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.5, 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 *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 VAX 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.7 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 J). 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
MOVC3    @(R1)[R2],74(R2),R3    ;moves a character string
```

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.

A.5 Extending the MIPS Pipeline to Handle Multicycle Operations

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 1 clock cycle, or even in 2. 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 *initiation interval* or *repeat interval*. We define latency the same way we defined it earlier: the number of intervening cycles between an instruction that produces a result and an instruction that uses the result. The initiation or repeat interval is the number of cycles that must elapse between issuing two operations of a given type. For example, we will use the latencies and initiation intervals shown in Figure 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 1 cycle later. Hence the latency to a store for the value being stored, but not for the base address register, will be

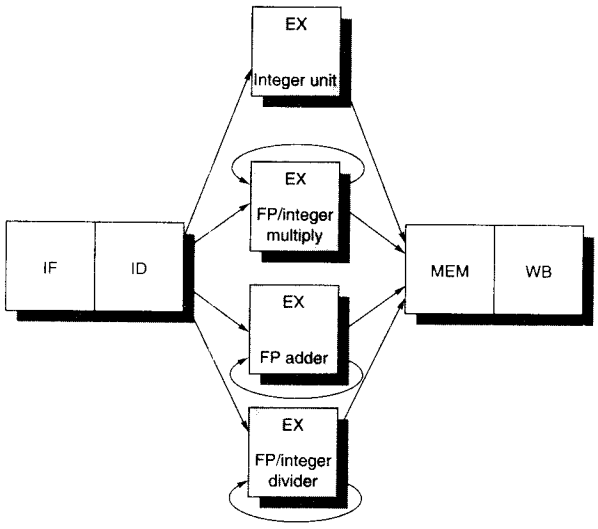

Figure A.29 The MIPS pipeline with three additional unpipelined, floating-point, functional units. Because only one instruction issues on every clock cycle, all instructions go through the standard pipeline for integer operations. The floating-point operations simply loop when they reach the EX stage. After they have finished the EX stage, they proceed to MEM and WB to complete execution.

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

Figure A.30 Latencies and initiation intervals for functional units.

1 cycle less. Pipeline latency is essentially equal to 1 cycle less than the depth of the execution pipeline, which is the number of stages from the EX stage to the stage that produces the result. Thus, for the example pipeline just above, the number of stages in an FP add is four, while the number of stages in an FP multiply is seven. To achieve a higher clock rate, designers need to put fewer logic levels in each pipe stage, which makes the number of pipe stages required for more complex operations larger. The penalty for the faster clock rate is thus longer latency for operations.

The example pipeline structure in Figure 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 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 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.

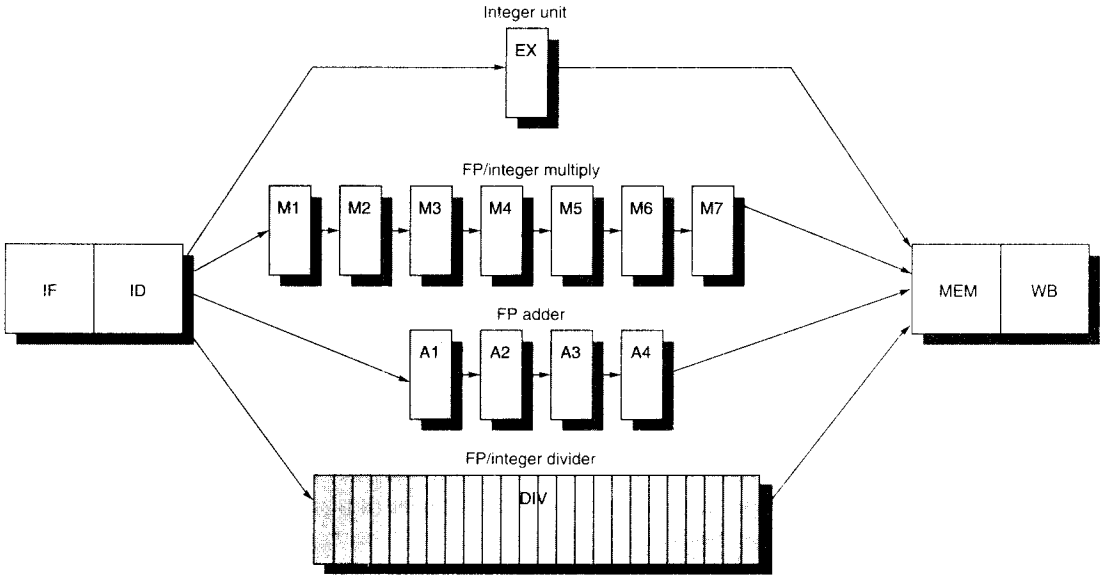

Figure A.31 A pipeline that supports multiple outstanding FP operations. The FP multiplier and adder are fully pipelined and have a depth of seven and four stages, respectively. The FP divider is not pipelined, but requires 24 clock cycles to complete. The latency in instructions between the issue of 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.

