“Who’s first?”

“America.”

“Who’s second?”

“Sir, there is no second.”

Dialog between two observers of the sailing race later named “The America’s Cup” and run every few years.

This quote was the inspiration for John Cocke’s naming of the IBM research processor as “America.” This processor was the precursor to the RS/6000 series and the first superscalar microprocessor.
All processors since about 1985, including those in the embedded space, use pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called instruction-level parallelism (ILP) since the instructions can be evaluated in parallel. In this chapter and the next, we look at a wide range of techniques for extending the pipelining ideas by increasing the amount of parallelism exploited among instructions. This chapter is at a considerably more advanced level than the material in Appendix A. If you are not familiar with the ideas in Appendix A, you should review that Appendix before venturing into this chapter.

We start this chapter by looking at the limitation imposed by data and control hazards and then turn to the topic of increasing the ability of the processor to exploit parallelism. Section 3.1 introduces a large number of concepts, which we build on throughout these two chapters. While some of the more basic material in
this chapter could be understood without all of the ideas in Section 3.1, this basic material is important to later sections of this chapter as well as to chapter 4.

There are two largely separable approaches to exploiting ILP. This chapter covers techniques that are largely dynamic and depend on the hardware to locate the parallelism. The next chapter focuses on techniques that are static and rely much more on software. In practice, this partitioning between dynamic and static and between hardware-intensive and software-intensive is not clean, and techniques from one camp are often used by the other. Nonetheless, for exposition purposes, we have separated the two approaches and tried to indicate where an approach is transferable.

The dynamic, hardware intensive approaches dominate the desktop and server markets and are used in a wide range of processors, including: the Pentium III and 4, the Althon, the MIPS R10000/12000, the Sun ultraSPARC III, the PowerPC 603, G3, and G4, and the Alpha 21264. The static, compiler-intensive approaches, which we focus on in the next chapter, have seen broader adoption in the embedded market than the desktop or server markets, although the new IA-64 architecture and Intel’s Itanium, use this more static approach.

In this section, we discuss features of both programs and processors that limit the amount of parallelism that can be exploited among instructions, as well as the critical mapping between program structure and hardware structure, which is key to understanding whether a program property will actually limit performance and under what circumstances.

Recall that the value of the CPI (Cycles per Instruction) for a pipelined processor is the sum of the base CPI and all contributions from stalls:

\[
\text{Pipeline CPI} = \text{Ideal pipeline CPI} + \text{Structural stalls} + \text{Data hazard stalls} + \text{Control stalls}
\]

The ideal pipeline CPI is a measure of the maximum performance attainable by the implementation. By reducing each of the terms of the right-hand side, we minimize the overall pipeline CPI and thus increase the IPC (Instructions per Clock). In this chapter we will see that the techniques we introduce to increase the ideal IPC, can increase the importance of dealing with structural, data hazard, and control stalls. The equation above allows us to characterize the various techniques we examine in this chapter by what component of the overall CPI a technique reduces. Figure 3.1 shows the techniques we examine in this chapter and in the next, as well as the topics covered in the introductory material in Appendix A.

Before we examine these techniques in detail, we need to define the concepts on which these techniques are built. These concepts, in the end, determine the limits on how much parallelism can be exploited.

**Instruction-Level Parallelism**

All the techniques in this chapter and the next exploit parallelism among instructions. As we stated above, this type of parallelism is called instruction-level parallelism or ILP. The amount of parallelism available within a basic block—a straight-
line code sequence with no branches in except to the entry and no branches out except at the exit—is quite small. For typical MIPS programs the average dynamic branch frequency often between 15% and 25%, meaning that between four and seven instructions execute between a pair of branches. Since these instructions are likely to depend upon one another, the amount of overlap we can exploit within a basic block is likely to be much less than the average basic block size. To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks.

The simplest and most common way to increase the amount of parallelism available among instructions is to exploit parallelism among iterations of a loop. This type of parallelism is often called loop-level parallelism. Here is a simple example of a loop, which adds two 1000-element arrays, that is completely parallel:

```
for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + y[i];
```

Every iteration of the loop can overlap with any other iteration, although within each loop iteration there is little or no opportunity for overlap.

There are a number of techniques we will examine for converting such loop-level parallelism into instruction-level parallelism. Basically, such techniques work by unrolling the loop either statically by the compiler (an approach we explore in the next chapter) or dynamically by the hardware (the subject of this chapter).

<table>
<thead>
<tr>
<th>Technique</th>
<th>Reduces</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forwarding and bypassing</td>
<td>Potential data hazard stalls</td>
<td>A.2</td>
</tr>
<tr>
<td>Delayed branches and simple branch scheduling</td>
<td>Control hazard stalls</td>
<td>A.2</td>
</tr>
<tr>
<td>Basic dynamic scheduling (scoreboarding)</td>
<td>Data hazard stalls from true dependences</td>
<td>A.8</td>
</tr>
<tr>
<td>Dynamic scheduling with renaming</td>
<td>Data hazard stalls and stalls from antidependences and output dependences</td>
<td>3.2</td>
</tr>
<tr>
<td>Dynamic branch prediction</td>
<td>Control stalls</td>
<td>3.4</td>
</tr>
<tr>
<td>Issuing multiple instructions per cycle</td>
<td>Ideal CPI</td>
<td>3.6</td>
</tr>
<tr>
<td>Speculation</td>
<td>Data hazard and control hazard stalls</td>
<td>3.5</td>
</tr>
<tr>
<td>Dynamic memory disambiguation</td>
<td>Data hazard stalls with memory</td>
<td>3.2, 3.7</td>
</tr>
<tr>
<td>Loop unrolling</td>
<td>Control hazard stalls</td>
<td>4.1</td>
</tr>
<tr>
<td>Basic compiler pipeline scheduling</td>
<td>Data hazard stalls</td>
<td>A.2, 4.1</td>
</tr>
<tr>
<td>Compiler dependence analysis</td>
<td>Ideal CPI, data hazard stalls</td>
<td>4.4</td>
</tr>
<tr>
<td>Software pipelining, trace scheduling</td>
<td>Ideal CPI, data hazard stalls</td>
<td>4.3</td>
</tr>
<tr>
<td>Compiler speculation</td>
<td>Ideal CPI, data, control stalls</td>
<td>4.4</td>
</tr>
</tbody>
</table>

**FIGURE 3.1** The major techniques examined in Appendix A, chapter 3, or chapter 4 are shown together with the component of the CPI equation that the technique affects.
An important alternative method for exploiting loop-level parallelism is the use of vector instructions (see Appendix B). Essentially, a vector instruction operates on a sequence of data items. For example, the above code sequence could execute in four instructions on some vector processors: two instructions to load the vectors $x$ and $y$ from memory, one instruction to add the two vectors, and an instruction to store back the result vector. Of course, these instructions would be pipelined and have relatively long latencies, but these latencies may be overlapped. Vector instructions and the operation of vector processors are described in detail in the online Appendix B. Although the development of the vector ideas preceded many of the techniques we examine in these two chapters for exploiting ILP, processors that exploit ILP have almost completely replaced vector-based processors. Vector instruction sets, however, may see a renaissance, at least for use in graphics, digital signal processing, and multimedia applications.

Data Dependence and Hazards

Determining how one instruction depends on another is critical to determining how much parallelism exists in a program and how that parallelism can be exploited. In particular, to exploit instruction-level parallelism we must determine which instructions can be executed in parallel. If two instructions are parallel, they can execute simultaneously in a pipeline without causing any stalls, assuming the pipeline has sufficient resources (and hence no structural hazards exist). If two instructions are dependent they are not parallel and must be executed in order, though they may often be partially overlapped. The key in both cases is to determine whether an instruction is dependent on another instruction.

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:
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
DADDUI R1,R1,#-8;decrement pointer 8 bytes (/e
BNE R1,R2,LOOP; branch R1!=zero
```

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, with 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 2 cycles, rather than 1.
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, \(100 \text{ (R4)}\) and \(20 \text{ (R6)}\) 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 \(20 \text{ (R4)}\) and \(20 \text{ (R4)}\) 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 shall see that these techniques also have limitations. The compiler techniques for detecting such dependences are critical in uncovering loop-level parallelism, as we shall 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 be 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 dependencies 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:

```c
if p1 {
    S1;
} else {
    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 in-
3.1 Instruction-Level Parallelism: Concepts and Challenges

A 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 3.5, 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:
```

...
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 DAAU 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 3.5.

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 ex-
Overcoming Data Hazards with Dynamic Scheduling

A simple satirically 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 3.5, 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 instruc-
tions after \(j\) must be stalled until \(i\) is finished and \(j\) can execute. For example, consider this code:

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

The \text{SUB.D} instruction cannot execute because the dependence of \text{ADD.D} on \text{DIV.D} causes the pipeline to stall; yet \text{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 \text{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 issue in program order), but we want an instruction to begin execution as soon as its data operand is available. Thus, this pipeline does \textit{out-of-order execution}, which implies \textit{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:

\[
\begin{align*}
\text{DIV.D} & \quad F0, F2, F4 \\
\text{ADD.D} & \quad F6, F0, F8 \\
\text{SUB.D} & \quad F8, F10, F14 \\
\text{MULT.D} & \quad F6, F10, F8 \\
\end{align*}
\]

There is an antidependence between the \text{ADD.D} and the \text{SUB.D}, and if the pipeline executes the \text{SUB.D} before the \text{ADD.D} (which is waiting for the \text{DIV.D}), it will violate the antidependence, yielding a WAR hazard. Likewise, to avoid violating output dependences, such as the write of \(F6\) by \text{MULT.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 \textit{exactly} those exceptions that would arise if the program were executed in strict program order \textit{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 \textit{imprecise} exceptions. An exception is \textit{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, and
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 3.5. 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 the online version of Appendix G that thoroughly discusses scoreboard 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, though the key concept of tracking instruction dependencies 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 pending 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
MULT.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 `MULT.D` leading to three possible hazards: a WAR hazard on the use of `F8` by `ADD.D` and on the use of `F8` by the `MULT.D`, and a WAW hazard since the `ADD.D` may finish later than the `MULT.D`. There are also three true data dependences between the `DIV.D` and the `ADD.D`, between the `SUB.D` and the `MULT.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:

```plaintext
DIV.D F0,F2,F4
ADD.D S,F0,F8
S.D S,0(R1)
SUB.D T,F10,F14
MULT.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 functional units that will be 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 Chapter 1. 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, enter the functional units that will produce the operands into the Qi and Qj fields. 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 (CDB) 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 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 from 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 schedule 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 four-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 eleven registers that can be designated as result registers (as opposed to the four 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 3.5.

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 six 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 the offset from the instruction. (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 as 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
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.

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.
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 cycle, Add is 2 clock cycles, multiply is 10 clock cycles, and divide is 40 clock cycles.

