Data Dependences

There are three different types of dependences: data dependences (also called true data dependences), name dependences, and control dependences. An instruction j is *data dependent* on instruction i if either of the following holds:

- instruction i produces a result that may be used by instruction j, or
- instruction j is data dependent on instruction k, and instruction k is data dependent on instruction i.

The second condition simply states that one instruction is dependent on another if there exists a chain of dependences of the first type between the two instructions. This dependence chain can be as long as the entire program.

For example, consider the following code sequence that increments a vector of values in memory (starting at 0(R1) and with the last element at 8(R2)) by a scalar in register F2:

```
Loop:   L.D     F0,0(R1)    ;F0=array element
        ADD.D   F4,F0,F2    ;add scalar in F2
        S.D     F4,0(R1)    ;store result
        DADDUI  R1,R1,#-8   ;decrement pointer 8 bytes
        BNE     R1,R2,LOOP  ;branch R1!=R2
```

The data dependences in this code sequence involve both floating-point data:

```
Loop:   L.D     F0,0(R1)    ;F0=array element
        ADD.D   F4,F0,F2    ;add scalar in F2
        S.D     F4,0(R1)    ;store result
```

and integer data:

```
        DADDIU  R1,R1,-8    ;decrement pointer
                            ;8 bytes (per DW)
        BNE     R1,R2,Loop  ;branch R1!=zero
```

Both of the above dependent sequences, as shown by the arrows, have each instruction depending on the previous one. The arrows here and in following examples show the order that must be preserved for correct execution. The arrow points from an instruction that must precede the instruction that the arrowhead points to.

If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped. The dependence implies that there would be a chain of one or more data hazards between the two instructions. Executing the instructions

simultaneously will cause a processor with pipeline interlocks to detect a hazard and stall, thereby reducing or eliminating the overlap. In a processor without interlocks that relies on compiler scheduling, the compiler cannot schedule dependent instructions in such a way that they completely overlap, since the program will not execute correctly. The presence of a data dependence in an instruction sequence reflects a data dependence in the source code from which the instruction sequence was generated. The effect of the original data dependence must be preserved.

Dependences are a property of *programs*. Whether a given dependence results in an actual hazard being detected and whether that hazard actually causes a stall are properties of the *pipeline organization*. This difference is critical to understanding how instruction-level parallelism can be exploited.

In our example, there is a data dependence between the `DADDIU` and the `BNE`; this dependence causes a stall because we moved the branch test for the MIPS pipeline to the ID stage. Had the branch test stayed in EX, this dependence would not cause a stall. Of course, the branch delay would then still be two cycles, rather than one.

The presence of the dependence indicates the potential for a hazard, but the actual hazard and the *length of any stall* is a property of the pipeline. The importance of the data dependences is that a dependence (1) indicates the possibility of a hazard, (2) determines the order in which results must be calculated, and (3) sets an upper bound on how much parallelism can possibly be exploited. Such limits are explored in Section 3.8.

Since a data dependence can limit the amount of instruction-level parallelism we can exploit, a major focus of this chapter and the next is overcoming these limitations. A dependence can be overcome in two different ways: maintaining the dependence but avoiding a hazard, and eliminating a dependence by transforming the code. Scheduling the code is the primary method used to avoid a hazard without altering a dependence. In this chapter, we consider hardware schemes for scheduling code dynamically as it is executed. As we will see, some types of dependences can be eliminated, primarily by software, and in some cases by hardware techniques.

A data value may flow between instructions either through registers or through memory locations. When the data flow occurs in a register, detecting the dependence is reasonably straightforward since the register names are fixed in the instructions, although it gets more complicated when branches intervene and correctness concerns cause a compiler or hardware to be conservative.

Dependences that flow through memory locations are more difficult to detect since two addresses may refer to the same location but look different: For example, `100(R4)` and `20(R6)` may be identical. In addition, the effective address of a load or store may change from one execution of the instruction to another (so that `20(R4)` and `20(R4)` will be different), further complicating the detection of a dependence. In this chapter, we examine hardware for detecting data dependences that involve memory locations, but we will see that these techniques also have limitations. The compiler techniques for detecting such dependences are critical in uncovering loop-level parallelism, as we will see in the next chapter.

Name Dependences

The second type of dependence is a *name dependence*. A name dependence occurs when two instructions use the same register or memory location, called a *name*, but there is no flow of data between the instructions associated with that name. There are two types of name dependences between an instruction i that *precedes* instruction j in program order:

1. An *antidependence* between instruction i and instruction j occurs when instruction j writes a register or memory location that instruction i reads. The original ordering must be preserved to ensure that i reads the correct value.
2. An *output dependence* occurs when instruction i and instruction j write the same register or memory location. The ordering between the instructions must be preserved to ensure that the value finally written corresponds to instruction j.