MUL.D	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
ADD.D		IF	ID	*A1*	A2	A3	**A4**	**MEM**	WB		
L.D			IF	ID	*EX*	**MEM**	WB				
S.D				IF	ID	*EX*	*MEM*	WB			

Figure A.32 The pipeline timing of a set of independent FP operations. The stages in italics show where data are 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.

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 earlier list. 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 A.34. In

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

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 S.D must be stalled an extra cycle so that its MEM does not conflict with the ADD.D. Extra hardware could easily handle this case.

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

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 MUL.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.

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.

There are two different ways to implement this interlock. The first is to track the use of the write port in the ID stage and to stall an instruction before it issues, just as we would for any other structural hazard. Tracking the use of the write port can be done with a shift register that indicates when already-issued instructions will use the register file. If the instruction in ID needs to use the register file at the same time as an instruction already issued, the instruction in ID is stalled for a cycle. On each clock the reservation register is shifted 1 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 A.34. If the `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 `ADD.D`. Note that this hazard only occurs when the result of the `ADD.D` is overwritten *without* any instruction ever using it! If there were a use of F2 between the `ADD.D` and the `L.D`, the pipeline would need to be stalled for a RAW hazard, and the `L.D` would not issue until the `ADD.D` 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 `L.D` appears in F2 when we are done. (As we will see in Section A.8, 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 `ADD.D` enters MEM. The second approach is to stamp out the result of the `ADD.D` by detecting the hazard and changing the control so that the `ADD.D` does not write its result. Then the `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 L.D is issuing. Then stalling the L.D or making the ADD.D a no-op is easy. The difficult situation is to detect that the L.D might finish before the ADD.D, because that requires knowing the length of the pipeline and the current position of the ADD.D. 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.7, 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.

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:

```
DIV.D     F0,F2,F4
ADD.D     F10,F10,F8
SUB.D     F12,F12,F14
```

This code sequence looks straightforward; there are no dependences. A problem arises, however, because an instruction issued early may complete after an instruction issued later. In this example, we can expect `ADD.D` and `SUB.D` to complete *before* the `DIV.D` completes. This is called *out-of-order completion* and is common in pipelines with long-running operations (see Section A.7). 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 `DIV.D` 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 `ADD.D` 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 Power1 and Power2, 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 autoincrement and autodecrement addressing on processors like VAXes. Another approach, the *future file*, proposed by Smith and 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 Chapter 2, we will see extensions of this idea, which are used in processors such as the PowerPC 620 and the 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 are 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 3 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 I.

Performance of a MIPS FP Pipeline

The MIPS FP pipeline of Figure A.31 on page A-50 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 convert). 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.

