4,F6</td>
<td>IF ID stall M1 M2 M3 M4 M5 M6 M7 MEM WB</td>
</tr>
<tr>
<td>ADD.D F2,F0,F8</td>
<td>IF stall ID stall stall stall stall stall stall A1 A2 A3 A4 MEM</td>
</tr>
<tr>
<td>S.D 0(R2),F2</td>
<td>IF stall stall stall stall stall stall ID ID EX stall stall stall MEM</td>
</tr>
</tbody>
</table>

**FIGURE A.33** 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 ADD. Extra hardware could easily handle this case.

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 instruc-

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Clock cycle number</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULT.D F0,F4,F6</td>
<td>IF ID M1 M2 M3 M4 M5 M6 M7 MEM WB</td>
</tr>
<tr>
<td>...</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>...</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>ADD,D F2,F4,F6</td>
<td>IF ID A1 A2 A3 A4 MEM WB</td>
</tr>
<tr>
<td>...</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>...</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>L,D F2,0(R2)</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>

**FIGURE A.34** 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 MULT, D, ADD, D, and L, D all are in the MEM stage in clock cycle 10, only the L, D actually uses the memory, so no structural hazard exists for MEM.
tions 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 \texttt{L.D} 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 \texttt{ADDD}. Note that this hazard only occurs when the result of the \texttt{ADDD} is overwritten \textit{without} any instruction ever using it! If there were a use of F2 between the \texttt{ADDD} and the \texttt{L.D}, the pipeline would need to be stalled for a RAW hazard, and the \texttt{L.D} would not issue until the \texttt{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 \texttt{L.D} appears in F2 when we are done. (As we will see in section A.A.9, 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 \texttt{ADDD} enters MEM. The second approach is to stamp out the result of the \texttt{ADDD} by detecting the hazard and changing the control so that the \texttt{ADDD} does not write its result. Then, the \texttt{L.D} 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 \texttt{L.D} is issuing. Then stalling the \texttt{L.D} or making the \texttt{ADDD} a no-op is easy. The difficult situation is to detect that the \texttt{L.D} might finish before the \texttt{ADDD}, because that requires knowing the length of the pipeline and the current position of the \texttt{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 Section A.8, 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, without 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 MIPS 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.
A.5 Extending the MIPS Pipeline to Handle Multicycle Operations

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:

\[
\begin{align*}
\text{DIV.D} & \quad F0, F2, F4 \\
\text{ADD.D} & \quad F10, F10, F8 \\
\text{SUB.D} & \quad F12, F12, F14
\end{align*}
\]

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 (see Section A.8). Because hazard detection will prevent any dependence among instructions from being violated, why is out-of-order completion a problem? Suppose that the SUB.D causes a floating-point arithmetic exception at a point where the ADD.D has completed but the DIV.D 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 DIV.D, 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 processors 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 processors 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 CPUs 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 earlier 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 auto-increment and autodecrement addressing on processors 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 Chapter 3, 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 MIPS-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 CPU 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 MIPS 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 G.

Performance of a MIPS FP Pipeline

The MIPS FP pipeline of Figure 31 on page 54 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 A.35 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 A.36 gives the complete breakdown of integer and floating-point stalls for five SPECfp benchmarks. 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 A.35** Stalls per FP operation for each major type of FP operation for the SPEC89 FP benchmarks. 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.
In this section we look at the pipeline structure and performance of the MIPS R4000 processor family, which includes the 4400. The R4000 implements MIPS-64 but uses a deeper pipeline than that of our 5-stage design both for integer and FP programs. This deeper pipeline allows it to achieve higher clock rates 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 A.37 shows the eight-stage pipeline structure using an abstracted version of the datapath. Figure A.38 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.

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 A.38 shows that load delays are two cycles, since the data value is available at the end of DS. Figure A.39 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 A.40 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 A.41 shows, untaken branches are simply one-cycle delayed branches, while taken branches have a one-cycle delay slot 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.

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 MIPS
A 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 MIPS 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. 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 A.42; these stages are combined in different orders to execute various FP operations.
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 A.43 shows the latency, initiation rate, and pipeline stages used by the most common double-precision FP operations.
From the information in Figure A.43, 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 A.44, A.45, A.46, and A.47 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 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.