Both antidependences and output dependences are name dependences, as opposed to true data dependences, since there is no value being transmitted between the instructions. Since a name dependence is not a true dependence, instructions involved in a name dependence can execute simultaneously or be reordered, if the name (register number or memory location) used in the instructions is changed so the instructions do not conflict. This renaming can be more easily done for register operands, where it is called *register renaming*. Register renaming can be done either statically by a compiler or dynamically by the hardware. Before describing dependences arising from branches, let's examine the relationship between dependences and pipeline data hazards.

Data Hazards

A hazard is created whenever there is a dependence between instructions, and they are close enough that the overlap caused by pipelining, or other reordering of instructions, would change the order of access to the operand involved in the dependence. Because of the dependence, we must preserve what is called *program order*, that is, the order that the instructions would execute in if executed sequentially one at a time as determined by the original source program. The goal of both our software and hardware techniques is to exploit parallelism by preserving program order *only where it affects the outcome of the program*. Detecting and avoiding hazards ensures that necessary program order is preserved.

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

- RAW *(read after write)*—j tries to read a source before i writes it, so j incorrectly gets the *old* value. This hazard is the most common type and corresponds to a true data dependence. Program order must be preserved to ensure

that *j* receives the value from *i*. In the simple common five-stage static pipeline (see Appendix A), a load instruction followed by an integer ALU instruction that directly uses the load result will lead to a RAW hazard.

- WAW *(write after write)*—*j* tries to write an operand before it is written by *i*. The writes end up being performed in the wrong order, leaving the value written by *i* rather than the value written by *j* in the destination. This hazard corresponds to an output dependence. WAW hazards are present only in pipelines that write in more than one pipe stage or allow an instruction to proceed even when a previous instruction is stalled. The classic five-stage integer pipeline used in Appendix A writes a register only in the WB stage and avoids this class of hazards, but this chapter explores pipelines that allow instructions to be reordered, creating the possibility of WAW hazards. WAW hazards can also exist between a short integer pipeline and a longer floating-point pipeline (see the pipelines in Sections A.5 and A.6 of Appendix A). For example, a floating-point multiply instruction that writes F4, shortly followed by a load of F4, could yield a WAW hazard, since the load could complete before the multiply completed.
- WAR *(write after read)*—*j* tries to write a destination before it is read by *i*, so *i* incorrectly gets the *new* value. This hazard arises from an antidependence. WAR hazards cannot occur in most static issue pipelines—even deeper pipelines or floating-point pipelines—because all reads are early (in ID) and all writes are late (in WB). (See Appendix A to convince yourself.) A WAR hazard occurs either when there are some instructions that write results early in the instruction pipeline *and* other instructions that read a source late in the pipeline, or when instructions are reordered, as we will see in this chapter.

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

Control Dependences

The last type of dependence is a *control dependence*. A control dependence determines the ordering of an instruction, *i*, with respect to a branch instruction so that the instruction *i* is executed in correct program order and only when it should be. Every instruction, except for those in the first basic block of the program, is control dependent on some set of branches, and, in general, these control dependences must be preserved to preserve program order. One of the simplest examples of a control dependence is the dependence of the statements in the "then" part of an if statement on the branch. For example, in the code segment

```
if p1 {
      S1;
};
if p2 {
      S2;
}
```

S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.

In general, there are two constraints imposed by control dependences:

1. An instruction that is control dependent on a branch cannot be moved *before* the branch so that its execution *is no longer controlled* by the branch. For example, we cannot take an instruction from the then portion of an if statement and move it before the if statement.
2. An instruction that is not control dependent on a branch cannot be moved *after* the branch so that its execution *is controlled* by the branch. For example, we cannot take a statement before the if statement and move it into the then portion.

Control dependence is preserved by two properties in a simple pipeline, such as that in Chapter 1. First, instructions execute in program order. This ordering ensures that an instruction that occurs before a branch is executed before the branch. Second, the detection of control or branch hazards ensures that an instruction that is control dependent on a branch is not executed until the branch direction is known.

Although preserving control dependence is a useful and simple way to help preserve program order, the control dependence in itself is not the fundamental performance limit. We may be willing to execute instructions that should not have been executed, thereby violating the control dependences, *if* we can do so without affecting the correctness of the program. Control dependence is not the critical property that must be preserved. Instead, the two properties critical to program correctness—and normally preserved by maintaining both data and control dependence—are the *exception behavior* and the *data flow*.