A.6 Putting It All Together: The MIPS R4000 Pipeline

In this section we look at the pipeline structure and performance of the MIPS R4000 processor family, which includes the 4400. The R4000 implements MIPS64 but uses a deeper pipeline than that of our five-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

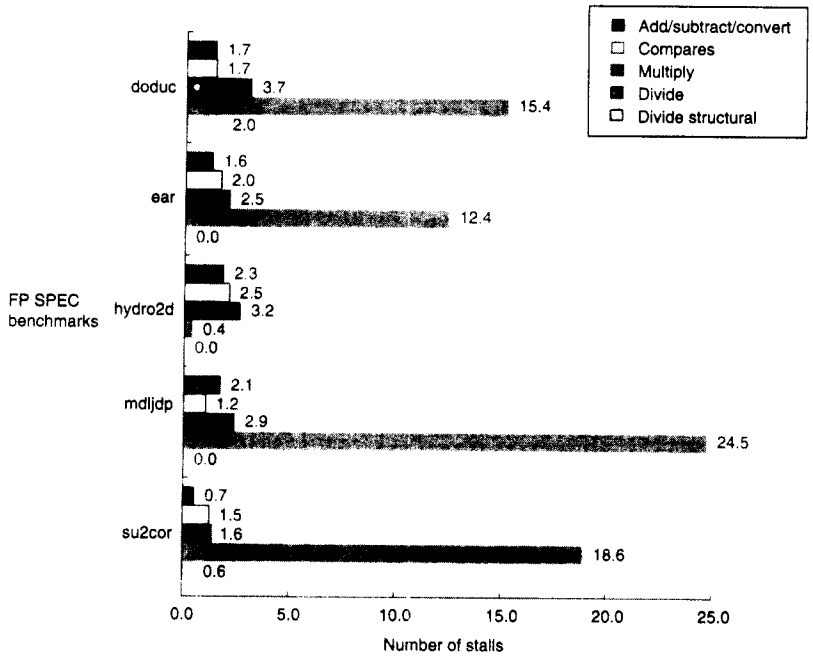

Figure A.35 Stalls per FP operation for each major type of FP operation for the SPEC89 FP benchmarks. Except for the divide structural hazards, these data do not depend on the frequency of an operation, only on its latency and the number of cycles before the result is used. The number of stalls from 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.

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 data path. 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.

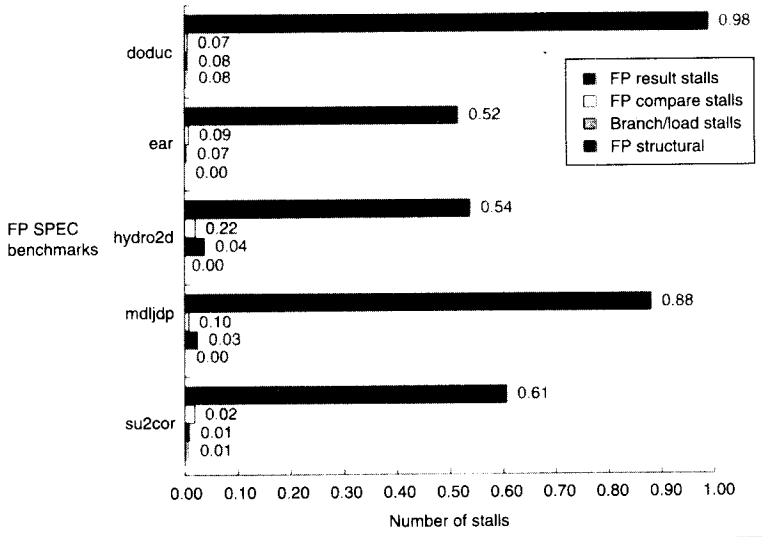

Figure A.36 The stalls occurring for the MIPS FP pipeline for five of the SPEC89 FP benchmarks. The total number of stalls per instruction ranges from 0.65 for su2cor to 1.21 for doduc, with an average of 0.87. FP result stalls dominate in all cases, with an average of 0.71 stalls per instruction, or 82% of the stalled cycles. Compares generate an average of 0.1 stalls per instruction and are the second largest source. The divide structural hazard is only significant for doduc.

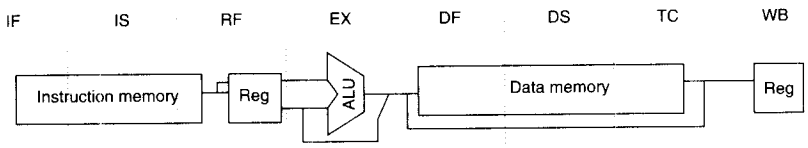

Figure A.37 The eight-stage pipeline structure of the R4000 uses pipelined instruction and data caches. The pipe stages are labeled and their detailed function is described in the text. The vertical dashed lines represent the stage boundaries as well as the location of pipeline latches. The instruction is actually available at the end of IS, but the tag check is done in RF, while the registers are fetched. Thus, we show the instruction memory as operating through RF. The TC stage is needed for data memory access, since we cannot write the data into the register until we know whether the cache access was a hit or not.

- 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.