**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 A.44 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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Issue/stall</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiply</td>
<td>Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>U</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>N</td>
<td>N+A</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>U</td>
<td>S+A</td>
<td>A+R</td>
<td>R+S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>U</td>
<td>S+A</td>
<td>A+R</td>
<td>R+S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stall</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>U</td>
<td>S+A</td>
<td>A+R</td>
<td>R+S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stall</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>U</td>
<td>S+A</td>
<td>A+R</td>
<td>R+S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>U</td>
<td>S+A</td>
<td>A+R</td>
<td>R+S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### FIGURE A.45 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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Issue/stall</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>U</td>
<td>S+A</td>
<td>A+R</td>
<td>R+S</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiply</td>
<td>Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>U</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>N</td>
<td>N+A</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Issue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>U</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>N</td>
<td>N+A</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### FIGURE A.46 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 A.45 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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Issue/stall</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Divide</td>
<td></td>
<td>25 26 27 28 29 30 31 32 33 34 35 36</td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td>D D D D D+A D+R D+A D+R A R</td>
</tr>
<tr>
<td></td>
<td>Issue</td>
<td>U S+A A+R R+S</td>
</tr>
<tr>
<td></td>
<td>Issue</td>
<td>U S+A A+R R+S</td>
</tr>
<tr>
<td></td>
<td>Stall</td>
<td>U S+A A+R R+S</td>
</tr>
<tr>
<td></td>
<td>Stall</td>
<td>U S+A A+R R+S</td>
</tr>
<tr>
<td></td>
<td>Stall</td>
<td>U S+A A+R R+S</td>
</tr>
<tr>
<td></td>
<td>Stall</td>
<td>U S+A A+R R+S</td>
</tr>
<tr>
<td></td>
<td>Issue</td>
<td>U S+A A+R R+S</td>
</tr>
<tr>
<td></td>
<td>Issue</td>
<td>U S+A A+R</td>
</tr>
<tr>
<td></td>
<td>Issue</td>
<td>U</td>
</tr>
</tbody>
</table>

### FIGURE A.47 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.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Issue/stall</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Issue</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12</td>
</tr>
<tr>
<td>Divide</td>
<td>Stall</td>
<td>U A R D D D D D D D D</td>
</tr>
<tr>
<td></td>
<td>Issue</td>
<td>U A R D D D D D D D D</td>
</tr>
<tr>
<td></td>
<td>Issue</td>
<td>U A R D D D D D D D</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Issue/stall</th>
<th>Clock cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Issue</td>
<td>U S+A A+R R+S</td>
</tr>
<tr>
<td>Divide</td>
<td>Stall</td>
<td>U A R D D D D D D D D D</td>
</tr>
<tr>
<td></td>
<td>Issue</td>
<td>U A R D D D D D D D D D</td>
</tr>
<tr>
<td></td>
<td>Issue</td>
<td>U A R D D D D D D D D</td>
</tr>
</tbody>
</table>
Figure A.48 shows the pipeline CPI breakdown for the R4000 pipeline for the 10 SPEC92 benchmarks. Figure A.49 shows the same data but in tabular form.

**FIGURE A.48** 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. Figure A.49 shows the numbers used to construct this plot.
From the data in Figures A.48 and A.49, we can see the penalty of the deeper pipelining. The R4000’s pipeline has much longer branch delays than the classic five-stage 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 result 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.

### A.7 Another View: The MIPS R4300 Pipeline

The five-stage pipeline described in section G.1 is considered a classic pipeline and was the basic structure used for the integer pipeline on the early RISC processors. Today, it is still heavily used as the pipeline structure for low-end to midrange embedded microprocessors. The MIPS R4300 series, manufactured by NEC, is a 64-bit processor implementing the MIPS-64 instruction set intended for the embedded space. Versions of this processor have been used in a wide vari-
Vety of embedded applications including: as the CPU in the Nintendo-64 game processor, in a range of high-end color laser printers, and as the CPU in a network router. The NEC VR 4122, which we mentioned in Chapter 1, is a version of this design incorporating only the integer datapath and using software for floating point operations.

The R4300 pipeline implements floating point by extending the pipeline length through the addition of multiple EX stages for floating point operations. This introduces one complexity that we have not seen in the simple integer pipeline: the possibility of instructions completing out-of-order. The extra length of the floating point pipeline means that unless we add unused stages to the integer pipeline (and thereby increase the number of bypass checks that must be done), an integer instruction can complete and write its results before an earlier floating point instruction completes. Normally this will not be a problem, but if the floating point instruction generates an exception, this exception could be raised after the integer instruction has completed, thereby leading to an *imprecise interrupt* (i.e., some instructions after the instruction causing an exception have completed, while the interrupting instruction has not yet completed). The MIPS R4300 uses the scheme discussed in Section A.5 to prevent imprecise interrupts from the FP unit. We will return to the topics of out-of-order instruction completion and precise interrupts in Chapter 3.