Preserving the exception behavior means that any changes in the ordering of instruction execution must not change how exceptions are raised in the program. Often this is relaxed to mean that the reordering of instruction execution must not cause any new exceptions in the program. A simple example shows how maintaining the control and data dependences can prevent such situations. Consider this code sequence:

```
        DADDU   R2,R3,R4
        BEQZ    R2,L1
        LW      R1,0(R2)
L1:
```

In this case, it is easy to see that if we do not maintain the data dependence involving R2, we can change the result of the program. Less obvious is the fact that if we ignore the control dependence and move the load instruction before the branch, the load instruction may cause a memory protection exception. Notice that *no data dependence* prevents us from interchanging the BEQZ and the LW; it is only the control dependence. To allow us to reorder these instructions (and still preserve the data dependence), we would like to just ignore the exception when

the branch is taken. In Section 3.7, we will look at a hardware technique, speculation, which allows us to overcome this exception problem. The next chapter looks at other techniques for the same problem.

The second property preserved by maintenance of data dependences and control dependences is the data flow. The data flow is the actual flow of data values among instructions that produce results and those that consume them. Branches make the data flow dynamic, since they allow the source of data for a given instruction to come from many points. Put another way, it is not sufficient to just maintain data dependences because an instruction may be data dependent on more than one predecessor. Program order is what determines which predecessor will actually deliver a data value to an instruction. Program order is ensured by maintaining the control dependences.

For example, consider the following code fragment:

```
        DADDU    R1,R2,R3
        BEQZ     R4,L
        DSUBU    R1,R5,R6
L:      ...

        OR       R7,R1,R8
```

In this example, the value of R1 used by the OR instruction depends on whether the branch is taken or not. Data dependence alone is not sufficient to preserve correctness. The OR instruction is data dependent on both the DADDU and DSUBU instructions, but preserving this order alone is insufficient for correct execution. Instead, when the instructions execute, the data flow must be preserved: If the branch is not taken, then the value of R1 computed by the DSUBU should be used by the OR, and if the branch is taken, the value of R1 computed by the DADDU should be used by the OR. By preserving the control dependence of the OR on the branch, we prevent an illegal change to the data flow. For similar reasons, the DSUBU instruction cannot be moved above the branch. Speculation, which helps with the exception problem, will also allow us to lessen the impact of the control dependence while still maintaining the data flow, as we will see in Section 3.7.

Sometimes we can determine that violating the control dependence cannot affect either the exception behavior or the data flow. Consider the following code sequence:

```
            DADDU    R1,R2,R3
            BEQZ     R12,skipnext
            DSUBU    R4,R5,R6
            DADDU    R5,R4,R9
skipnext:   OR       R7,R8,R9
```

Suppose we knew that the register destination of the DSUBU instruction (R4) was unused after the instruction labeled skipnext. (The property of whether a value will be used by an upcoming instruction is called *liveness*.) If R4 were unused,

then changing the value of R4 just before the branch would not affect the data flow since R4 would be *dead* (rather than live) in the code region after `skipnext`. Thus, if R4 were dead and the existing DSUBU instruction could not generate an exception (other than those from which the processor resumes the same process), we could move the DSUBU instruction before the branch, since the data flow cannot be affected by this change. If the branch is taken, the DSUBU instruction will execute and will be useless, but it will not affect the program results. This type of code scheduling is sometimes called *speculation*, since the compiler is betting on the branch outcome; in this case, the bet is that the branch is usually not taken. More ambitious compiler speculation mechanisms are discussed in Chapter 4.

Control dependence is preserved by implementing control hazard detection that causes control stalls. Control stalls can be eliminated or reduced by a variety of hardware and software techniques. Delayed branches, which we saw in Chapter 1, can reduce the stalls arising from control hazards; scheduling a delayed branch requires that the compiler preserve the data flow.

The key focus of the rest of this chapter is on techniques that exploit instruction-level parallelism using hardware. The data dependences in a compiled program act as a limit on how much ILP can be exploited. The challenge is to approach that limit by trying to minimize the actual hazards and associated stalls that arise. The techniques we examine become ever more sophisticated in an attempt to exploit all the available parallelism while maintaining the necessary true data dependences in the code.

3.2 Overcoming Data Hazards with Dynamic Scheduling

A simple statically scheduled pipeline fetches an instruction and issues it, unless there was 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 a data dependence that cannot be hidden, 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.

In this section, we explore an important technique, called *dynamic scheduling*, in which the hardware rearranges the instruction execution to reduce the stalls while maintaining data flow and exception behavior. Dynamic scheduling offers several advantages: It enables handling some cases when dependences are unknown at compile time (e.g., because they may involve a memory reference), and it simplifies the compiler. Perhaps most importantly, it also allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline. In Section 3.7, we will explore hardware speculation, a technique with significant performance advantages, which builds on dynamic scheduling. As we will see, the advantages of dynamic scheduling are gained at a cost of a significant increase in hardware complexity.