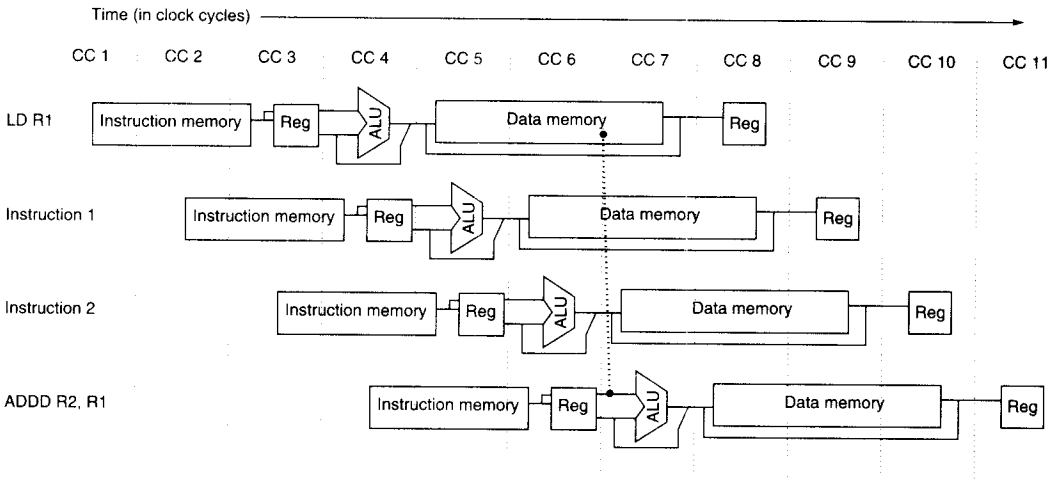

Figure A.38 The structure of the R4000 integer pipeline leads to a 2-cycle load delay. A 2-cycle delay is possible because the data value is available at the end of DS and can be bypassed. If the tag check in TC indicates a miss, the pipeline is backed up a cycle, when the correct data are available.

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

Figure A.39 A load instruction followed by an immediate use results in a 2-cycle stall. Normal forwarding paths can be used after two cycles, so the DADD and DSUB get the value by forwarding after the stall. The OR instruction gets the value from the register file. Since the two instructions after the load could be independent and hence not stall, the bypass can be to instructions that are 3 or 4 cycles after the load.

- 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 2 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 3 or 4 cycles later.

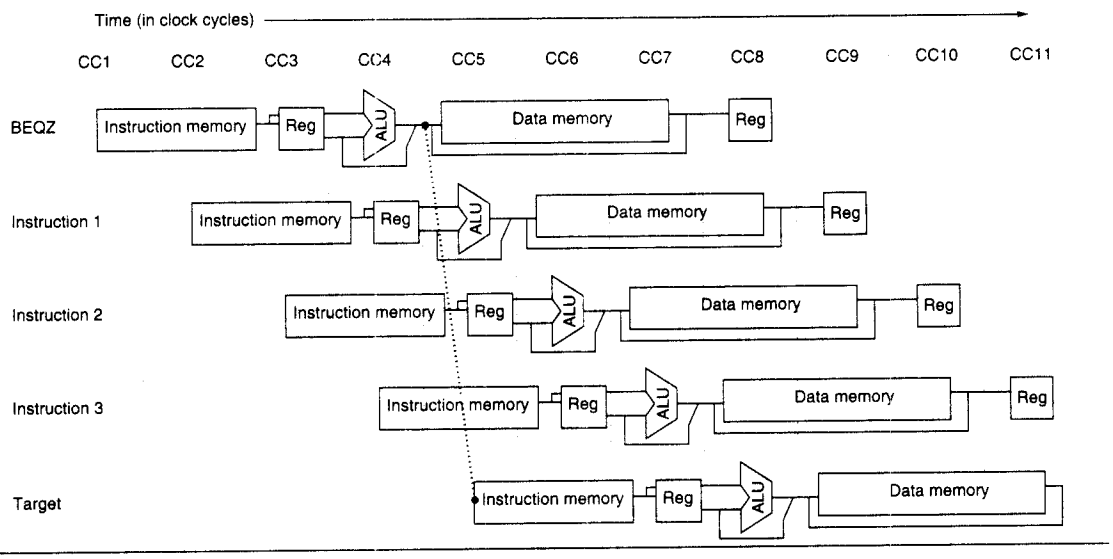

Figure A.40 The basic branch delay is 3 cycles, since the condition evaluation is performed during EX.

Figure A.40 shows that the basic branch delay is 3 cycles, since the branch condition is computed during EX. The MIPS architecture has a single-cycle delayed branch. The R4000 uses a predicted-not-taken strategy for the remaining 2 cycles of the branch delay. As Figure A.41 shows untaken branches are simply 1-cycle delayed branches, while taken branches have a 1-cycle delay slot followed by 2 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 2-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 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 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 2 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.