Figure 50 summarizes four versions of the MIPS R4300 processor.

<table>
<thead>
<tr>
<th>Clock Frequency</th>
<th>Performance SPECInt 92</th>
<th>Performance SPECFP 92</th>
<th>Power consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>80 MHz</td>
<td>48</td>
<td>36</td>
<td>1.5 W</td>
</tr>
<tr>
<td>100 MHz</td>
<td>60</td>
<td>45</td>
<td>1.8 W</td>
</tr>
<tr>
<td>133 MHz</td>
<td>80</td>
<td>60</td>
<td>1.9 W</td>
</tr>
<tr>
<td>167 MHz</td>
<td>100</td>
<td>75</td>
<td>2.4 W</td>
</tr>
</tbody>
</table>

**FIGURE A.50  Four version of the MIPS R4300.** Like many other processors in the embedded space, the manufacturers of the R4300 typically publish “Dhrystone MIPS”. In addition, they do not ship a full system, so it is impossible to generate true SPEC measurements. The SPEC92 numbers are manufacturer’s estimates.

### A.8 Cross Cutting Issues

#### RISC Instruction Sets and Efficiency of Pipelining

We have already discussed the advantages of instruction set simplicity in building pipelines. Simple instruction sets offer another advantage: they make it easier to
schedule code to achieve efficiency of execution in a pipeline. To see this, consider a simple example: suppose we need to add two values in memory and store the result back to memory. In some sophisticated instruction sets this will take only a single instruction; in others it will take two or three. A typical RISC architecture would require four instructions (two loads, an add, and a store). These instructions cannot be scheduled sequentially in most pipelines without intervening stalls.

With a RISC instruction set, the individual operations are separate instructions and may be individually scheduled either by the compiler (using the techniques we discussed earlier and more powerful techniques discussed in Chapter 4) or using dynamic hardware scheduling techniques (which we discuss next and in further detail in Chapter 3). These efficiency advantages, coupled with the greater ease of implementation, appear to be so significant that almost all recent pipelined implementations of complex instruction sets, actually translate their complex instructions into simple RISC-like operations, and then schedule and pipeline those operations. Chapter 3 shows that both the Pentium III and Pentium 4 use this approach.

**Dynamically Scheduled Pipelines**

Simple pipelines fetch an instruction and issue it, unless there is a data dependence between an instruction already in the pipeline and the fetched instruction that cannot be hidden with bypassing or forwarding. Forwarding logic reduces the effective pipeline latency so that the certain dependences do not result in hazards. If there is an unavoidable hazard, then the hazard detection hardware stalls the pipeline (starting with the instruction that uses the result). No new instructions are fetched or issued until the dependence is cleared. To overcome these performance losses, the compiler can attempt to schedule instructions to avoid the hazard; this approach is called *compiler* or *static scheduling*.

Several early processors used another approach, called *dynamic scheduling*, whereby the hardware rearranges the instruction execution to reduce the stalls. This section offers a simpler introduction to dynamic scheduling by explaining the scoreboard technique of the CDC 6600. Some readers will find it easier to read this material before plunging into the more complicated Tomasulo scheme, which is covered in Chapter 3. Before proceeding, you may find useful to read the first few pages (up to the section labeled “Dynamic Scheduling Using Tomasulo’s Approach” to understand the core idea and its motivation better.

All the techniques discussed in this Appendix so far use in-order instruction issue, which means that if an instruction is stalled in the pipeline, no later instructions can proceed. With in-order issue, if two instructions have a hazard between them, the pipeline will stall, even if there are later instructions which are independent and would not stall.

In the MIPS pipeline developed earlier, both structural and data hazards were checked during instruction decode (ID): When an instruction could execute prop-
erly, it was issued from ID. To allow us to begin executing the \texttt{SUBD} in the above example, we must separate the issue process into two parts: checking the structural hazards and waiting for the absence of a data hazard. We can still check for structural hazards when we issue the instruction; thus, we still use in-order instruction issue. However, we want the instructions to begin execution as soon as their data operands are available. Thus, the pipeline will do \textit{out-of-order execution}, which implies \textit{out-of-order completion}.