Although a dynamically scheduled processor cannot change the data flow, it tries to avoid stalling when dependences, which could generate hazards, are present. In contrast, static pipeline scheduling by the compiler (covered in the next chapter) tries to minimize stalls by separating dependent instructions so that they will not lead to hazards. Of course, compiler pipeline scheduling can also be used on code destined to run on a processor with a dynamically scheduled pipeline.

Dynamic Scheduling: The Idea

A major limitation of the simple pipelining techniques we discuss in Appendix A is that they all use in-order instruction issue and execution: Instructions are issued in program order, and if an instruction is stalled in the pipeline, no later instructions can proceed. Thus, if there is a dependence between two closely spaced instructions in the pipeline, this will lead to a hazard and a stall will result. If there are multiple functional units, these units could lie idle. If instruction *j* depends on a long-running instruction *i*, currently in execution in the pipeline, then all instructions after *j* must be stalled until *i* is finished and *j* can execute. For example, consider this code:

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

The `SUB.D` instruction cannot execute because the dependence of `ADD.D` on `DIV.D` causes the pipeline to stall; yet `SUB.D` is not data dependent on anything in the pipeline. This hazard creates a performance limitation that can be eliminated by not requiring instructions to execute in program order.

In the classic five-stage pipeline developed in the first chapter, both structural and data hazards could be checked during instruction decode (ID): When an instruction could execute without hazards, it was issued from ID knowing that all data hazards had been resolved. To allow us to begin executing the `SUB.D` in the above example, we must separate the issue process into two parts: checking for any 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 (i.e., instructions issued in program order), but we want an instruction to begin execution as soon as its data operand is available. Thus, this pipeline does *out-of-order execution,* which implies *out-of-order completion.*

Out-of-order execution introduces the possibility of WAR and WAW hazards, which do not exist in the five-stage integer pipeline and its logical extension to an in-order floating-point pipeline. Consider the following MIPS floating-point code sequence:

```
DIV.D     F0,F2,F4
ADD.D     F6,F0,F8
SUB.D     F8,F10,F14
MUL.D     F6,F10,F8
```

There is an antidependence between the ADD.D and the SUB.D, and if the pipeline executes the SUB.D before the ADD.D (which is waiting for the DIV.D), it will violate the antidependence, yielding a WAR hazard. Likewise, to avoid violating output dependences, such as the write of F6 by MUL.D, WAW hazards must be handled. As we will see, both these hazards are avoided by the use of register renaming..

Out-of-order completion also creates major complications in handling exceptions. Dynamic scheduling with out-of-order completion must preserve exception behavior in the sense that *exactly* those exceptions that would arise if the program were executed in strict program order *actually* do arise. Dynamically scheduled processors preserve exception behavior by ensuring that no instruction can generate an exception until the processor knows that the instruction raising the exception will be executed; we will see shortly how this property can be guaranteed. Although exception behavior must be preserved, dynamically scheduled processors may generate *imprecise* exceptions. An exception is *imprecise* if the processor state when an exception is raised does not look exactly as if the instructions were executed sequentially in strict program order. Imprecise exceptions can occur because of two possibilities:

1. The pipeline may have *already completed* instructions that are *later* in program order than the instruction causing the exception.
2. The pipeline may have *not yet completed* some instructions that are *earlier* in program order than the instruction causing the exception.

Imprecise exceptions make it difficult to restart execution after an exception. Rather than address these problems in this section, we will discuss a solution that provides precise exceptions in the context of a processor with speculation in Section 3.7. For floating-point exceptions, other solutions have been used, as discussed in Appendix A.

To allow out-of-order execution, we essentially split the ID pipe stage of our simple five-stage pipeline into two stages:

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

An instruction fetch stage precedes the issue stage and may fetch either into an instruction register or into a queue of pending instructions; instructions are then issued from the register or queue. The EX stage follows the read operands stage, just as in the five-stage pipeline. Execution may take multiple cycles, depending on the operation.

We will distinguish when an instruction *begins execution* and when it *completes execution;* between the two times, the instruction is *in execution.* Our pipeline allows multiple instructions to be in execution at the same time, and without this capability, a major advantage of dynamic scheduling is lost. Having multiple instructions in execution at once requires multiple functional units, pipelined functional units, or 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.

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. We focus on a more sophisticated technique, called *Tomasulo's algorithm*, that has several major enhancements over scoreboarding. The reader wishing a gentler introduction to these concepts may want to consult Appendix A, which thoroughly discusses scoreboarding and includes several examples.

Dynamic Scheduling Using Tomasulo's Approach

A key approach to allow execution to proceed in the presence of dependences was used by the IBM 360/91 floating-point unit. Invented by Robert Tomasulo, this scheme tracks when operands for instructions are available, to minimize RAW hazards, and introduces register renaming, to minimize WAW and RAW hazards. There are many variations on this scheme in modern processors, although the key concept of tracking instruction dependences to allow execution as soon as operands are available and renaming registers to avoid WAR and WAW hazards are common characteristics.