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

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

Figure A.41 A taken branch, shown in the top portion of the figure, has a 1-cycle delay slot followed by a 2-cycle stall, while an untaken branch, shown in the bottom portion, has simply a 1-cycle delay slot. The branch instruction can be an ordinary delayed branch or a branch-likely, which cancels the effect of the instruction in the delay slot if the branch is untaken.

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

Figure A.42 The eight stages used in the R4000 floating-point pipelines.

There is a single copy of each of these stages, and various instructions may use a stage zero or more times and in different orders. Figure 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

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

Figure A.43 The latencies and initiation intervals for the FP operations both depend on the FP unit stages that a given operation must use. The latency values assume that the destination instruction is an FP operation; the latencies are 1 cycle less when the destination is a store. The pipe stages are shown in the order in which they are used for any operation. The notation S + A indicates a clock cycle in which both the S and A stages are used. The notation D^{28} indicates that the D stage is used 28 times in a row.

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

Figure 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 4 or 5 cycles after the multiply; otherwise, it issues without stalling. Notice that the add will be stalled for 2 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 1 clock cycle, since that will eliminate the conflicts.

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.

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

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.

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

Figure 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 28–33. Notice the add starting in cycle 28 will be stalled until cycle 36. 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.

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 1 or 2 cycles after the load

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

Figure A.47 A double-precision add is followed by a double-precision divide. If the divide starts 1 cycle after the add, the divide stalls, but after that there is no conflict.

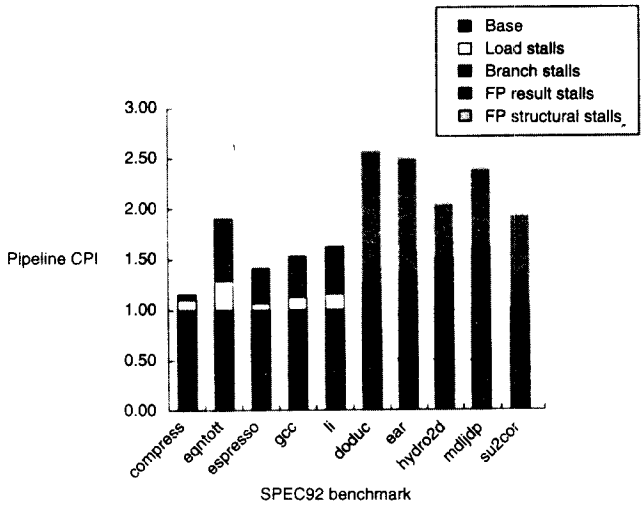

Figure A.48 The pipeline CPI for 10 of the SPEC92 benchmarks, assuming a perfect cache. The pipeline CPI varies from 1.2 to 2.8. The leftmost five programs are integer programs, and branch delays are the major CPI contributor for these. The rightmost five programs are FP, and FP result stalls are the major contributor for these. Figure A.49 shows the numbers used to construct this plot.

2. *Branch stalls*—2-cycle stall on every taken branch plus unfilled or canceled 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.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.

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

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

Figure A.49 The total pipeline CPI and the contributions of the four major sources of stalls are shown. The major contributors are FP result stalls (both for branches and for FP inputs) and branch stalls, with loads and FP structural stalls adding less.

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 Crosscutting 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 2) or using dynamic hardware scheduling techniques (which we discuss next and in further detail in Chapter 2). 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 2 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 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 scoreboarding 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 2.

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 that 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 properly, it was issued from ID. To allow an instruction to begin execution as soon as its operands are available, even if a predecessor is stalled, we must separate the issue process into two parts: checking the structural hazards and waiting for the absence of a data hazard. We decode and issue instructions in order. However, we want the instructions to begin execution as soon as their data operands are available. Thus, the pipeline will do *out-of-order execution*, which implies *out-of-order completion*. To implement out-of-order execution, we must split the ID pipe stage into two stages:

1. *Issue*—Decode instructions, check for structural hazards.
2. *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.