**EXAMPLE**  Using the same code segment as the previous example (page 239), 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 destination is a register or memory. This restriction simplifies Tomasulo's algorithm and is critical to its extension with speculation in section 3.5.
<table>
<thead>
<tr>
<th>Instruction state</th>
<th>Wait until</th>
<th>Action or bookkeeping</th>
</tr>
</thead>
</table>
| Issue             | Station $r$ empty | if (Register Stat[$rs$].Qi ≠ 0)  
{RS[$r$].Qj ← RegisterStat[$rs$].Qi}  
else {RS[$r$].Vj ← Regs[$rs$]; RS[$r$].Qj ← 0};  
if (RegisterStat[$rt$].Qi ≠ 0)  
{RS[$r$].Qk ← RegisterStat[$rt$].Q.i}  
else {RS[$r$].Vk ← Regs[$rt$]; RS[$r$].Qk ← 0};  
RS[$r$].Busy ← yes; RegisterStat[$rd$].Qi ← $r$; |
| Load or Store     | Buffer $r$ empty | if (Register Stat[$rs$].Qi ≠ 0)  
{RS[$r$].Qj ← RegisterStat[$rs$].Qi}  
else {RS[$r$].Vj ← Regs[$rs$]; RS[$r$].Qj ← 0};  
RS[$r$].A ← imm; RS[$r$].Busy ← yes; |
| Load only         |             | RegisterStat[$rt$].Qi ← $r$; |
| Store only        |             | if (Register Stat[$rt$].Qi ≠ 0)  
{RS[$r$].Qk ← RegisterStat[$rs$].Qi}  
else {RS[$r$].Vk ← Regs[$rt$]; RS[$r$].Qk ← 0}; |
| Execute           | (RS[$r$].Qj=0) and (RS[$r$].Qk=0) | Compute result: operands are in Vj and Vk |
| Load/Store step 1 | RS[$r$].Qi=0 & $r$ is head of load/store queue | RS[$r$].A ← RS[$r$].Vj + RS[$r$].A; |
| Load step 2       | RS[$r$].A<>0 | Read from Mem[RS[$r$].A] |
| Write result      | Execution complete at $r$ & CDB available | ∀x(if (RegisterStat[$x$].Qi= $r$)  
{Regx[$x$] ← result;  
RegisterStat[$x$].Qi ← 0});  
∀x(if (RS[$x$].Qj=$r$)  
{RS[$x$].Vj ← result;RS[$x$].Qj ← 0});  
∀x(if (RS[$x$].Qk=$r$)  
{RS[$x$].Vk ← result;RS[$x$].Qk ← 0});  
RS[$r$].Busy ← no; |
| Store             | Execution complete at $r$ & RS[$r$].Qk=0 | Mem[RS[$r$].A] ← RS[$r$].Vk;  
RS[$r$].Busy ← no; |

**FIGURE 3.5** Steps in the algorithm and what is required for each step. For the issuing instruction, $rd$ is the destination, $rs$ and $rt$ are the source register numbers, imm is the sign-extended immediate field, and $r$ is the reservation station or buffer that the instruction is assigned to. RS is the reservation-station data structure. The value returned by a FP unit or by the load unit is called result. RegisterStat is the register status data structure (not the register file, which is Regs[]). When an instruction is issued, the destination register has its Qi field set to the number of the buffer or reservation station to which the instruction is issued. If the operands are available in the registers, they are stored in the V fields. Otherwise, the Q fields are set to indicate the reservation station that will produce the values needed as source operands. The instruction waits at the reservation station until both its operands are available, indicated by zero in the Q fields. The Q fields are set to zero either when this instruction is issued, or when an instruction on which this instruction depends completes and does its write back. When an instruction has finished execution and the CDB is available, it can do its write back. All the buffers, registers, and reservation stations whose value of Qj or Qk is the same as the completing reservation station update their values from the CDB and mark the Q fields to indicate that values have been received. Thus, the CDB can broadcast its result to many destinations in a single clock cycle, and if the waiting instructions have their operands, they can all begin execution on the next clock cycle. Loads go through two steps in Execute, and stores perform slightly differently during Write Result, where they may have to wait for the value to store. Remember that to preserve exception behavior, instructions should not be allowed to execute if a branch that is earlier in program order has not yet completed. Because any concept of program order is not maintained after the Issue stage, this restriction is usually implemented by preventing any instruction from leaving the Issue step, if there is a pending branch already in the pipeline. In Section 3.7, we will see how speculation support removes this restriction.
Tomasulo’s Algorithm: A Loop-Based Example
To understand the full power of eliminating WAW and WAR hazards through dynamic renaming of registers, we must look at a loop. Consider the earlier following simple sequence for multiplying the elements of an array by a scalar in F2:

```
Loop:  L.D  F0,0(R1)
       MUL.D F4,F0,F2
       S.D  F4,0(R1)
       DADDUI R1,R1,-8
       BNE  R1,R2,Loop; branches if R1≠0
```

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 loads-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 the multiplies could complete in four clock cycles. As we will see later in this chapter, when extended with multiple instruction issue, Tomasulo’s approach can sustain more than one instruction per clock.
A load and store can safely be done in a different order, provided they access different addresses. If a load and a store access the same address, then either:

- the load is before the store in program order and interchanging them results in a WAR hazard, or
- the store is before the load in program order and interchanging them results in a RAW hazard.
Similarly, interchanging two stores to the same address results in WAW hazard.

Hence, to determine if a load can be executed at a given time, the processor can check whether any uncompleted store that precedes the load in program order shares the same data memory address as the load. Similarly, a store must wait until there are no unexecuted loads or stores that are earlier in program order and share the same data memory address.

To detect such hazards, the processor must have computed the data memory address associated with any earlier memory operation. A simple, but not necessarily optimal, way to guarantee that the processor has all such addresses is to perform the effective address calculations in program order. (We really only need keep the relative order between stores and other memory references; that is, loads can be reordered freely.).

Let's consider the situation of a load first. If we perform effective address calculation in program order, then when a load has completed effective address calculation, we can check whether there is an address conflict by examining the A field of all active store buffers. If the load address matches the address of any active entries in the store buffer, the load instruction is not sent to the load buffer until the conflicting store completes. (Some implementations bypass the value directly to the load from a pending store, reducing the delay for this RAW hazard.)

Stores operate similarly, except that the processor must check for conflicts in both the load buffers and the store buffers, since conflicting stores cannot be reordered with respect to either a load or a store. This dynamic disambiguation of addresses is an alternative to the techniques, discussed in the next chapter, that a compiler would use when interchanging a load and store.

A dynamically scheduled pipeline can yield very high performance, provided branches are predicted accurately—an issue we address in the next section. The major drawback of this approach is the complexity of the Tomasulo scheme, which requires a large amount of hardware. In particular, each reservation station must contain an associative buffer, which must run at high speed, as well as complex control logic. Lastly, the performance can be limited by the single completion bus (CDB). Although additional CDBs can be added, each CDB must interact with each the reservation station, and the associative tag-matching hardware would need to be duplicated at each station for each CDB.

In Tomasulo’s scheme two different techniques are combined: the renaming of the architectural registers to a larger set of registers and the buffering of source operands from the register file. Source operand buffering resolves WAR hazards that arise when the operand is available in the registers. As we will see later, it is also possible to eliminate WAR hazards by the renaming of a register together with the buffering of a result until no outstanding references to the earlier version of the register remain. This approach will be used when we discuss hardware speculation.

Tomasulo’s scheme is particularly appealing if the designer is forced to pipeline an architecture for which it is difficult to schedule code, that has a shortage
of registers, or for which the designer wishes to obtain high performance without pipeline specific compilation. On the other hand, the advantages of the Tomasulo approach versus compiler scheduling for an efficient single-issue pipeline are probably fewer than the costs of implementation. But, as processors become more aggressive in their issue capability and designers are concerned with the performance of difficult-to-schedule code (such as most nonnumeric code), techniques such as register renaming and dynamic scheduling have become more important. Furthermore, the role of dynamic scheduling as a basis for hardware speculation has made this approach very popular in the past five years.

The key components for enhancing ILP in Tomasulo’s algorithm are dynamic scheduling, register renaming, and dynamic memory disambiguation. It is difficult to assess the value of these features independently. When we examine the studies of ILP in section 3.8, we will look at how these features affect the amount of parallelism discovered until ideal circumstances.

Corresponding to the dynamic hardware techniques for scheduling around data dependences are dynamic techniques for handling branches efficiently. These techniques are used for two purposes: to predict whether a branch will be taken and to find the target more quickly. \textit{Hardware branch prediction}, the name for these techniques, is the next topic we discuss.

The previous section describes techniques for overcoming data hazards. The frequency of branches and jumps demands that we also attack the potential stalls arising from control dependences. Indeed, as the amount of ILP we attempt to exploit grows, control dependences rapidly become the limiting factor. Although schemes in this section are helpful in processors that try to maintain one instruction issue per clock, for two reasons they are \textit{crucial} to any processor that tries to issue more than one instruction per clock. First, branches will arrive up to \( n \) times faster in an \( n \)-issue processor and providing an instruction stream to the processor will probably require that we predict the outcome of branches. Second, Amdahl’s Law reminds us that relative impact of the control stalls will be larger with the lower potential CPI in such machines.

In the first chapter, we examined a variety of basic schemes (e.g., predict not taken and delayed branch) for dealing with branches. Those schemes were all static: the action taken does not depend on the dynamic behavior of the branch. This section focuses on using hardware to dynamically predict the outcome of a branch—the prediction will depend on the behavior of the branch at runtime and will change if the branch changes its behavior during execution.

We start with a simple branch prediction scheme and then examine approaches that increase the accuracy of our branch prediction mechanisms. After that, we look at more elaborate schemes that try to find the instruction following a branch even earlier. The goal of all these mechanisms is to allow the processor to resolve the outcome of a branch early, thus preventing control dependences from causing stalls. The effectiveness of a branch prediction scheme depends not only on the accuracy, but also on the cost of a branch when the prediction is correct and when the prediction is incorrect. These branch penalties depend on the structure of the
pipeline, the type of predictor, and the strategies used for recovering from misprediction.

Basic Branch Prediction and Branch-Prediction Buffers

The simplest dynamic branch-prediction scheme is a branch-prediction buffer or branch history table. A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not. This scheme is the simplest sort of buffer; it has no tags and is useful only to reduce the branch delay when it is longer than the time to compute the possible target PCs. We don’t know, in fact, if the prediction is correct—it may have been put there by another branch that has the same low-order address bits. But this doesn’t matter. The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back. Of course, this buffer is effectively a cache where every access is a hit, and, as we will see, the performance of the buffer depends on both how often the prediction is for the branch of interest and how accurate the prediction is when it matches. Before we analyze the performance, it is useful to make a small, but important, improvement in the accuracy of the branch prediction scheme.

This simple one-bit prediction scheme has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken. The following example shows this.