The IBM 360/91 was completed just before caches appeared in commercial processors. IBM's goal was to achieve high floating-point performance from an instruction set and from compilers designed for the entire 360 computer family, rather than from specialized compilers for the high-end processors. The 360 architecture had only four double-precision floating-point registers, which limits the effectiveness of compiler scheduling; this fact was another motivation for the Tomasulo approach. In addition, the IBM 360/91 had long memory accesses and long floating-point delays, which Tomasulo's algorithm was designed to overcome. At the end of the section, we will see that Tomasulo's algorithm can also support the overlapped execution of multiple iterations of a loop.

We explain the algorithm, which focuses on the floating-point unit and load-store unit, in the context of the MIPS instruction set. The primary difference between MIPS and the 360 is the presence of register-memory instructions in the latter processor. Because Tomasulo's algorithm uses a load functional unit, no significant changes are needed to add register-memory addressing modes. The IBM 360/91 also had pipelined functional units, rather than multiple functional units, but we describe the algorithm as if there were multiple functional units. It is a simple conceptual extension to also pipeline those functional units.

As we will see, RAW hazards are avoided by executing an instruction only when its operands are available. WAR and WAW hazards, which arise from name dependences, are eliminated by register renaming. *Register renaming* eliminates these hazards by renaming all destination registers, including those with a pend-

ing read or write for an earlier instruction, so that the out-of-order write does not affect any instructions that depend on an earlier value of an operand.

To better understand how register renaming eliminates WAR and WAW hazards, consider the following example code sequence that includes both a potential WAR and WAW hazard:

```
DIV.D     F0,F2,F4
ADD.D     F6,F0,F8
S.D       F6,0(R1)
SUB.D     F8,F10,F14
MUL.D     F6,F10,F8
```

There is an antidependence between the ADD.D and the SUB.D and an output dependence between the ADD.D and the MUL.D, leading to two possible hazards: a WAR hazard on the use of F8 by ADD.D and a WAW hazard since the ADD.D may finish later than the MUL.D. There are also three true data dependences: between the DIV.D and the ADD.D, between the SUB.D and the MUL.D, and between the ADD.D and the S.D.

These name dependences can both be eliminated by register renaming. For simplicity, assume the existence of two temporary registers, S and T. Using S and T, the sequence can be rewritten without any dependences as

```
DIV.D     F0,F2,F4
ADD.D     S,F0,F8
S.D       S,0(R1)
SUB.D     T,F10,F14
MUL.D     F6,F10,T
```

In addition, any subsequent uses of F8 must be replaced by the register T. In this code segment, the renaming process can be done statically by the compiler. Finding any uses of F8 that are later in the code requires either sophisticated compiler analysis or hardware support, since there may be intervening branches between the above code segment and a later use of F8. As we will see, Tomasulo's algorithm can handle renaming across branches.

In Tomasulo's scheme, register renaming is provided by the *reservation stations*, which buffer the operands of instructions waiting to issue, and by the issue logic. The basic idea is that a reservation station fetches and buffers an operand as soon as it is available, eliminating the need to get the operand from a register. In addition, pending instructions designate the reservation station that will provide their input. Finally, when successive writes to a register overlap in execution, only the last one is actually used to update the register. As instructions are issued, the register specifiers for pending operands are renamed to the names of the reservation station, which provides register renaming. Since there can be more reservation stations than real registers, the technique can even eliminate hazards arising from name dependences that could not be eliminated by a compiler. As we

explore the components of Tomasulo's scheme, we will return to the topic of register renaming and see exactly how the renaming occurs and how it eliminates WAR and WAW hazards.

The use of reservation stations, rather than a centralized register file, leads to two other important properties. First, hazard detection and execution control are distributed: The information held in the reservation stations at each functional unit determine when an instruction can begin execution at that unit. Second, results are passed directly to functional units from the reservation stations where they are buffered, rather than going through the registers. This bypassing is done with a common result bus that allows all units waiting for an operand to be loaded simultaneously (on the 360/91 this is called the *common data bus,* or CDB). In pipelines with multiple execution units and issuing multiple instructions per clock, more than one result bus will be needed.

Figure 3.2 shows the basic structure of a Tomasulo-based MIPS processor, including both the floating-point unit and the load-store unit; none of the execution control tables are shown. Each reservation station holds an instruction that has been issued and is awaiting execution at a functional unit, and either the operand values for that instruction, if they have already been computed, or else the names of the reservation stations that will provide the operand values.