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. *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 WAR hazards, which did not exist in the MIPS floating-point or integer pipelines, may arise when instructions execute out of order. For example, consider the following code sequence:

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

There is an antidependence between the `ADD.D` and the `SUB.D`: If the pipeline executes the `SUB.D` before the `ADD.D`, it will violate the antidependence, yielding incorrect execution. Likewise, to avoid violating output dependences, WAW hazards (e.g., as would occur if the destination of the `SUB.D` were `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.50 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

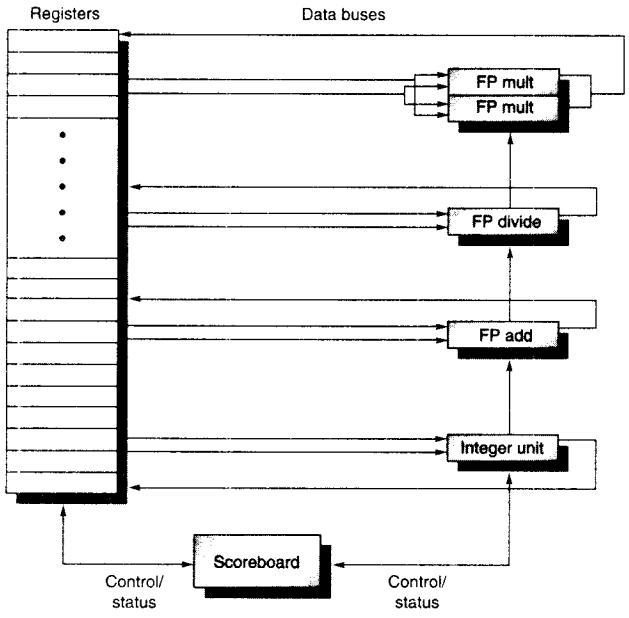

Figure A.50 The basic structure of a MIPS processor with a scoreboard. The scoreboard's function is to control instruction execution (vertical control lines). All data flows between the register file and the functional units over the buses (the horizontal lines, called trunks in the CDC 6600). There are two FP multipliers, an FP divider, an FP adder, and an integer unit. One set of buses (two inputs and one output) serves a group of functional units. The details of the scoreboard are shown in Figures A.51–A.54.

see a picture of the scoreboard later (Figure A.51 on page A-71), 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.
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.

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 arises 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.51 shows what the scoreboard's information looks like partway through the execution of this simple sequence of instructions:

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

		Instruction status			
Instruction		Issue	Read operands	Execution complete	Write result
L.D	F6,34(R2)	√	√	√	√
L.D	F2,45(R3)	√	√	√	
MUL.D	F0,F2,F4	√			
SUB.D	F8,F6,F2	√			
DIV.D	F10,F0,F6	√			
ADD.D	F6,F8,F2				

	Functional unit status								
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	Yes	Load	F2	R3				No	
Mult1	Yes	Mult	F0	F2	F4	Integer		No	Yes
Mult2	No								
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

	Register result status								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult1	Integer			Add	Divide			

Figure A.51 Components of the scoreboard. Each instruction that has issued or is pending issue has an entry in the instruction status table. There is one entry in the functional unit status table for each functional unit. Once an instruction issues, the record of its operands is kept in the functional unit status table. Finally, the register result table indicates which unit will produce each pending result; the number of entries is equal to the number of registers. The instruction status table says that (1) the first L.D has completed and written its result, and (2) the second L.D has completed execution but has not yet written its result. The MUL.D, SUB.D, and DIV.D have all issued but are stalled, waiting for their operands. The functional unit status says that the first multiply unit is waiting for the integer unit, the add unit is waiting for the integer unit, and the divide unit is waiting for the first multiply unit. The ADD.D instruction is stalled because of a structural hazard; it will clear when the SUB.D completes. If an entry in one of these scoreboard tables is not being used, it is left blank. For example, the Rk field is not used on a load and the Mult2 unit is unused, hence their fields have no meaning. Also, once an operand has been read, the Rj and Rk fields are set to No. Figure A.54 shows why this last step is crucial.

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.
- Qj, 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.51 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.51 and beginning with the point indicated by the instruction status in Figure A.51, show what the status tables look like when `MUL.D` and `DIV.D` are each ready to go to the Write Result state.

Answer There are RAW data hazards from the second `L.D` to `MUL.D`, `ADD.D`, and `SUB.D`, from `MUL.D` to `DIV.D`, and from `SUB.D` to `ADD.D`. There is a WAR data hazard between `DIV.D` and `ADD.D` and `SUB.D`. Finally, there is a structural hazard on the add functional unit for `ADD.D` and `SUB.D`. What the tables look like when `MUL.D` and `DIV.D` are ready to write their results is shown in Figures A.52 and A.53, respectively.