To implement out-of-order execution, we must split the ID pipe stage into two stages:

1. \textit{Issue}—Decode instructions, check for structural hazards.
2. \textit{Read operands}—Wait until no data hazards, then read operands.

The IF stage proceeds the Issue stage and the EX stage follows the read operands stage, just as in the MIPS pipeline. As in the MIPS floating-point pipeline, execution may take multiple cycles, depending on the operation. Thus, we may need to distinguish when an instruction begins execution and when it completes execution; between the two times, the instruction is in execution. This allows multiple instructions to be in execution at the same time. In addition to these changes to the pipeline structure, we will also change the functional unit design by varying the number of units, the latency of operations, and the functional unit pipelining, so as to better explore these more advanced pipelining techniques.

\textbf{Dynamic Scheduling with a Scoreboard}

In a dynamically scheduled pipeline, all instructions pass through the issue stage in order (in-order issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out of order. \textit{Scoreboarding} is a technique for allowing instructions to execute out of order when there are sufficient resources and no data dependences; it is named after the CDC 6600 scoreboard, which developed this capability.

Before we see how scoreboarding could be used in the MIPS pipeline, it is important to observe that \textit{WAR} hazards, which did not exist in the MIPS floating-point or integer pipelines, may arise when instructions execute out of order. Suppose in the earlier example, the \texttt{SUBD} destination is \texttt{F8}, so that the code sequence is

\begin{verbatim}
DIV.D F0,F2,F4
ADD.D F10,F0,F8
SUB.D F8,F8,F14
\end{verbatim}

Now there is an antidependence between the \texttt{ADD.D} and the \texttt{SUB.D}: If the pipeline executes the \texttt{SUB.D} before the \texttt{ADD.D}, it will violate the antidependence, yielding incorrect execution. Likewise, to avoid violating output dependences, \textit{WAW} hazards (e.g., as would occur if the destination of the \texttt{SUB.D} were \texttt{F10}) must also be
detected. As we will see, both these hazards are avoided in a scoreboard by stalling the later instruction involved in the antidependence.

The goal of a scoreboard is to maintain an execution rate of one instruction per clock cycle (when there are no structural hazards) by executing an instruction as early as possible. Thus, when the next instruction to execute is stalled, other instructions can be issued and executed if they do not depend on any active or stalled instruction. The scoreboard takes full responsibility for instruction issue and execution, including all hazard detection. Taking advantage of out-of-order execution requires multiple instructions to be in their EX stage simultaneously. This can be achieved with multiple functional units, with pipelined functional units, or with both. Since these two capabilities—pipelined functional units and multiple functional units—are essentially equivalent for the purposes of pipeline control, we will assume the processor has multiple functional units.

The CDC 6600 had 16 separate functional units, including 4 floating-point units, 5 units for memory references, and 7 units for integer operations. On a processor for the MIPS architecture, scoreboards make sense primarily on the floating-point unit since the latency of the other functional units is very small. Let’s assume that there are two multipliers, one adder, one divide unit, and a single integer unit for all memory references, branches, and integer operations. Although this example is simpler than the CDC 6600, it is sufficiently powerful to demonstrate the principles without having a mass of detail or needing very long examples. Because both MIPS and the CDC 6600 are load-store architectures, the techniques are nearly identical for the two processors. Figure A.51 shows what the processor looks like.

Every instruction goes through the scoreboard, where a record of the data dependences is constructed; this step corresponds to instruction issue and replaces part of the ID step in the MIPS pipeline. The scoreboard then determines when the instruction can read its operands and begin execution. If the scoreboard decides the instruction cannot execute immediately, it monitors every change in the hardware and decides when the instruction can execute. The scoreboard also controls when an instruction can write its result into the destination register. Thus, all hazard detection and resolution is centralized in the scoreboard. We will see a picture of the scoreboard later (Figure E.52 on page E-80), but first we need to understand the steps in the issue and execution segment of the pipeline.