**Example** Consider a loop branch whose behavior is taken nine times in a row, then not taken once. What is the prediction accuracy for this branch, assuming the prediction bit for this branch remains in the prediction buffer?

**Answer** The steady-state prediction behavior will mispredict on the first and last loop iterations. Mispredicting the last iteration is inevitable since the prediction bit will say taken (the branch has been taken nine times in a row at that point). The misprediction on the first iteration happens because the bit is flipped on prior execution of the last iteration of the loop, since the branch was not taken on that iteration. Thus, the prediction accuracy for this branch that is taken 90% of the time is only 80% (two incorrect predictions and eight correct ones). In general, for branches used to form loops—a branch is taken many times in a row and then not taken once—a one-bit predictor will mispredict at twice the rate that the branch is not taken. It seems that we should expect that the accuracy of the predictor would at least match the taken branch frequency for these highly regular branches.

To remedy this, two-bit prediction schemes are often used. In a two-bit scheme, a prediction must miss twice before it is changed. Figure 3.7 shows the finite-state processor for a two-bit prediction scheme.
The two-bit scheme is actually a specialization of a more general scheme that has an \( n \)-bit saturating counter for each entry in the prediction buffer. With an \( n \)-bit counter, the counter can take on values between 0 and \( 2^n - 1 \); when the counter is greater than or equal to one half of its maximum value (\( 2^{n-1} \)), the branch is predicted as taken; otherwise, it is predicted untaken. As in the two-bit scheme, the counter is incremented on a taken branch and decremented on an untaken branch. Studies of \( n \)-bit predictors have shown that the two-bit predictors do almost as well, and thus most systems rely on two-bit branch predictors rather than the more general \( n \)-bit predictors.

A branch-prediction buffer can be implemented as a small, special “cache” accessed with the instruction address during the IF pipe stage, or as a pair of bits attached to each block in the instruction cache and fetched with the instruction. If the instruction is decoded as a branch and if the branch is predicted as taken, fetching begins from the target as soon as the PC is known. Otherwise, sequential fetching and executing continue. If the prediction turns out to be wrong, the prediction bits are changed as shown in Figure 3.7.

**FIGURE 3.7** The states in a two-bit prediction scheme. By using two bits rather than one, a branch that strongly favors taken or not taken—as many branches do—will be mispredicted less often than with a one-bit predictor. The two bits are used to encode the four states in the system. In a counter implementation, the counters are incremented when a branch is taken and decremented when it is not taken; the counters saturate at 00 or 11. One complication of the two-bit scheme is that it updates the prediction bits more often than a one-bit predictor, which only updates the prediction bit on a mispredict. Since we typically read the prediction bits on every cycle, a two-bit predictor will typically need both a read and a write access port.
Although this scheme is useful for most pipelines, the five-stage, classic pipeline finds out both whether the branch is taken and what the target of the branch is at roughly the same time, assuming no hazard in accessing the register specified in the conditional branch. (Remember that this is true for the five-stage pipeline because the branch does a compare of a register against zero during the ID stage, which is when the effective address is also computed.) Thus, this scheme does not help for the five-stage pipeline; we will explore a scheme that can work for such pipelines, and for machines issuing multiple instructions per clock, a little later. First, let’s see how well branch prediction works in general.

What kind of accuracy can be expected from a branch-prediction buffer using two bits per entry on real applications? For the SPEC89 benchmarks a branch-prediction buffer with 4096 entries results in a prediction accuracy ranging from over 99% to 82%, or a misprediction rate of 1% to 18%, as shown in Figure 3.8.

To show the differences more clearly, we plot misprediction frequency rather
3.4 Reducing Branch Costs with Dynamic Hardware Prediction

than prediction frequency. A 4K-entry buffer, like that used for these results, is considered large; smaller buffers would have worse results.

Knowing just the prediction accuracy, as shown in Figure 3.8, is not enough to determine the performance impact of branches, even given the branch costs and penalties for misprediction. We also need to take into account the branch frequency, since the importance of accurate prediction is larger in programs with higher branch frequency. For example, the integer programs—li, eqntott, espresso, and gcc—have higher branch frequencies than those of the more easily predicted FP programs.

As we try to exploit more ILP, the accuracy of our branch prediction becomes critical. As we can see in Figure 3.8, the accuracy of the predictors for integer programs, which typically also have higher branch frequencies, is lower than for the loop-intensive scientific programs. We can attack this problem in two ways: by increasing the size of the buffer and by increasing the accuracy of the scheme we use for each prediction. A buffer with 4K entries is already large and, as Figure 3.9 shows, performs quite comparably to an infinite buffer. The data in Figure 3.9 make it clear that the hit rate of the buffer is not the limiting factor. As we mentioned above, simply increasing the number of bits per predictor without changing the predictor structure also has little impact. Instead, we need to look at how we might increase the accuracy of each predictor.

Correlating Branch Predictors

These two-bit predictor schemes use only the recent behavior of a single branch to predict the future behavior of that branch. It may be possible to improve the prediction accuracy if we also look at the recent behavior of other branches rather than just the branch we are trying to predict. Consider a small code fragment from the SPEC92 benchmark eqntott (the worst case for the two-bit predictor):

```plaintext
if (aa==2)
    aa=0;
if (bb==2)
    bb=0;
if (aa!=bb) {
```

Here is the MIPS code that we would typically generate for this code fragment assuming that aa and bb are assigned to registers R1 and R2:

```plaintext
DSUBUI R3,R1,#2
BNEZ R3,L1 ;branch b1 (aa!=2)
DADD R1,R0,R0 ;aa=0
L1:   DSUBUI R3,R2,#2
BNEZ R3,L2 ;branch b2 (bb!=2)
DADD R2,R0,R0 ; bb=0
```

```plaintext
if (aa==2)
    aa=0;
if (bb==2)
    bb=0;
if (aa!=bb) {
```
Let’s label these branches b1, b2, and b3. The key observation is that the behavior of branch b3 is correlated with the behavior of branches b1 and b2. Clearly, if branches b1 and b2 are both not taken (i.e., the if conditions both evaluate to true and \(\text{aa} \) and \(\text{bb} \) are both assigned 0), then b3 will be taken, since \(\text{aa} \) and \(\text{bb} \) are clearly equal. A predictor that uses only the behavior of a single branch to predict the outcome of that branch can never capture this behavior.

Branch predictors that use the behavior of other branches to make a prediction are called *correlating predictors* or *two-level predictors*. To see how such predic-
 tors work, let's choose a simple hypothetical case. Consider the following simplified code fragment (chosen for illustrative purposes):

```c
if (d==0)
    d=1;
if (d==1)
```

Here is the typical code sequence generated for this fragment, assuming that `d` is assigned to R1:

```assembly
BNEZ R1,L1;branch b1(d!=0)
DADDIU R1,R0,#1;d==0, so d=1
L1:  DADDIU R3,R1,#-1
    BNEZ R3,L2;branch b2(d!=1)
    ...
L2:
```

The branches corresponding to the two if statements are labeled b1 and b2. The possible sequences for an execution of this fragment, assuming `d` has values 0, 1, and 2, are shown in Figure 3.10. To illustrate how a correlating predictor works, assume the sequence above is executed repeatedly and ignore other branches in the program (including any branch needed to cause the above sequence to repeat).

<table>
<thead>
<tr>
<th>Initial value of d</th>
<th>d==0?</th>
<th>b1 prediction</th>
<th>Value of d before b2</th>
<th>d==1?</th>
<th>b2 prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>yes</td>
<td>not taken</td>
<td>1</td>
<td>yes</td>
<td>not taken</td>
</tr>
<tr>
<td>1</td>
<td>no</td>
<td>taken</td>
<td>1</td>
<td>yes</td>
<td>not taken</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>taken</td>
<td>2</td>
<td>no</td>
<td>taken</td>
</tr>
</tbody>
</table>

**FIGURE 3.10** Possible execution sequences for a code fragment.

From Figure 3.10, we see that if b1 is not taken, then b2 will be not taken. A correlating predictor can take advantage of this, but our standard predictor cannot. Rather than consider all possible branch paths, consider a sequence where `d` alternates between 2 and 0. A one-bit predictor initialized to not taken has the behavior shown in Figure 3.11. As the figure shows, all the branches are mispredicted!

<table>
<thead>
<tr>
<th>d=?</th>
<th>b1 prediction</th>
<th>b1 action</th>
<th>New b1 prediction</th>
<th>b2 prediction</th>
<th>b2 action</th>
<th>New b2 prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
</tr>
<tr>
<td>2</td>
<td>NT</td>
<td>T</td>
<td>T</td>
<td>NT</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>0</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
<td>T</td>
<td>NT</td>
<td>NT</td>
</tr>
</tbody>
</table>

**FIGURE 3.11** Behavior of a one-bit predictor initialized to not taken. T stands for taken, NT for not taken.
Alternatively, consider a predictor that uses one bit of correlation. The easiest way to think of this is that every branch has two separate prediction bits: one prediction assuming the last branch executed was not taken and another prediction that is used if the last branch executed was taken. Note that, in general, the last branch executed is not the same instruction as the branch being predicted, though this can occur in simple loops consisting of a single basic block (since there are no other branches in the loops).

We write the pair of prediction bits together, with the first bit being the prediction if the last branch in the program is not taken and the second bit being the prediction if the last branch in the program is taken. The four possible combinations and the meanings are listed in Figure 4.18.

<table>
<thead>
<tr>
<th>Prediction bits</th>
<th>Prediction if last branch not taken</th>
<th>Prediction if last branch taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>NT/NT</td>
<td>not taken</td>
<td>not taken</td>
</tr>
<tr>
<td>NT/T</td>
<td>not taken</td>
<td>taken</td>
</tr>
<tr>
<td>T/NT</td>
<td>taken</td>
<td>not taken</td>
</tr>
<tr>
<td>T/T</td>
<td>taken</td>
<td>taken</td>
</tr>
</tbody>
</table>

**FIGURE 3.12** Combinations and meaning of the taken/not taken prediction bits. T stands for taken, NT for not taken.

The action of the one-bit predictor with one bit of correlation, when initialized to NT/NT is shown in Figure 3.13.

<table>
<thead>
<tr>
<th>d=?</th>
<th>b1 prediction</th>
<th>b1 action</th>
<th>New b1 prediction</th>
<th>b2 prediction</th>
<th>b2 action</th>
<th>New b2 prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>NT/NT</td>
<td>T</td>
<td>T/NT</td>
<td>NT/NT</td>
<td>T</td>
<td>NT/T</td>
</tr>
<tr>
<td>0</td>
<td>T/NT</td>
<td>NT</td>
<td>T/NT</td>
<td>NT/T</td>
<td>NT</td>
<td>NT/T</td>
</tr>
<tr>
<td>2</td>
<td>T/NT</td>
<td>T</td>
<td>T/NT</td>
<td>NT/T</td>
<td>T</td>
<td>NT/T</td>
</tr>
<tr>
<td>0</td>
<td>T/NT</td>
<td>NT</td>
<td>T/NT</td>
<td>NT/T</td>
<td>NT</td>
<td>NT/T</td>
</tr>
</tbody>
</table>

**FIGURE 3.13** The action of the one-bit predictor with one bit of correlation, initialized to not taken/not taken. T stands for taken, NT for not taken. The prediction used is shown in bold.