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.54 shows what the scoreboard requires for each instruction to advance and the bookkeeping action necessary when the instruction does advance. The scoreboard 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, `Fj[FU]` ← S1 causes the register *name* S1 to be placed in `Fj[FU]`, rather than the *contents* of register S1.

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 sur-

	Instruction status			
Instruction	Issue	Read operands	Execution complete	Write result
L.D F6,34(R2)	√	√	√	√
L.D F2,45(R3)	√	√	√	√
MUL.D F0,F2,F4	√	√	√	
SUB.D F8,F6,F2	√	√	√	√
DIV.D F10,F0,F6	√			
ADD.D F6,F8,F2	√	√	√	

	Functional unit status								
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	Yes	Mult	F0	F2	F4			No	No
Mult2	No								
Add	Yes	Add	F6	F8	F2			No	No
Divide	Yes	Div	F10	F0	F6	Mult1		No	Yes

	Register result status								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU	Mult 1			Add		Divide			

Figure A.52 Scoreboard tables just before the MUL.D goes to write result. The DIV.D 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 SUB.D 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.

prisingly 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 4.7) 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:

	Instruction status			
Instruction	Issue	Read operands	Execution complete	Write result
L.D F6,34(R2)	√	√	√	√
L.D F2,45(R3)	√	√	√	√
MUL.D F0,F2,F4	√	√	√	√
SUB.D F8,F6,F2	√	√	√	√
DIV.D F10,F0,F6	√	√	√	
ADD.D F6,F8,F2	√	√	√	√

	Functional unit status								
Name	Busy	Op	Fi	Fj	Fk	Qj	Qk	Rj	Rk
Integer	No								
Mult1	No								
Mult2	No								
Add	No								
Divide	Yes	Div	F10	F0	F6			No	No

	Register result status								
	F0	F2	F4	F6	F8	F10	F12	...	F30
FU						Divide			

Figure A.53 Scoreboard tables just before the DIV.D goes to write result. ADD.D was able to complete as soon as DIV.D passed through read operands and got a copy of F6. Only the DIV.D remains to finish.

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. Chapter 2 shows how the window can be extended beyond a branch.

Instruction status	Wait until	Bookkeeping
Issue	Not Busy [FU] and not Result [D]	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;
Read operands	Rj and Rk	Rj← No; Rk← No; Qj←0; Qk←0
Execution complete	Functional unit done	
Write result	$\forall f$((Fj[f] ≠ Fi[FU] or Rj[f] = No) & (Fk[f] ≠ Fi[FU] or Rk[f] = No))	$\forall f$(if Qj[f]=FU then Rj[f]←Yes); $\forall f$(if Qk[f]=FU then Rk[f]←Yes); Result[Fi[FU]]← 0; Busy[FU]← No

Figure A.54 Required checks and bookkeeping actions for each step in instruction execution. FU stands for the functional unit used by the instruction, D is the destination register name, S1 and S2 are the source register names, and op is the operation to be done. To access the scoreboard entry named Fj for functional unit FU we use the notation Fj[FU]. Result[D] is the name of the functional unit that will write register D. The test on the write result case prevents the write when there is a WAR hazard, which exists if another instruction has this instruction's destination (Fi[FU]) as a source (Fj[f] or Fk[f]) and if some other instruction has written the register (Rj = Yes or Rk = Yes). The variable f is used for any functional unit.

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 2 and 3 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.8 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 `DIV.D` can complete, the `L.D` will reach WB, causing a WAW hazard. The hardware must detect this and may stall the issue of the `L.D`. 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 dynamic or static scheduling 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 compile time scheduler or run time dynamic scheduling, you must use optimized code, since in the real system you will derive good performance from other optimizations in addition to scheduling.

A.9 Concluding Remarks

At the beginning of the 1980s, pipelining was a technique reserved primarily for supercomputers and large multimillion dollar mainframes. By the mid-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 toward the use of the sophisticated dynamically scheduled, multiple-issue approaches discussed in Chapter 2. The material in this appendix, 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.10 Historical Perspective and References

Section K.4 on the companion CD features a discussion on the development of pipelining and instruction-level parallelism. We provide numerous references for further reading and exploration of these topics.