Each instruction undergoes four steps in executing. (Since we are concentrating on the FP operations, we will not consider a step for memory access.) Let’s first examine the steps informally and then look in detail at how the scoreboard keeps the necessary information that determines when to progress from one step to the next. The four steps, which replace the ID, EX, and WB steps in the standard MIPS pipeline, are as follows:

1. **Issue**—If a functional unit for the instruction is free and no other active instruction has the same destination register, the scoreboard issues the instruction to the functional unit and updates its internal data structure. This step
replaces a portion of the ID step in the MIPS pipeline. By ensuring that no other active functional unit wants to write its result into the destination register, we guarantee that WAW hazards cannot be present. If a structural or WAW hazard exists, then the instruction issue stalls, and no further instructions will issue until these hazards are cleared. When the issue stage stalls, it causes the buffer between instruction fetch and issue to fill; if the buffer is a single entry, instruction fetch stalls immediately. If the buffer is a queue with multiple instructions, it stalls when the queue fills; later we will see how a queue is used in the PowerPC 620 to connect fetch and issue.

2. *Read operands*—The scoreboard monitors the availability of the source operands. A source operand is available if no earlier issued active instruction is going to write it. When the source operands are available, the scoreboard tells the functional unit to proceed to read the operands from the registers and begin execution. The scoreboard resolves RAW hazards dynamically in this step,
and instructions may be sent into execution out of order. This step, together with issue, completes the function of the ID step in the simple MIPS pipeline.

3. **Execution**—The functional unit begins execution upon receiving operands. When the result is ready, it notifies the scoreboard that it has completed execution. This step replaces the EX step in the MIPS pipeline and takes multiple cycles in the MIPS FP pipeline.

4. **Write result**—Once the scoreboard is aware that the functional unit has completed execution, the scoreboard checks for WAR hazards and stalls the completing instruction, if necessary.

A WAR hazard exists if there is a code sequence like our earlier example with `ADD D` and `SUB D` that both use F8. In that example we had the code

```
DIV D F0, F2, F4
ADD D F10, F0, F8
SUB D F8, F8, F14
```

`ADD D` has a source operand F8, which is the same register as the destination of `SUB D`. But `ADD D` actually depends on an earlier instruction. The scoreboard will still stall the `SUB D` in its write result stage until `ADD D` reads its operands. In general, then, a completing instruction cannot be allowed to write its results when

- there is an instruction that has not read its operands that precedes (i.e., in order of issue) the completing instruction, and
- one of the operands is the same register as the result of the completing instruction.

If this WAR hazard does not exist, or when it clears, the scoreboard tells the functional unit to store its result to the destination register. This step replaces the WB step in the simple MIPS pipeline.

At first glance, it might appear that the scoreboard will have difficulty separating RAW and WAR hazards. Exercise 4.6 will help you understand how the scoreboard distinguishes these two cases and thus knows when to prevent a WAR hazard by stalling an instruction that is ready to write its results.

Because the operands for an instruction are read only when both operands are available in the register file, this scoreboard does not take advantage of forwarding. Instead registers are only read when they are both available. This is not as large a penalty as you might initially think. Unlike our simple pipeline of earlier, instructions will write their result into the register file as soon as they complete execution (assuming no WAR hazards), rather than wait for a statically assigned write slot that may be several cycles away. The effect is reduced pipeline latency and benefits of forwarding. There is still one additional cycle of latency that aris-
es since the write result and read operand stages cannot overlap. We would need additional buffering to eliminate this overhead.

Based on its own data structure, the scoreboard controls the instruction progression from one step to the next by communicating with the functional units. There is a small complication, however. There are only a limited number of source operand buses and result buses to the register file, which represents a structural hazard. The scoreboard must guarantee that the number of functional units allowed to proceed into steps 2 and 4 do not exceed the number of buses available. We will not go into further detail on this, other than to mention that the CDC 6600 solved this problem by grouping the 16 functional units together into four groups and supplying a set of buses, called *data trunks*, for each group. Only one unit in a group could read its operands or write its result during a clock.

Now let’s look at the detailed data structure maintained by a MIPS scoreboard with five functional units. Figure A.52 shows what the scoreboard’s information looks like part way through the execution of this simple sequence of instructions:

```
L.D F6,34(R2)
L.D F2,45(R3)
MULT.D F0,F2,F4
SUB.D F8,F6,F2
DIV.D F10,F0,F6
ADD.D F6,F8,F2
```

There are three parts to the scoreboard:

1. *Instruction status*—Indicates which of the four steps the instruction is in.
2. *Functional unit status*—Indicates the state of the functional unit (FU). There are nine fields for each functional unit:
   - Busy—Indicates whether the unit is busy or not.
   - Op—Operation to perform in the unit (e.g., add or subtract).
   - Fi—Destination register.
   - Fj, Fk—Source-register numbers.
   - Qi, Qk—Functional units producing source registers Fj, Fk.
   - Rj, Rk—Flags indicating when Fj, Fk are ready and not yet read. Set to No after operands are read.
3. *Register result status*—Indicates which functional unit will write each register, if an active instruction has the register as its destination. This field is set to blank whenever there are no pending instructions that will write that register.
Now let's look at how the code sequence begun in Figure A.52 continues execution. After that, we will be able to examine in detail the conditions that the scoreboard uses to control execution.