In this case, the only misprediction is on the first iteration, when d = 2. The correct prediction of b1 is because of the choice of values for \( d \), since b1 is not obviously correlated with the previous prediction of b2. The correct prediction of b2, however, shows the advantage of correlating predictors. Even if we had chosen different values for \( d \), the predictor for b2 would correctly predict the case when b1 is not taken on every execution of b2 after one initial incorrect prediction.

The predictor in Figures 3.12 and 3.13 is called a (1,1) predictor since it uses the behavior of the last branch to choose from among a pair of one-bit branch
predictors. In the general case an \((m,n)\) predictor uses the behavior of the last \(m\) branches to choose from \(2^m\) branch predictors, each of which is a \(n\)-bit predictor for a single branch. The attraction of this type of correlating branch predictor is that it can yield higher prediction rates than the two-bit scheme and requires only a trivial amount of additional hardware. The simplicity of the hardware comes from a simple observation: The global history of the most recent \(m\) branches can be recorded in an \(m\)-bit shift register, where each bit records whether the branch was taken or not taken. The branch-prediction buffer can then be indexed using a concatenation of the low-order bits from the branch address with the \(m\)-bit global history. For example, Figure 3.14 shows a (2,2) predictor and how the prediction is accessed.

![Diagram](image)

**FIGURE 3.14** A (2,2) branch-prediction buffer uses a two-bit global history to choose from among four predictors for each branch address. Each predictor is in turn a two-bit predictor for that particular branch. The branch-prediction buffer shown here has a total of 64 entries; the branch address is used to choose four of these entries and the global history is used to choose one of the four. The two-bit global history can be implemented as a shifter register that simply shifts in the behavior of a branch as soon as it is known.
There is one subtle effect in this implementation. Because the prediction buffer is not a cache, the counters indexed by a single value of the global predictor may in fact correspond to different branches at some point in time. This insight is no different from our earlier observation that the prediction may not correspond to the current branch. In Figure 3.14 we draw the buffer as a two-dimensional object to ease understanding. In reality, the buffer can simply be implemented as a linear memory array that is two bits wide; the indexing is done by concatenating the global history bits and the number of required bits from the branch address. For the example in Figure 3.14, a (2,2) buffer with 64 total entries, the four low-order address bits of the branch (word address) and the two global bits form a six-bit index that can be used to index the 64 counters.

How much better do the correlating branch predictors work when compared with the standard two-bit scheme? To compare them fairly, we must compare predictors that use the same number of state bits. The number of bits in an \((m,n)\) predictor is

\[
2^m \times n \times \text{Number of prediction entries selected by the branch address}
\]

A two-bit predictor with no global history is simply a \((0,2)\) predictor.

**Example**

How many bits are in the \((0,2)\) branch predictor we examined earlier? How many bits are in the branch predictor shown in Figure 3.14?

**Answer**

The earlier predictor had 4K entries selected by the branch address. Thus the total number of bits is

\[
2^0 \times 2 \times 4K = 8K.
\]

The predictor in Figure 3.14 has

\[
2^2 \times 2 \times 16 = 128 \text{ bits}.
\]

To compare the performance of a correlating predictor with that of our simple two-bit predictor examined in Figure 3.8, we need to determine how many entries we should assume for the correlating predictor.

**Example**

How many branch-selected entries are in a \((2,2)\) predictor that has a total of 8K bits in the prediction buffer?

**Answer**

We know that
3.4 Reducing Branch Costs with Dynamic Hardware Prediction

\[2^2 \times 2 \times \text{Number of prediction entries selected by the branch} = 8K.\]

Hence

\[\text{Number of prediction entries selected by the branch} = 1K.\]

Figure 3.15 compares the performance of the earlier two-bit simple predictor

![Comparison of two-bit predictors](image)

**FIGURE 3.15 Comparison of two-bit predictors.** A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating two-bit predictor with unlimited entries and a two-bit predictor with two bits of global history and a total of 1024 entries.
with 4K entries and a (2,2) predictor with 1K entries. As you can see, this predictor not only outperforms a simple two-bit predictor with the same total number of state bits, it often outperforms a two-bit predictor with an unlimited number of entries.

There are a wide spectrum of correlating predictors, with the (0,2) and (2,2) predictors being among the most interesting. The Exercises ask you to explore the performance of a third extreme: a predictor that does not rely on the branch address. For example, a (12,2) predictor that has a total of 8K bits does not use the branch address in indexing the predictor, but instead relies solely on the global branch history. Surprisingly, this degenerate case can outperform a noncorrelating two-bit predictor if enough global history is used and the table is large enough!

**Tournament Predictors: Adaptively Combining Local and Global Predictors**

The primary motivation for correlating branch predictors came from the observation that the standard 2-bit predictor using only local information failed on some important branches and that by adding global information, the performance could be improved. Tournament predictors take this insight to the next level, by using multiple predictors, usually one based on global information and one based on local information, and combining them with a selector. Tournament predictors can achieve both better accuracy at medium sizes (8Kb-32Kb) and also make use of very large numbers of prediction bits effectively.

Tournament predictors are the most popular form of multilevel branch predictors. A multilevel branch predictor use several levels of branch prediction tables together with an algorithm for choosing among the multiple predictors; we will see several variations on multilevel predictors in this section. Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors. The four states of the counter dictate whether to use predictor 1 or predictor 2. The state transition diagram is shown in Figure 3.16.

The advantage of a tournament predictor is its ability to select the right predictor for the right branch. Figure 3.17 shows how the tournament predictor selects between a local and global predictor depending on the benchmark, as well as on the branch. The ability to choose between a prediction based on strictly local information and one incorporating global information on a per branch basis is particularly critical in the integer benchmarks.

Figure 3.18 looks at the performance of three different predictors (a local 2-bit predictor, a correlating predictor, and a tournament predictor) for different numbers of bits using SPEC89 as the benchmark. As we saw earlier, the prediction capability of the local predictor does not improve beyond a certain size. The correlating predictor shows a significant improvement, and the tournament predictor generates slightly better performance.

**An Example: the Alpha 21264 Branch Predictor**

The 21264 uses the most sophisticated branch predictor in any processor as of 2001. The 21264 has a tournament predictor using 4K 2-bit counters indexed by the local branch address to choose from among a global predictor and a local pre-
3.4 Reducing Branch Costs with Dynamic Hardware Prediction

The global predictor also has 4K entries and is indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor.

The local predictor consists of a two-level predictor. The top level is a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent ten branch outcomes for the entry. That is, if the branch was taken 10 or more times in a row, the entry in the local history table will be all 1s. If the branch is alternately taken and untaken the history entry consist of alternating 0s and 1s. This 10-bit history allows patterns of up to ten branches to be discovered and predicted. The selected entry from the local history table is used to index a table of 1K entries consisting a three-bit saturating counters, which provide the local prediction. This combination, which uses a total of 29 Kbits, leads to high accuracy in branch prediction. For the SPECfp95 benchmarks there is less than

FIGURE 3.16 The state transition diagram for a tournament predictor has four states corresponding to which predictor to use. The counter is incremented whenever the “predicted” predictor is correct and the other predictor is incorrect, and it is decremented in the reverse situation.
FIGURE 3.17  The fraction of predictions coming from the local predictor for a tournament predictor using the SPEC89 benchmarks. The tournament predictor selects between a local 2-bit predictor and a 2-bit local/global predictor, called gshare. Gshare is indexed by an exclusive or of the branch address bits and the global history; it performs similarly to the correlating predictor discussed earlier. In this case each predictor has 1,024 entries, each 2-bits, for a total of 6Kbits.
one misprediction per 1000 completed instructions, and for SPECint95, there are about 11.5 mispredictions per 1000 completed instructions.

3.5 High Performance Instruction Delivery

In a high performance pipeline, especially one with multiple issue, predicting branches well is not enough: we actually have to be able to deliver a high bandwidth instruction stream. In recent multiple issue processors, this has meant delivering 4-8 instructions every clock cycle. To accomplish this, we consider three concepts in this section: a branch target buffer, an integrated instruction fetch unit, and dealing with indirect branches, by predicting return addresses.

Branch Target Buffers

To reduce the branch penalty for our five-stage pipeline, we need to know from what address to fetch by the end of IF. This requirement means we must know whether the as-yet-undecoded instruction is a branch and, if so, what the next PC should be. If the instruction is a branch and we know what the next PC should be, we can have a branch penalty of zero. A branch-prediction cache that stores the predicted address for the next instruction after a branch is called a branch-target buffer or branch-target cache.
For the classic, five-stage pipeline, a branch-prediction buffer is accessed during the ID cycle, so that at the end of ID we know the branch-target address (since it is computed during ID), the fall-through address (computed during IF), and the prediction. Thus, by the end of ID we know enough to fetch the next predicted instruction. For a branch-target buffer, we access the buffer during the IF stage using the instruction address of the fetched instruction, a possible branch, to index the buffer. If we get a hit, then we know the predicted instruction address at the end of the IF cycle, which is one cycle earlier than for a branch-prediction buffer.

Because we are predicting the next instruction address and will send it out before decoding the instruction, we must know whether the fetched instruction is predicted as a taken branch. Figure 3.19 shows what the branch-target buffer looks like. If the PC of the fetched instruction matches a PC in the buffer, then the corresponding predicted PC is used as the next PC. In Chapter 5 we will discuss caches in much more detail; we will see that the hardware for this branch-target buffer is essentially identical to the hardware for a cache.

![FIGURE 3.19 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and the second field, predicted PC, contains the prediction for the next PC after the branch. Fetching begins immediately at that address. The third field, which is optional, may be used for extra prediction state bits.](image-url)
If a matching entry is found in the branch-target buffer, fetching begins immediately at the predicted PC. Note that (unlike a branch-prediction buffer) the entry must be for this instruction, because the predicted PC will be sent out before it is known whether this instruction is even a branch. If we did not check whether the entry matched this PC, then the wrong PC would be sent out for instructions that were not branches, resulting in a slower processor. We only need to store the predicted-taken branches in the branch-target buffer, since an untaken branch follows the same strategy (fetch the next sequential instruction) as a nonbranch. Complications arise when we are using a two-bit predictor, since this requires that we store information for both taken and untaken branches. One way to resolve this is to use both a target buffer and a prediction buffer, which is the solution used by several PowerPC processors. We assume that the buffer only holds PC-relative conditional branches, since this makes the target address a constant; it is not hard to extend the mechanism to work with indirect branches.