The load buffers and store buffers hold data or addresses coming from and going to memory and behave almost exactly like reservation stations, so we distinguish them only when necessary. The floating-point registers are connected by a pair of buses to the functional units and by a single bus to the store buffers. All results from the functional units and from memory are sent on the common data bus, which goes everywhere except to the load buffer. All reservation stations have tag fields, employed by the pipeline control.

Before we describe the details of the reservation stations and the algorithm, let's look at the steps an instruction goes through, just as we did for the five-stage pipeline of Appendix A. Since the structure is dramatically different, there are only three steps (though each one can now take an arbitrary number of clock cycles):

1. *Issue*—Get the next instruction from the head of the instruction queue, which is maintained in FIFO order to ensure the maintenance of correct data flow. If there is a matching reservation station that is empty, issue the instruction to the station with the operand values, if they are currently in the registers. If there is not an empty reservation station, then there is a structural hazard and the instruction stalls until a station or buffer is freed. If the operands are not in the registers, keep track of the functional units that will produce the operands. This step renames registers, eliminating WAR and WAW hazards.
2. *Execute*—If one or more of the operands is not yet available, monitor the common data bus while waiting for it to be computed. When an operand becomes available, it is placed into the corresponding reservation station. When all the operands are available, the operation can be executed at the corresponding functional unit. By delaying instruction execution until the operands are available, RAW hazards are avoided. Notice that several instructions

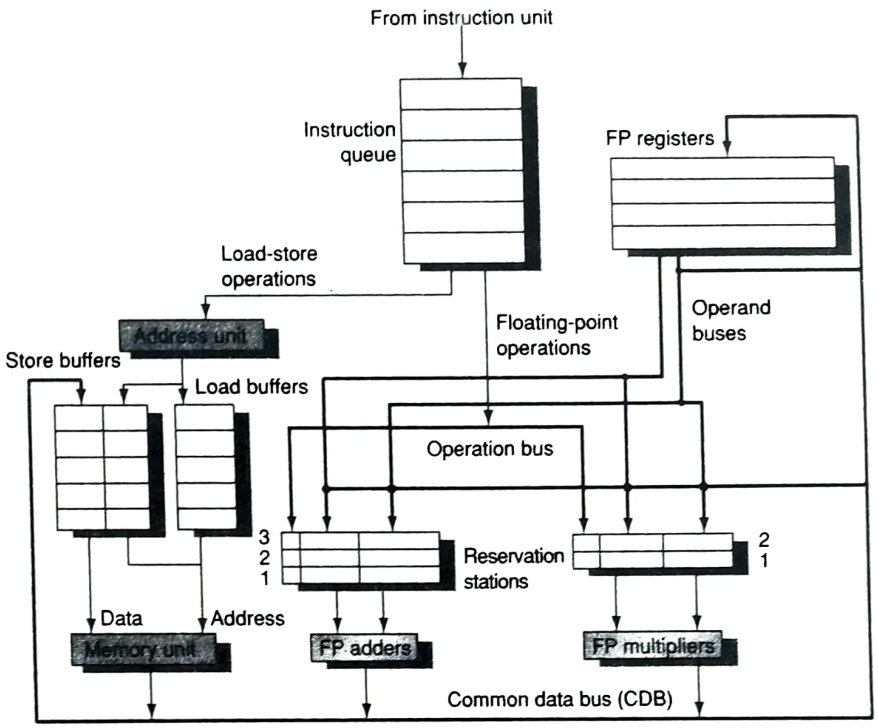

Figure 3.2 The basic structure of a MIPS floating-point unit using Tomasulo's algorithm. Instructions are sent from the instruction unit into the instruction queue from which they are issued in FIFO order. The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards. Load buffers have three functions: hold the components of the effective address until it is computed, track outstanding loads that are waiting on the memory, and hold the results of completed loads that are waiting for the CDB. Similarly, store buffers have three functions: hold the components of the effective address until it is computed, hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and hold the address and value to store until the memory unit is available. All results from either the FP units or the load unit are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers. The FP adders implement addition and subtraction, and the FP multipliers do multiplication and division.

could become ready in the same clock cycle for the same functional unit. Although independent functional units could begin execution in the same clock cycle for different instructions, if more than one instruction is ready for a single functional unit, the unit will have to choose among them. For the floating-point reservation stations, this choice may be made arbitrarily; loads and stores, however, present an additional complication.

Loads and stores require a two-step execution process. The first step computes the effective address when the base register is available, and the effective address is then placed in the load or store buffer. Loads in the load buffer execute as soon as the memory unit is available. Stores in the store buffer wait for the value to be stored before being sent to the memory unit. Loads and

stores are maintained in program order through the effective address calculation, which will help to prevent hazards through memory, as we will see shortly.