**EXAMPLE**  Assume the following EX cycle latencies (chosen to illustrate the behavior and not representative) for the floating-point functional units: Add is 2 clock
cycles, multiply is 10 clock cycles, and divide is 40 clock cycles. Using the code segment in Figure A.52 and beginning with the point indicated by the instruction status in Figure A.52, show what the status tables look like when MULTD and DIVD are each ready to go to the write-result state.

**ANSWER**

There are RAW data hazards from the second LD to MULT and SUBD, from MULT.D to DIV.D, and from SUB.D to ADD.D. There is a WAR data hazard between DIV.D and ADD.D. Finally, there is a structural hazard on the add functional unit for ADD.D. What the tables look like when MULT.D and DIV.D are ready to write their results is shown in Figures A.53 and A.54, respectively.

---

**FIGURE A.53**  Scoreboard tables just before the MULTD goes to write result. The DIVD has not yet read either of its operands, since it has a dependence on the result of the multiply. The ADD.D has read its operands and is in execution, although it was forced to wait until the SUBD finished to get the functional unit. ADD.D cannot proceed to write result because of the WAR hazard on F6, which is used by the DIV.D. The Q fields are only relevant when a functional unit is waiting for another unit.
Now we can see how the scoreboard works in detail by looking at what has to happen for the scoreboard to allow each instruction to proceed. Figure A.55 shows what the scoreboard requires for each instruction to advance and the bookkeeping action necessary when the instruction does advance. The scoreboard, like a number of other structures that we examine in this appendix, records operand specifier information, such as register numbers. For example, we must record the source registers when an instruction is issued. Because we refer to the contents of a register as Regs[D] where D is a register name, there is no ambiguity. For example, \( F_j[FU] \leftarrow S_1 \) causes the register name \( S_1 \) to be placed in \( F_j[FU] \), rather than the contents of the register of register \( S_1 \).

The costs and benefits of scoreboarding are interesting considerations. The CDC 6600 designers measured a performance improvement of 1.7 for FORTRAN programs and 2.5 for hand-coded assembly language. However, this was
measured in the days before software pipeline scheduling, semiconductor main memory, and caches (which lower memory-access time). The scoreboard on the CDC 6600 had about as much logic as one of the functional units, which is surprisingly low. The main cost was in the large number of buses—about four times as many as would be required if the CPU only executed instructions in order (or if it only initiated one instruction per execute cycle). The recently increasing interest in dynamic scheduling is motivated by attempts to issue more instructions per clock (so the cost of more buses must be paid anyway) and by ideas like speculation (explored in section 4.6) that naturally build on dynamic scheduling.

A scoreboard uses the available ILP to minimize the number of stalls arising from the program’s true data dependences. In eliminating stalls, a scoreboard is limited by several factors:

1. *The amount of parallelism available among the instructions*—This determines whether independent instructions can be found to execute. If each instruction depends on its predecessor, no dynamic scheduling scheme can reduce stalls. If the instructions in the pipeline simultaneously must be chosen from the same basic block (as was true in the 6600), this limit is likely to be quite severe.

2. *The number of scoreboard entries*—This determines how far ahead the pipeline can look for independent instructions. The set of instructions examined as candidates for potential execution is called the *window*. The size of the scoreboard determines the size of the window. In this section, we assume a window does not extend beyond a branch, so the window (and the scoreboard) always contains straight-line code from a single basic block. Section 4.6 shows how the window can be extended beyond a branch.