Figure 3.20 shows the steps followed when using a branch-target buffer and where these steps occur in the pipeline. From this we can see that there will be no branch delay if a branch-prediction entry is found in the buffer and is correct. Otherwise, there will be a penalty of at least two clock cycles. In practice, this penalty could be larger, since the branch-target buffer must be updated. We could assume that the instruction following a branch or at the branch target is not a branch, and do the update during that instruction time; however, this does complicate the control. Instead, we will take a two-clock-cycle penalty when the branch is not correctly predicted or when we get a miss in the buffer. Dealing with the mispredictions and misses is a significant challenge, since we typically will have to halt instruction fetch while we rewrite the buffer entry. Thus, we would like to make this process fast to minimize the penalty.

To evaluate how well a branch-target buffer works, we first must determine the penalties in all possible cases. Figure 3.21 contains this information.

**Example** Determine the total branch penalty for a branch-target buffer assuming the penalty cycles for individual mispredictions from Figure 3.21. Make the following assumptions about the prediction accuracy and hit rate:

- prediction accuracy is 90% (for instructions in the buffer)
- hit rate in the buffer is 90% (for branches predicted taken)

Assume that 60% of the branches are taken.

**Answer** We compute the penalty by looking at the probability of two events: the branch is predicted taken but ends up being not taken, and the branch is taken but is not found in the buffer. Both carry a penalty of two cycles.
FIGURE 3.20  The steps involved in handling an instruction with a branch-target buffer. If the PC of an instruction is found in the buffer, then the instruction must be a branch that is predicted taken; thus, fetching immediately begins from the predicted PC in ID. If the entry is not found and it subsequently turns out to be a taken branch, it is entered in the buffer along with the target, which is known at the end of ID. If the entry is found, but the instruction turns out not to be a taken branch, it is removed from the buffer. If the instruction is a branch, is found, and is correctly predicted, then execution proceeds with no delays. If the prediction is incorrect, we suffer a one-clock-cycle delay fetching the wrong instruction and restart the fetch one clock cycle later, leading to a total mispredict penalty of two clock cycles. If the branch is not found in the buffer and the instruction turns out to be a branch, we will have proceeded as if the instruction were not a branch and can turn this into an assume-not-taken strategy. The penalty will differ depending on whether the branch is actually taken or not.
This penalty compares with a branch penalty for delayed branches, which we evaluated in Chapter 1, of about 0.5 clock cycles per branch. Remember, though, that the improvement from dynamic branch prediction will grow as the branch delay grows; in addition, better predictors will yield a larger performance advantage.

One variation on the branch-target buffer is to store one or more target instructions instead of, or in addition to, the predicted target address. This variation has two potential advantages. First, it allows the branch-target buffer access to take longer than the time between successive instruction fetches, possibly allowing a larger branch-target buffer. Second, buffering the actual target instructions allows us to perform an optimization called branch folding. Branch folding can be used to obtain zero-cycle unconditional branches, and sometimes zero-cycle conditional branches. Consider a branch-target buffer that buffers instructions from the predicted path and is being accessed with the address of an unconditional branch. The only function of the unconditional branch is to change the PC. Thus, when the branch-target buffer signals a hit and indicates that the branch is unconditional, the pipeline can simply substitute the instruction from the branch-target buffer in place of the instruction that is returned from the cache (which is the unconditional branch). If the processor is issuing multiple instructions per cycle, then the buffer will need to supply multiple instructions to obtain the maxi-
mum benefit. In some cases, it may be possible to eliminate the cost of a conditional branch when the condition codes are preset.

**Integrated Instruction Fetch Units**

To meet the demands of multiple issue processors, many recent designers have chosen to implement an integrated instruction fetch unit, as a separate autonomous unit that feeds instructions to the rest of the pipeline. Essentially, this amounts to recognizing that characterizing instruction fetch as a simple single pipeline stage given the complexities of multiple issue is no longer valid.

Instead, recent designs have used an integrated instruction fetch unit that integrates several functions:

1. **Integrated branch prediction:** the branch predictor becomes part of the instruction fetch unit and is constantly predicting branches, so to drive the fetch pipeline.

2. **Instruction prefetch:** to deliver multiple instructions per clock, the instruction fetch unit will likely need to fetch ahead. The unit autonomously manages the prefetching of instructions (see Chapter 5 for discussion of techniques for doing this), integrating it with branch prediction.

3. **Instruction memory access and buffering:** when fetching multiple instructions per cycle, a variety of complexities are encountered, including the difficulty that fetching multiple instructions may require accessing multiple cache lines. The instruction fetch unit encapsulates this complexity, using prefetch to try to hide the cost of crossing cache blocks. The instruction fetch unit also provides buffering, essentially acting as an on-demand unit to provide instructions to the issue stage as needed and in the quantity needed.

As designers try to increase the number of instructions executed per clock, instruction fetch will become an ever more significant bottleneck and clever new ideas will be needed to deliver instructions at the necessary rate. One of the emerging ideas, called *trace caches*, is discussed in Chapter 5.

**Return Address Predictors**

Another method that designers have studied and included in many recent processors is a technique for predicting indirect jumps, that is, jumps whose destination address varies at runtime. Although high-level language programs will generate such jumps for indirect procedure calls, select or case statements, and FORTRAN-computed gotos, the vast majority of the indirect jumps come from procedure returns. For example, for the SPEC89 benchmarks, procedure returns account for 85% of the indirect jumps on average. For languages like C++ and Java, procedure returns are even more frequent. Thus, focusing on procedure returns seems appropriate.

Though procedure returns can be predicted with a branch-target buffer, the accuracy of such a prediction technique can be low if the procedure is called from
multiple sites and the calls from one site are not clustered in time. To overcome this problem, the concept of a small buffer of return addresses operating as a stack has been proposed. This structure caches the most recent return addresses: pushing a return address on the stack at a call and popping one off at a return. If the cache is sufficiently large (i.e., as large as the maximum call depth), it will predict the returns perfectly. Figure 3.22 shows the performance of such a return buffer with 1–16 elements for a number of the SPEC benchmarks. We will use this type of return predictor when we examine the studies of ILP in section 3.8.

Branch prediction schemes are limited both by prediction accuracy and by the penalty for misprediction. As we have seen, typical prediction schemes achieve prediction accuracy in the range of 80–95% depending on the type of program and the size of the buffer. In addition to trying to increase the accuracy of the predictor, we can try to reduce the penalty for misprediction. The penalty can be reduced by fetching from both the predicted and unpredicted direction. Fetching both paths requires that the memory system be dual-ported, have an interleaved cache, or fetch from one path and then the other. Although this adds cost to the system, it may be the only way to reduce branch penalties below a certain point. Caching addresses or instructions from multiple paths in the target buffer is another alternative that some processors have used.

**FIGURE 3.22 Prediction accuracy for a return address buffer operated as a stack.** The accuracy is the fraction of return addresses predicted correctly. Since call depths are typically not large, with some exceptions, a modest buffer works well. On average returns account for 81% of the indirect jumps in these six benchmarks.
We have seen a variety of software-based static schemes and hardware-based dynamic schemes for trying to boost the performance of our pipelined processor. These schemes attack both the data dependences (discussed in the previous subsections) and the control dependences (discussed in this subsection). Our focus to date has been on sustaining the throughput of the pipeline at one instruction per clock. In the next section we will look at techniques that attempt to exploit more parallelism by issuing multiple instructions in a clock cycle.

3.6 Taking Advantage of More ILP with Multiple Issue

The techniques of the previous two sections can be used to eliminate data and control stalls and achieve an ideal CPI of 1. To improve performance further we would like to decrease the CPI to less than one. But the CPI cannot be reduced below one if we issue only one instruction every clock cycle.

The goal of the multiple-issue processors, discussed in this section, is to allow multiple instructions to issue in a clock cycle. Multiple-issue processors come in two basic flavors: superscalar processors and VLIW (very long instruction word) processors. Superscalar processors issue varying numbers of instructions per clock and are either statically scheduled (using compiler techniques covered in the next chapter) or dynamically scheduled using techniques based on Tomasulo’s algorithm. Statically scheduled processor use in-order execution, while dynamically scheduled processors use out-of-order execution.

VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction (hence, they are also known as EPIC—Explicitly Parallel Instruction Computers). VLIW and EPIC processors are inherently statically scheduled by the compiler. The next chapter covers both VLIWs and the necessary compiler technology in detail, so between this chapter and the next, we will have cover most of the techniques for exploiting instruction level parallelism through multiple issue that are in use in existing processors. Figure 3.23 summarizes the basic approaches to multiple issue, their distinguishing characteristics, and shows processors that use each approach.

Although early superscalar processors used static instruction scheduling, and embedded processors still do, most leading-edge desktop and servers now use superscalars with some degree of dynamic scheduling. In this section, we introduce the superscalar concept with a simple, statically scheduled processor, which will require the techniques from the next chapter to achieve good efficiency. We then explore in detail a dynamically scheduled superscalar that builds on the Tomasulo scheme.

Statically-Scheduled Superscalar Processors

In a typical superscalar processor, the hardware might issue from zero (since it may be stalled) to eight instructions in a clock cycle. In a statically-scheduled superscalar, instructions issue in order and all pipeline hazards are checked for at issue time. The pipeline control logic must check for hazards among the
3.6 Taking Advantage of More ILP with Multiple Issue

Taking Advantage of More ILP with Multiple Issue

instructions being issued in a given clock cycle, as well as among the issuing instructions and all those still in execution. If some instruction in the instruction stream is dependent (i.e., will cause a data hazard) or doesn’t meet the issue criteria (i.e., will cause a structural hazard), only the instructions preceding that one in instruction sequence will be issued. In contrast, in VLIWs, the compiler has complete responsibility for creating a package of instructions that can be simultaneously issued, and the hardware does not dynamically make any decisions about multiple issue. (As we will see, for example, the Intel IA-64 architecture relies on the programmer to describe the presence of register dependences within an issue packet.) Thus, we say that a superscalar processor has dynamic issue capability, and a VLIW processor has static issue capability.