To preserve exception behavior, no instruction is allowed to initiate execution until all branches that precede the instruction in program order have completed. This restriction guarantees that an instruction that causes an exception during execution really would have been executed. In a processor using branch prediction (as all dynamically scheduled processors do), this means that the processor must know that the branch prediction was correct before allowing an instruction after the branch to begin execution. It is possible by recording the occurrence of the exception, but not actually raising it, to allow execution of the instruction to start and not stall the instruction until it enters Write Result. As we will see, speculation provides a more flexible and more complete method to handle exceptions, so we will delay making this enhancement and show how speculation handles this problem later.

3. *Write result*—When the result is available, write it on the CDB and from there into the registers and into any reservation stations (including store buffers) waiting for this result. Stores also write data to memory during this step: When both the address and data value are available, they are sent to the memory unit and the store completes.

The data structures used to detect and eliminate hazards are attached to the reservation stations, to the register file, and to the load and store buffers with slightly different information attached to different objects. These tags are essentially names for an extended set of virtual registers used in renaming. In our example, the tag field is a 4-bit quantity that denotes one of the five reservation stations or one of the six load buffers. As we will see, this produces the equivalent of 11 registers that can be designated as result registers (as opposed to the 4 double-precision registers that the 360 architecture contains). In a processor with more real registers, we would want renaming to provide an even larger set of virtual registers. The tag field describes which reservation station contains the instruction that will produce a result needed as a source operand.

Once an instruction has issued and is waiting for a source operand, it refers to the operand by the reservation station number where the instruction that will write the register has been assigned. Unused values, such as zero, indicate that the operand is already available in the registers. Because there are more reservation stations than actual register numbers, WAW and WAR hazards are eliminated by renaming results using reservation station numbers. Although in Tomasulo's scheme the reservation stations are used as the extended virtual registers, other approaches could use a register set with additional registers or a structure like the reorder buffer, which we will see in Section 3.7.

In describing the operation of this scheme, we use a terminology taken from the CDC scoreboard scheme, showing the terminology used by the IBM 360/91 for historical reference. It is important to remember that the tags in the Tomasulo scheme refer to the buffer or unit that will produce a result; the register names are discarded when an instruction issues to a reservation station.

Each reservation station has seven fields:

- Op—The operation to perform on source operands S1 and S2.
- Qj, Qk—The reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary. (The IBM 360/91 calls these SINKunit and SOURCEunit.)
- Vj, Vk—The value of the source operands. Note that only one of the V field or the Q field is valid for each operand. For loads, the Vk field is used to hold the offset field. (These fields are called SINK and SOURCE on the IBM 360/91.)
- A–Used to hold information for the memory address calculation for a load or store. Initially, the immediate field of the instruction is stored here; after the address calculation, the effective address is stored here.
- Busy—Indicates that this reservation station and its accompanying functional unit are occupied.

The register file has a field, Qi:

- Qi—The number of the reservation station that contains the operation whose result should be stored into this register. If the value of Qi is blank (or 0), no currently active instruction is computing a result destined for this register, meaning that the value is simply the register contents.

The load and store buffers each have a field, A, which holds the result of the effective address once the first step of execution has been completed.

In the next section, we will first consider some examples that show how these mechanisms work and then examine the detailed algorithm.

3.3 Dynamic Scheduling: Examples and the Algorithm

Before we examine Tomasulo's algorithm in detail, let's consider a few examples, which will help illustrate how the algorithm works.

Example Show what the information tables look like for the following code sequence when only the first load has completed and written its result:

```
1.    L.D     F6,34(R2)
2.    L.D     F2,45(R3)
3.    MUL.D   F0,F2,F4
4.    SUB.D   F8,F2,F6
5.    DIV.D   F10,F0,F6
6.    ADD.D   F6,F8,F2
```

Answer The result is shown in the three tables in Figure 3.3. The numbers appended to the names add, mult, and load stand for the tag for that reservation station—Add1 is the tag for the result from the first add unit. In addition we have included an instruction status table. This table is included only to help you understand the algorithm; it is *not* actually a part of the hardware. Instead, the reservation station keeps the state of each operation that has issued.

		Instruction status		
Instruction		**Issue**	**Execute**	**Write Result**
`L.D`	`F6,34(R2)`	√	√	√
`L.D`	`F2,45(R3)`	√	√	
`MUL.D`	`F0,F2,F4`	√		
`SUB.D`	`F8,F2,F6`	√		
`DIV.D`	`F10,F0,F6`	√		
`ADD.D`	`F6,F8,F2`	√		

	Reservation stations						
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**A**
Load1	no						
Load2	yes	Load					45 + Regs[R3]
Add1	yes	SUB		Mem[34 + Regs[R2]]	Load2		
Add2	yes	ADD			Add1	Load2	
Add3	no						
Mult1	yes	MUL		Regs[F4]	Load2		
Mult2	yes	DIV		Mem[34 + Regs[R2]]	Mult1		