### FIGURE A.55 Required checks and bookkeeping actions for each step in instruction execution

<table>
<thead>
<tr>
<th>Instruction status</th>
<th>Wait until</th>
<th>Bookkeeping</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Issue</strong></td>
<td>Not Busy [FU] and not Result [D]</td>
<td>Busy[FU] ← yes; Op[FU] ← op; Fi[FU] ← D; Fj[FU] ← S1; Fk[FU] ← S2; Qj ← Result[S1]; Qk ← Result[S2]; Rj ← not Qj; Rk ← not Qk; Result[D] ← FU;</td>
</tr>
<tr>
<td><strong>Read operands</strong></td>
<td>Rj and Rk</td>
<td>Rj ← No; Rk ← No; Qj ← 0; Qk ← 0</td>
</tr>
<tr>
<td><strong>Execution complete</strong></td>
<td>Functional unit done</td>
<td></td>
</tr>
<tr>
<td><strong>Write result</strong></td>
<td>∀f((Fj[f] ≠ Fi[FU] or Rj[f] = No) &amp; (Fk[f] ≠ Fi[FU] or Rk[f] = No))</td>
<td>∀f(if Qj[f] = FU then Rj[f] ← Yes); ∀f(if Qk[f] = FU then Rk[f] ← Yes); Result[Fi[FU]] ← 0; Busy[FU] ← 0</td>
</tr>
</tbody>
</table>
3. *The number and types of functional units*—This determines the importance of structural hazards, which can increase when dynamic scheduling is used.

4. *The presence of antidependences and output dependences*—These lead to WAR and WAW stalls.

Chapters 3 and 4 focus on techniques that attack the problem of exposing and better utilizing available ILP. The second and third factors can be attacked by increasing the size of the scoreboard and the number of functional units; however, these changes have cost implications and may also affect cycle time. WAW and WAR hazards become more important in dynamically scheduled processors, because the pipeline exposes more name dependences. WAW hazards also become more important if we use dynamic scheduling with a branch prediction scheme that allows multiple iterations of a loop to overlap.

---

**A.9 Fallacies and Pitfalls**

*Pitfall: Unexpected execution sequences may cause unexpected hazards.*

At first glance, WAW hazards look like they should never occur in a code sequence 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
DIV.D F0,F2,F4 ; moved into delay slot
; from fall through
.....
.....

foo: L.D 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.

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 branches) and delays arising from RAW hazards. In gcc running on an R3000, which has a pipeline almost identical to that of Section A.1, 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.

A.10 Concluding Remarks

At the beginning of the 1980s, pipelining was a technique reserved primarily for supercomputer and large multimillion dollar mainframes. By the middle 1980s, the first pipelined microprocessors appeared and helped transform the world of computing, allowing microprocessors to bypass minicomputers in performance and eventually to take on and outperform mainframes. By the early 1990s, high-end embedded microprocessors embraced pipelining, and desktops were headed to wards the use of the sophisticated dynamically schedule, multiple issue approaches discussed in Chapter 3. The material in this chapter, which was considered reasonably advanced for graduate students when this text first appeared in 1990, is now considered basic undergraduate material and can be found in processors costing less than $10!
A.11 Historical Perspective and References

This section describes some of the major advances in pipelining through the CDC 6600. Chapters 3 and 4 expand on this basic history.

Early Pipelined CPUs

The first general-purpose pipelined processor 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.

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, in either designing pipelines or in creating software to schedule them.

The RISC processors 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 (see Figure 2.33 on page???). 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.

J. E. Smith and his colleagues have written a number of papers examining instruction issue, exception handling, and pipeline depth for high-speed scalar CPUs. 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 was one of the first deeply pipelined microprocessors and is described by Killian [1991] and by Heinrich [1993]. The initial Alpha implemen-
The 21064 also has a similar instruction set and similar integer pipeline structure, with more pipelining in the floating-point unit.

The Introduction of Dynamic Scheduling

In 1964 CDC delivered the first CDC 6600. The CDC 6600 was unique in many ways. In addition to introducing scoreboarding, the CDC 6600 was the first processor to make extensive use of multiple functional units. It also had peripheral processors that used multithreading. The interaction between pipelining and instruction set design was understood, and a simple, load/store instruction set was used to promote pipelining. The CDC 6600 also used an advanced packaging technology. Thornton [1964] describes the pipeline and I/O processor architecture, including the concept of out-of-order instruction execution. Thornton’s book [1970] provides an excellent description of the entire processor, from technology to architecture, and includes a foreword by Cray. (Unfortunately, this book is currently out of print.) The CDC 6600 also has an instruction scheduler for the FORTRAN compilers, described by Thorlin [1967].

References


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.


THORNTON, J. E. [1970]. *Design of a Computer, the Control Data 6600*, Scott, Foresman, Glenview, Ill.

**EXERCISES**