Before we look at an example, let’s explore the process of instruction issue in slightly more detail. Suppose we had a four-issue, static superscalar processor. During instruction fetch the pipeline would receive from one to four instructions from the instruction fetch unit, which may not always be able to deliver four instructions. We call this group of instructions received from the fetch unit that could potentially issue in one clock cycle the issue packet. Conceptually, the instruction fetch unit examines each instruction in the issue packet in program order. If an instruction would cause a structural hazard or a data hazard either due to an earlier instruction already in execution or due to an instruction earlier in the issue packet, then the instruction is not issued. This issue limitation results in zero to four instructions from the issue packet actually being issued in a given clock cycle. Although the instruction decode and issue process logically proceeds in sequential order through the instructions, in practice, the issue unit examines all the instructions in the issue packet at once, checks for hazards among the in-

### Table: Common Name, Issue Structure, Hazard Detection, Scheduling, Distinguishing Characteristic, Examples

<table>
<thead>
<tr>
<th>Common name</th>
<th>Issue structure</th>
<th>Hazard detection</th>
<th>Scheduling</th>
<th>Distinguishing characteristic</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Superscalar (static)</td>
<td>dynamic</td>
<td>hardware</td>
<td>static</td>
<td>in-order execution</td>
<td>Sun UltraSPARC II/III</td>
</tr>
<tr>
<td>Superscalar (dynamic)</td>
<td>dynamic</td>
<td>hardware</td>
<td>dynamic</td>
<td>some out-of-order execution</td>
<td>HP PA 8500, IBM RS64 III</td>
</tr>
<tr>
<td>Superscalar (speculative)</td>
<td>dynamic</td>
<td>hardware</td>
<td>dynamic with speculation</td>
<td>out-of-order execution with speculation</td>
<td>Pentium III/4, MIPS R10K, Alpha 21264</td>
</tr>
<tr>
<td>VLIW/LIW</td>
<td>static</td>
<td>software</td>
<td>static</td>
<td>no hazards between issue packets</td>
<td>Trimedia, i860</td>
</tr>
<tr>
<td>EPIC</td>
<td>mostly static</td>
<td>mostly software</td>
<td>mostly static</td>
<td>explicit dependences marked by compiler</td>
<td>Itanium</td>
</tr>
</tbody>
</table>

FIGURE 3.23 There are five primary approaches in use for multiple-issue processors, and this table shows the primary characteristics that distinguish them. This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. The next chapter focuses on compiler-based approaches, which are either VLIW or EPIC. Figure 3.61 on page 341 near the end of this chapter provides more details on a variety of recent superscalar processors.
Instructions in the packet and those in the pipeline, and decides which instructions can issue.

These issue checks are sufficiently complex that performing them in one cycle could mean that the issue logic determined the minimum clock cycle length. As a result, in many statically scheduled and all dynamically scheduled superscalars, the issue stage is split and pipelined, so that it can issue instructions every clock cycle.

This division is not, however, totally straightforward because the processor must also detect any hazards between the two packets of instructions while they are still in the issue pipeline. One approach is to use the first stage of the issue pipeline to decide how many instructions from the packet can issue simultaneously, ignoring instructions already issued, and use the second stage to examine hazards among the selected instructions and those that have already been issued. By splitting the issue pipeline and pipelining it, the performance cost of superscalar instruction issue tends to be higher branch penalties, further increasing the importance of branch prediction.

As we increase the processor’s issue rate, further pipelining of the issue stage could become necessary. Although breaking the issue stage into two stages is reasonably straightforward, it is less obvious how to pipeline it further. Thus, instruction issue is likely to be one limitation on the clock rate of superscalar processors.

A Statically Scheduled Superscalar MIPS Processor

What would the MIPS processor look like as a superscalar? For simplicity, let’s assume two instructions can be issued per clock cycle and that one of the instructions can be a load, store, branch, or integer ALU operation, and the other can be any floating-point operation. Note that we consider loads and stores, including those to floating-point registers, as integer operations. As we will see, issue of an integer operation in parallel with a floating-point operation is much simpler and less demanding than arbitrary dual issue. This configuration is, in fact, very close to the organization used in the HP 7100 processor. Although high-end desktop processors now do four or more issues per clock, dual issue superscalar pipelines are becoming common at the high-end of the embedded processor market.

Issuing two instructions per cycle will require fetching and decoding 64 bits of instructions. Early superscalars often limited the placement of the instruction types; for example, the integer instruction must be first, but modern superscalars have dropped this restriction. Assuming the instruction placement is not limited, there are three steps involved in fetch and issue: fetch two instructions from the cache, determine whether zero, one, or two instructions can issue, and issue them to the correct functional unit.

Fetching two instructions is more complex than fetching one, since the instruction pair could appear anywhere in the cache block. Many processors will only fetch one instruction if the first instruction of the pair is the last word of a cache block. High-end superscalars generally rely on an independent instruction
prefetch unit, as mentioned in the previous section and described further in Chapter 5.

For this simple superscalar doing the hazard checking is relatively straightforward, since the restriction of one integer and one FP instruction eliminates most hazard possibilities within the issue packet, making it sufficient in many cases to look only at the opcodes of the instructions. The only difficulties that arise are when the integer instruction is a floating-point load, store, or move. This possibility creates contention for the floating-point register ports and may also create a new RAW hazard when the second instruction of the pair depends on the first (e.g., the first is an FP load and the second an FP operation, or the first is an FP operation and the second an FP store). This use of an issue restriction, which represents a structural hazard, to reduce the complexity of both hazard detection and pipeline structure is common in multiple issue processors. (There is also the possibility of new WAR and WAW hazards across issue packet boundaries.)

Finally, the instructions chosen for execution are dispatched to their appropriate functional units. Figure 3.24 shows how the instructions look as they go into the pipeline in pairs; for simplicity the integer instruction is always shown first, though it may be the second instruction in the issue packet.

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Pipe stages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Integer instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Integer instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>Integer instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
<tr>
<td>FP instruction</td>
<td>IF ID EX MEM WB</td>
</tr>
</tbody>
</table>

FIGURE 3.24 Superscalar pipeline in operation. The integer and floating-point instructions are issued at the same time, and each executes at its own pace through the pipeline. This figure assumes that all the FP instructions are adds that take three execution cycles. This scheme will only improve the performance of programs with a large fraction of floating-point operations.

With this pipeline, we have substantially boosted the rate at which we can issue floating-point instructions. To make this worthwhile, however, we need either pipelined floating-point units or multiple independent units. Otherwise, the floating-point datapath will quickly become the bottleneck, and the advantages gained by dual issue will be small.

By issuing an integer and a floating-point operation in parallel, the need for additional hardware, beyond the enhanced hazard detection logic, is minimized—integer and floating-point operations use different register sets and dif-
ferent functional units on load-store architectures. Allowing FP loads and stores
to issue with FP operations, a highly desirable capability for performance rea-
sons, creates the need for an additional read/write port on the FP register file. In
addition, because there are twice as many instructions in the pipeline, a larger set
of bypass paths will be needed.

A final complication is maintaining a precise exception model. To see how im-
precise exceptions can happen, consider the following:

- A floating point instruction can finish execution after an integer instruction that
  is later in program order (e.g., when an FP instruction is the first instruction in
  an issue packet and both instructions are issued).

- The floating point instruction exception could be detected after the integer in-
  struction completed.

Left untouched, this situation would result in an imprecise exception because the
integer instruction, which in program order follows the FP instruction that raised
the exception, will have been completed. This situation represents a slight com-
plification over those that can arise in a single issue pipeline when the floating
point pipeline is deeper than the integer pipeline, but is no different than what we
saw could arise with a dynamically scheduled pipeline. Several solutions are pos-
sible: early detection of FP exceptions (see the pipelining appendix), the use of
software mechanisms to restore a precise exception state before resuming execu-
tion, and delaying instruction completion until we know an exception is impos-
nible (the speculation approach we cover in the next section uses this approach).

Maintaining the peak throughput for this dual issue pipeline is much harder
than it is for a single-issue pipeline. In our classic, five-stage pipeline, loads had a
latency of one clock cycle, which prevented one instruction from using the result
without stalling. In the superscalar pipeline, the result of a load instruction cannot
be used on the same clock cycle or on the next clock cycle, and hence, the next
three instructions cannot use the load result without stalling. The branch delay for
a taken branch becomes either two or three instructions, depending on whether
the branch is the first or second instruction of a pair.

To effectively exploit the parallelism available in a superscalar processor,
more ambitious compiler or hardware scheduling techniques will be needed. In
fact, without such techniques, a superscalar processor is likely to provide little or
no additional performance.

In the next chapter, we will show how relatively simple compiler techniques
suffice for a two-issue pipeline such as this one. Alternatively, we can employ an
extension of Tomasulo’s algorithm to schedule the pipeline, as the next section
shows.
Multiple Instruction Issue with Dynamic Scheduling

Dynamic scheduling is one method for improving performance in a multiple instruction issue processor. When applied to a superscalar processor, dynamic scheduling has the traditional benefit of boosting performance in the face of data hazards, but it also allows the processor to potentially overcome the issue restrictions. Put another way, although the hardware may not be able to initiate execution of more than one integer and one FP operation in a clock cycle, dynamic scheduling can eliminate this restriction at instruction issue, at least until the hardware runs out of reservation stations.

Let’s assume we want to extend Tomasulo’s algorithm to support our two-issue superscalar pipeline. We do not want to issue instructions to the reservation stations out of order, since this could lead to a violation of the program semantics. To gain the full advantage of dynamic scheduling we should remove the constraint of issuing one integer and one FP instruction in a clock, but this will significantly complicate instruction issue.

Alternatively, we could use a simpler scheme: separate the data structures for the integer and floating-point registers, then we can simultaneously issue a floating-point instruction and an integer instruction to their respective reservation stations, as long as the two issued instructions do not access the same register set. Unfortunately, this approach bars issuing two instructions with a dependence in the same clock cycle, such as a floating-point load (an integer instruction) and a floating-point add. Rather than try to fix this problem, let’s explore the general scheme for allowing the issue stage to handle two arbitrary instructions per clock.

Two different approaches have been used to issue multiple instructions per clock in a dynamically scheduled processor, and both rely on the observation that they key is assigning a reservation station and updating the pipeline control tables. One approach is to run this step in half a clock cycle, so that two instructions can be processed in one clock cycle. A second alternative is to build the logic necessary to handle two instructions at once, including any possible dependences between the instructions. Modern superscalar processors that issue four or more instructions per clock often include both approaches: they both pipeline and widen the issue logic.

There is one final issue to discuss before we look at an example: how should dynamic branch prediction be integrated into a dynamically scheduled pipeline. The IBM 360/91 used a simple static prediction scheme, but only allowed instructions to be fetched and issued (but not actually executed) until the branch had completed. In this section, we follow the same approach. In the next section, we will examine speculation, which takes this a step further and actually executes instructions based on branch predictions.

Assume that we have the most general implementation of a two issue dynamically scheduled processor, meaning that it can issue any pair of instructions if there are reservation stations of the right type available. Because the interaction of the integer and floating point instructions is crucial, we also extend Tomasulo’s
scheme to deal with both the integer and floating point functional units and registers. Let’s see how a simple loop executes on this processor.

**EXAMPLE**

Consider the execution of the following simple loop, which adds a scalar in F2 to each element of a vector in memory. Use a MIPS pipeline extended with Tomasulo’s algorithm and with multiple issue:

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

Assume that both a floating-point and an integer operation can be issued on every clock cycle, even if they are dependent. Assume one integer functional unit used for both ALU operations and effective address calculations and a separate pipelined FP functional unit for each operation type. Assume that issue and write results take one cycle each and that there is dynamic branch-prediction hardware and a separate functional unit to evaluate branch conditions. As in most dynamically scheduled processors, the presence of the write results stage means that the effective instruction latencies will be one cycle longer than in a simple in-order pipeline. Thus, the number of cycles of latency between a source instruction and an instruction consuming the result is one cycle for integer ALU operations, two cycles for loads, and three cycles for FP add. Create a table showing when each instruction issues, begins execution, and writes its result to the CDB for the first three iterations of the loop. Assume two CDBs and assume that branches single issue (no delayed branches) but that branch prediction is perfect. Also show the resource usage for the integer unit, the floating point unit, the data cache, and the two CDBs.