	Register status								
Field	**F0**	**F2**	**F4**	**F6**	**F8**	**F10**	**F12**	...	**F30**
Qi	Mult1	Load2		Add2	Add1	Mult2			

Figure 3.3 Reservation stations and register tags shown when all of the instructions have issued, but only the first load instruction has completed and written its result to the CDB. The second load has completed effective address calculation, but is waiting on the memory unit. We use the array Regs[ ] to refer to the register file and the array Mem[ ] to refer to the memory. Remember that an operand is specified by either a Q field or a V field at any time. Notice that the `ADD.D` instruction, which has a WAR hazard at the WB stage, has issued and could complete before the `DIV.D` initiates.

Tomasulo's scheme offers two major advantages over earlier and simpler schemes: (1) the distribution of the hazard detection logic and (2) the elimination of stalls for WAW and WAR hazards.

The first advantage arises from the distributed reservation stations and the use of the CDB. If multiple instructions are waiting on a single result, and each instruction already has its other operand, then the instructions can be released simultaneously by the broadcast on the CDB. If a centralized register file were used, the units would have to read their results from the registers when register buses are available.

The second advantage, the elimination of WAW and WAR hazards, is accomplished by renaming registers using the reservation stations, and by the process of storing operands into the reservation station as soon as they are available. For example, in our code sequence in Figure 3.3 we have issued both the `DIV.D` and the `ADD.D`, even though there is a WAR hazard involving F6. The hazard is eliminated in one of two ways. First, if the instruction providing the value for the `DIV.D` has completed, then Vk will store the result, allowing `DIV.D` to execute independent of the `ADD.D` (this is the case shown).

On the other hand, if the `L.D` had not completed, then Qk would point to the Load1 reservation station, and the `DIV.D` instruction would be independent of the `ADD.D`. Thus, in either case, the `ADD.D` can issue and begin executing. Any uses of the result of the `DIV.D` would point to the reservation station, allowing the `ADD.D` to complete and store its value into the registers without affecting the `DIV.D`. We'll see an example of the elimination of a WAW hazard shortly. But let's first look at how our earlier example continues execution. In this example, and the ones that follow in this chapter, assume the following latencies: Load is 1 clock cycle, Add is 2 clock cycles, multiply is 10 clock cycles, and divide is 40 clock cycles.

Example Using the same code segment as in the previous example (page 189), show what the status tables look like when the `MUL.D` is ready to write its result.

Answer The result is shown in the three tables in Figure 3.4. Notice that `ADD.D` has completed since the operands of `DIV.D` were copied, thereby overcoming the WAR hazard. Notice that even if the load of F6 was delayed, the add into F6 could be executed without triggering a WAW hazard.

Tomasulo's Algorithm: The Details

Figure 3.5 gives the checks and steps that each instruction must go through. As mentioned earlier, loads and stores go through a functional unit for effective address computation before proceeding to independent load or store buffers. Loads take a second execution step to access memory and then go to Write Result to send the value from memory to the register file and/or any waiting reservation stations. Stores complete their execution in the Write Result stage, which writes the result to memory. Notice that all writes occur in Write Result, whether the

If we predict that branches are taken, using reservation stations will allow multiple executions of this loop to proceed at once. This advantage is gained without changing the code—in effect, the loop is unrolled dynamically by the hardware, using the reservation stations obtained by renaming to act as additional registers.

Let's assume we have issued all the instructions in two successive iterations of the loop, but none of the floating-point load-stores or operations has completed. The reservation stations, register status tables, and load and store buffers at this point are shown in Figure 3.6. (The integer ALU operation is ignored, and it is assumed the branch was predicted as taken.) Once the system reaches this state, two copies of the loop could be sustained with a CPI close to 1.0, provided

		Instruction status			
Instruction		From iteration	Issue	Execute	Write Result
L.D	F0,0(R1)	1	√	√	
MUL.D	F4,F0,F2	1	√		
S.D	F4,0(R1)	1	√		
L.D	F0,0(R1)	2	√	√	
MUL.D	F4,F0,F2	2	√		
S.D	F4,0(R1)	2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	yes	Load					Regs[R1] + 0
Load2	yes	Load					Regs[R1] – 8
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	MUL		Regs[F2]	Load1		
Mult2	yes	MUL		Regs[F2]	Load2		
Store1	yes	Store	Regs[R1]			Mult1	
Store2	yes	Store	Regs[R1] – 8			Mult2	

	Register status								
Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Load2		Mult2						

Figure 3.6 Two active iterations of the loop with no instruction yet completed. Entries in the multiplier reservation stations indicate that the outstanding loads are the sources. The store reservation stations indicate that the multiply destination is the source of the value to store.