**ANSWER**

The loop will be dynamically unwound and, whenever possible, instructions will be issued in pairs. The execution timing is shown in Figure 3.25 and Figure 3.26 shows the resource utilization. The loop will continue to fetch and issue a new loop iteration every three clock cycles and sustaining one iteration every three cycles would lead to an IPC of $5/3 = 1.67$. The instruction execution rate, however, is lower: by looking at the execute stage we can see that the sustained instruction completion rate is $15/16 = 0.94$. Assuming the branches are perfectly predicted, the issue unit will eventually fill all the reservation stations and will stall.
The throughput improvement versus a single issue pipeline is small because there is only one floating-point operation per iteration and, thus, the integer pipeline becomes a bottleneck. The performance could be enhanced by compiler techniques we will discuss in the next chapter. Alternatively, if the processor could execute more integer operations per cycle, larger improvements would be possible. A revised example demonstrates this potential improvement and the flexibility of dynamic scheduling to adapt to different hardware capabilities.

**Example** Consider the execution of the same loop on two-issue processor, but, in addition, assume that there are separate integer functional units for effective address calculation and for ALU operations. Create a table as in Figure 3.25 for the first three iterations of the same loop and another table.
to show the resource usage.

**Answer** Figure 3.27 shows the improvement in performance: the loop executes in five clock cycles less (11 versus 16 execution cycles). The cost of this improvement is both a separate address adder and the logic to issue to it; note that, in contrast to the earlier example, a second CDB is needed. As Figure 3.28 shows this example has a higher instruction execution rate but lower efficiency as measured by the utilization of the functional units.

Three factors limit the performance (as shown in Figure 3.27) of the two-issue dynamically scheduled pipeline:

<table>
<thead>
<tr>
<th>Clock #</th>
<th>Integer ALU</th>
<th>FP ALU</th>
<th>Data Cache</th>
<th>CDB</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1 / L.D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1 / S.D</td>
<td></td>
<td>1 / L.D</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1 / DADDIU</td>
<td></td>
<td></td>
<td>1 / L.D</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>1 / ADD.D</td>
<td></td>
<td>1 / DADDIU</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>2 / L.D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2 / S.D</td>
<td></td>
<td>2 / L.D</td>
<td>1 / ADD.D</td>
</tr>
<tr>
<td>9</td>
<td>2 / DADDIU</td>
<td></td>
<td>1 / S.D</td>
<td>2 / L.D</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>2 / ADD.D</td>
<td></td>
<td>2 / DADDIU</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>3 / L.D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>3 / S.D</td>
<td></td>
<td>3 / L.D</td>
<td>2 / ADD.D</td>
</tr>
<tr>
<td>14</td>
<td>3 / DADDIU</td>
<td></td>
<td>2 / S.D</td>
<td>3 / L.D</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>3 / ADD.D</td>
<td></td>
<td>3 / DADDIU</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td>3 / ADD.D</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td>3 / S.D</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 3.26** Resource usage table for the example shown in Figure 3.25. The entry in each box shows the opcode and iteration number of the instruction that uses the functional unit heading the column at the clock cycle corresponding to the row. Only a single CDB is actually required and that is what we show.
### FIGURE 3.27  The clock cycle of issue, execution, and writing result for a dual-issue version of our Tomasulo pipeline with separate functional units for integer ALU operations and effective address calculation, which also uses a wider CDB. The extra integer ALU allows the DADDIU to execute earlier, in turn allowing the BNE to execute earlier, and, thereby, starting the next iteration earlier.

<table>
<thead>
<tr>
<th>Iter. #</th>
<th>Instructions</th>
<th>Issues at</th>
<th>Executes</th>
<th>Memory access at</th>
<th>Write CDB at</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>L.D F0,0(R1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>First issue</td>
</tr>
<tr>
<td>1</td>
<td>ADD.D F4,F0,F2</td>
<td>1</td>
<td>5</td>
<td>8</td>
<td></td>
<td>Wait for L.D</td>
</tr>
<tr>
<td>1</td>
<td>S.D F4,0(R1)</td>
<td>2</td>
<td>3</td>
<td>9</td>
<td>4</td>
<td>Wait for ADD.D</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R1,R1,#-8</td>
<td>2</td>
<td>3</td>
<td></td>
<td>4</td>
<td>Executes earlier</td>
</tr>
<tr>
<td>1</td>
<td>BNE R1,R2,Loop</td>
<td>3</td>
<td>5</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>L.D F0,0(R1)</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>Wait for BNE complete</td>
</tr>
<tr>
<td>2</td>
<td>ADD.D F4,F0,F2</td>
<td>4</td>
<td>9</td>
<td>12</td>
<td></td>
<td>Wait for L.D</td>
</tr>
<tr>
<td>2</td>
<td>S.D F4,0(R1)</td>
<td>5</td>
<td>7</td>
<td>13</td>
<td></td>
<td>Wait for ADD.D</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R1,R1,#-8</td>
<td>5</td>
<td>6</td>
<td></td>
<td>7</td>
<td>Executes earlier</td>
</tr>
<tr>
<td>2</td>
<td>BNE R1,Loop</td>
<td>6</td>
<td>8</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>L.D F0,0(R1)</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>Wait for BNE complete</td>
</tr>
<tr>
<td>3</td>
<td>ADD.D F4,F0,F2</td>
<td>7</td>
<td>12</td>
<td>15</td>
<td></td>
<td>Wait for L.D</td>
</tr>
<tr>
<td>3</td>
<td>S.D F4,0(R1)</td>
<td>8</td>
<td>10</td>
<td>16</td>
<td></td>
<td>Wait for ADD.D</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R1,R1,#-8</td>
<td>8</td>
<td>9</td>
<td></td>
<td>10</td>
<td>Executes earlier</td>
</tr>
<tr>
<td>3</td>
<td>BNE R1,Loop</td>
<td>9</td>
<td>11</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
</tbody>
</table>

### FIGURE 3.28  Resource usage table for the example shown in Figure 3.27, using the same format as Figure 3.26.
1. There is an imbalance between the functional unit structure of the pipeline and the example loop. This imbalance means that it is impossible to fully use the FP units. To remedy this, we would need fewer dependent integer operations per loop. The next point is a different way of looking at this limitation.

2. The amount of overhead per loop iteration is very high: two of out of five instructions (the `DADDIU` and the `BNE`) are overhead. In the next chapter we look at how this overhead can be reduced.

3. The control hazard, which prevents us from starting the next `L.D` before we know whether the branch was correctly predicted, causes a one-cycle penalty on every loop iteration. The next section introduces a technique that addresses this limitation.

3.7 Hardware-Based Speculation

As we try to exploit more instruction level parallelism, maintaining control dependences becomes an increasing burden. Branch prediction reduces the direct stalls attributable to branches, but for a processor executing multiple instructions per clock, just predicting branches accurately may not be sufficient to generate the desired amount of instruction level parallelism. A wide issue processor may need to execute a branch every clock cycle to maintain maximum performance. Hence, exploiting more parallelism requires that we overcome the limitation of control dependence. The performance of the pipeline in Figure 3.25 makes this clear: there is one stall cycle each loop iteration due to a branch hazard. In programs with more branches and more data dependent branches, this penalty could be larger.

Overcoming control dependence is done by speculating on the outcome of branches and executing the program as if our guesses were correct. This mechanism represents a subtle, but important, extension over branch prediction with dynamic scheduling. In particular, with speculation, we fetch, issue, and execute instructions, as if our branch predictions were always correct; dynamic scheduling only fetches and issues such instructions. Of course, we need mechanisms to handle the situation where the speculation is incorrect. The next chapter discusses a variety of mechanisms for supporting speculation by the compiler. In this section, we explore hardware speculation, which extends the ideas of dynamic scheduling.

Hardware-based speculation combines three key ideas: dynamic branch prediction to choose which instructions to execute, speculation to allow the execution of instructions before the control dependences are resolved (with the ability to undo the effects of an incorrectly speculated sequence), and dynamic scheduling to deal with the scheduling of different combinations of basic blocks. (In comparison, dynamic scheduling without speculation only partially overlaps basic blocks, because it requires that a branch be resolved before actually executing any instructions in the successor basic block.) Hardware-based speculation fol-
allows the predicted flow of data values to choose when to execute instructions. This method of executing programs is essentially a data-flow execution: operations execute as soon as their operands are available.

The approach we examine here, and the one implemented in a number of processors (PowerPC 603/604/G3/G4, MIPS R10000/R12000, Intel Pentium II/III/4, Alpha 21264, and AMD K5/K6/Athlon), is to implement speculative execution based on Tomasulo’s algorithm. Just as with Tomasulo’s algorithm, we explain hardware speculation in the context of the floating-point unit, but the ideas are easily applicable to the integer unit.

The hardware that implements Tomasulo’s algorithm can be extended to support speculation. To do so, we must separate the bypassing of results among instructions, which is needed to execute an instruction speculatively, from the actual completion of an instruction. By making this separation, we can allow an instruction to execute and to bypass its results to other instructions, without allowing the instruction to perform any updates that cannot be undone, until we know that the instruction is no longer speculative. Using the bypassed value is like performing a speculative register read, since we do not know whether the instruction providing the source register value is providing the correct result until the instruction is no longer speculative. When an instruction is no longer speculative, we allow it to update the register file or memory; we call this additional step in the instruction execution sequence instruction commit.

The key idea behind implementing speculation is to allow instructions to execute out of order but to force them to commit in order and to prevent any irrecoverable action (such as updating state or taking an exception) until an instruction commits. In the simple single-issue five-stage pipeline we could ensure that instructions committed in order, and only after any exceptions for that instruction had been detected, simply by moving writes to the end of the pipeline. When we add speculation, we need to separate the process of completing execution from instruction commit, since instructions may finish execution considerably before they are ready to commit. Adding this commit phase to the instruction execution sequence requires some changes to the sequence as well as an additional set of hardware buffers that hold the results of instructions that have finished execution but have not committed. This hardware buffer, which we call the reorder buffer, is also used to pass results among instructions that may be speculated.

The reorder buffer (ROB, for short) provides additional registers in the same way as the reservation stations in Tomasulo’s algorithm extend the register set. The ROB holds the result of an instruction between the time the operation associated with the instruction completes and the time the instruction commits. Hence, the ROB is a source of operands for instructions, just as the reservation stations provide operands in Tomasulo’s algorithm. The key difference is that in Tomasulo’s algorithm, once an instruction writes its result, any subsequently issued instructions will find the result in the register file. With speculation, the register file is not updated until the instruction commits (and we know definitively that the instruction should execute); thus, the ROB supplies operands in the
interval between completion of instruction execution and instruction commit. The ROB is similar to the store buffer in Tomasulo’s algorithm, and we integrate the function of the store buffer into the ROB for simplicity.

Each entry in the ROB contains three fields: the instruction type, the destination field, and the value field. The instruction-type field indicates whether the instruction is a branch (and has no destination result), a store (which has a memory address destination), or a register operation (ALU operation or load, which have register destinations). The destination field supplies the register number (for loads and ALU operations) or the memory address (for stores), where the instruction result should be written. The value field is used to hold the value of the instruction result until the instruction commits. We will see an example of ROB entries shortly.

Figure 3.29 shows the hardware structure of the processor including the ROB. The ROB completely replaces the store buffers. Stores still execute in two steps, but the second step is performed by instruction commit. Although the renaming function of the reservation stations is replaced by the ROB, we still need a place to buffer operations (and operands) between the time they issue and the time they begin execution. This function is still provided by the reservation stations. Since every instruction has a position in the ROB until it commits, we tag a result using the ROB entry number rather than using the reservation station number. This tagging requires that the ROB assigned for an instruction must be tracked in the reservation station. Later in this section, we will explore an alternative implementation that uses extra registers for renaming and the ROB only to track when instructions can commit.

Here are the four steps involved in instruction execution:

1. Issue—Get an instruction from the instruction queue. Issue the instruction if there is an empty reservation station and an empty slot in the ROB, send the operands to the reservation station if they available in either the registers or the ROB. Update the control entries to indicate the buffers are in use. The number of the ROB allocated for the result is also sent to the reservation station, so that the number can be used to tag the result when it is placed on the CDB. If either all reservations are full or the ROB is full, then instruction issue is stalled until both have available entries. This stage is sometimes called dispatch in a dynamically scheduled processor.

2. Execute—If one or more of the operands is not yet available, monitor the CDB (common data bus) while waiting for the register to be computed. This step checks for RAW hazards. When both operands are available at a reservation station, execute the operation. (Some dynamically scheduled processors call this step issue, but we use the name execute, which was used in the first dynamically scheduled processor, the CDC 6600.) Instructions may take multiple clock cycles in this stage, and loads still require two steps in this stage. Stores need only have the base register available at this step, since execution
3.7 Hardware-Based Speculation

for a store at this point is only effective address calculation.

3. **Write result**—When the result is available, write it on the CDB (with the ROB tag sent when the instruction issued) and from the CDB into the ROB, as well as to any reservation stations waiting for this result. Mark the reservation station as available. Special actions are required for store instructions. If the value to be stored is available, it is written into the Value field of the ROB entry for the store. If the value to be stored is not available yet, the CDB must be monitored until that value is broadcast, at which time the Value field of the ROB entry of the store is updated. For simplicity in our description, we assume that this occurs during the Write Results stage of a store; we discuss relaxing this requirement later.

**FIGURE 3.29** The basic structure of a MIPS FP unit using Tomasulo’s algorithm and extended to handle speculation. Comparing this to Figure 3.2 on page 237, which implemented Tomasulo’s algorithm, the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated into the ROB. This mechanism can be extended to multiple issue by making the CDB (common data bus) wider to allow for multiple completions per clock.
4. **Commit**—There are three different sequences of actions at commit depending on whether the committing instruction is: a branch with an incorrect prediction, a store, or any other instruction (normal commit). The normal commit case occurs when an instruction reaches the head of the ROB and its result is present in the buffer; at this point, the processor updates the register with the result and removes the instruction from the ROB. Committing a store is similar except that memory is updated rather than a result register. When a branch with incorrect prediction reaches the head of the ROB, it indicates that the speculation was wrong. The ROB is flushed and execution is restarted at the correct successor of the branch. If the branch was correctly predicted, the branch is finished. Some machines call this commit phase *completion* or *graduation*.

Once an instruction commits, its entry in the ROB is reclaimed and the register or memory destination is updated, eliminating the need for the ROB entry. If the ROB fills, we simply stop issuing instructions until an entry is made free. Now, let’s examine how this scheme would work with the same example we used for Tomasulo’s algorithm.

**Example**

Assume the same latencies for the floating-point functional units as in earlier examples: add is 2 clock cycles, multiply is 10 clock cycles, and divide is 40 clock cycles. Using the code segment below, the same one we used to generate Figure 3.4 on page 242, show what the status tables look like when the `MUL.D` is ready to go to commit.

```plaintext
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
```

**Answer**

The result is shown in the three tables in Figure 3.30. Notice that although the `SUB.D` instruction has completed execution, it does not commit until the `MUL.D` commits. The reservation stations and register status field contain the same basic information that they did for Tomasulo’s algorithm (see page 238 for a description of those fields). The differences are that reservation station numbers are replaced with ROB entry numbers in the Qj and Qk fields, as well as in the register status fields, and we have added the Dest field to the reservation stations. The Dest field designates the ROB number that is the destination for the result produced by this reservation station entry.
The above Example illustrates the key important difference between a processor with speculation and a processor with dynamic scheduling. Compare the content of Figure 3.30 with that of Figure 3.4 (page 242), which shows the same code sequence in operation on a processor with Tomasulo’s algorithm. The key difference is that in the example above, no instruction after the earliest uncompleted instruction (\texttt{MUL.D} above) is allowed to complete. In contrast, in Figure 3.4 the \texttt{SUB.D} and \texttt{ADD.D} instructions have also completed.
One implication of this difference is that the processor with the ROB can dynamically execute code while maintaining a precise interrupt model. For example, if the \texttt{MUL.D} instruction caused an interrupt, we could simply wait until it reached the head of the ROB and take the interrupt, flushing any other pending instructions. Because instruction commit happens in order, this yields a precise
exception. By contrast, in the example using Tomasulo’s algorithm, the SUB.D and ADD.D instructions could both complete before the MUL.D raised the exception. The result is that the registers F8 and F6 (destinations of the SUB.D and ADD.D instructions) could be overwritten, and the interrupt would be imprecise. Some users and architects have decided that imprecise floating-point exceptions are acceptable in high-performance processors, since the program will likely terminate; see Appendix A for further discussion of this topic. Other types of exceptions, such as page faults, are much more difficult to accommodate if they are imprecise, since the program must transparently resume execution after handling such an exception. The use of a ROB with in-order instruction commit provides precise exceptions, in addition to supporting speculative execution, as the next Example shows.

**Example**  Consider the code example used earlier for Tomasulo’s algorithm and shown in Figure 3.6 on page 245 in execution:

```
Loop:     L.D  F0,0(R1)
         MUL.D F4,F0,F2
         S.D  F4,0(R1)
         DADDIU R1,R1,#-8
         BNE  R1,R2,Loop  ; branches if R1≠0
```

Assume that we have issued all the instructions in the loop twice. Let's also assume that the L.D and MUL.D from the first iteration have committed and all other instructions have completed execution. Normally, the store would wait in the ROB for both the effective address operand (R1 in this example) and the value (F4 in this example). Since we are only considering the floating-point pipeline, assume the effective address for the store is computed by the time the instruction is issued.

**Answer**  The result is shown in the two tables in Figure 3.31.

Because neither the register values nor any memory values are actually written until an instruction commits, the processor can easily undo its speculative actions when a branch is found to be mispredicted. Suppose that in the above example (see Figure 3.31), the branch BNE is not taken the first time. The instructions prior to the branch will simply commit when each reaches the head of the ROB; when the branch reaches the head of that buffer, the buffer is simply cleared and the processor begins fetching instructions from the other path.
In practice, machines that speculate try to recover as early as possible after a branch is mispredicted. This recovery can be done by clearing the ROB for all entries that appear after the mispredicted branch, allowing those that are before the branch in the ROB to continue, and restarting the fetch at the correct branch successor. In speculative processors, however, performance is more sensitive to the branch prediction mechanisms, since the impact of a misprediction will be higher. Thus, all the aspects of handling branches—prediction accuracy, misprediction detection, and misprediction recovery—increase in importance.

Exceptions are handled by not recognizing the exception until it is ready to commit. If a speculated instruction raises an exception, the exception is recorded in the ROB. If a branch misprediction arises and the instruction should not have been executed, the exception is flushed along with the instruction when the ROB is cleared. If the instruction reaches the head of the ROB, then we know it is no longer speculative and the exception should really be taken. We can also try to handle exceptions as soon as they arise and all earlier branches are resolved, but this is more challenging in the case of exceptions than for branch mispredict and, because it occurs less frequently, not as critical.

**FIGURE 3.31** Only the L.D and MUL.D instructions have committed, though all the others have completed execution. Hence, no reservation stations are busy and none are shown. The remaining instructions will be committed as fast as possible. The first two reorder buffers are empty, but are shown for completeness.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Busy</th>
<th>Instruction</th>
<th>State</th>
<th>Destination</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>no</td>
<td>L.D F0,0(R1)</td>
<td>Commit</td>
<td>F0</td>
<td>Mem[0+Regs[R1]]</td>
</tr>
<tr>
<td>2</td>
<td>no</td>
<td>MUL.D F4,F0,F2</td>
<td>Commit</td>
<td>F4</td>
<td>#1 x Regs[F2]</td>
</tr>
<tr>
<td>3</td>
<td>yes</td>
<td>S.D F4,0(R1)</td>
<td>Write result</td>
<td>0+Regs[R1]</td>
<td>#2</td>
</tr>
<tr>
<td>4</td>
<td>yes</td>
<td>DADDIU R1,R1,#-8</td>
<td>Write result</td>
<td>R1</td>
<td>Regs[R1]-8</td>
</tr>
<tr>
<td>5</td>
<td>yes</td>
<td>BNE R1,R2,Loop</td>
<td>Write result</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>yes</td>
<td>L.D F0,0(R1)</td>
<td>Write result</td>
<td>F0</td>
<td>Mem[#4]</td>
</tr>
<tr>
<td>7</td>
<td>yes</td>
<td>MUL.D F4,F0,F2</td>
<td>Write result</td>
<td>F4</td>
<td>#6 x Regs[F2]</td>
</tr>
<tr>
<td>8</td>
<td>yes</td>
<td>S.D F4,0(R1)</td>
<td>Write result</td>
<td>0+#4</td>
<td>#7</td>
</tr>
<tr>
<td>9</td>
<td>yes</td>
<td>DADDIU R1,R1,#-8</td>
<td>Write result</td>
<td>R1</td>
<td>#4 – 8</td>
</tr>
<tr>
<td>10</td>
<td>yes</td>
<td>BNE R1,R2,Loop</td>
<td>Write result</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**FP register status**

<table>
<thead>
<tr>
<th>Field</th>
<th>F0</th>
<th>F1</th>
<th>F2</th>
<th>F3</th>
<th>F4</th>
<th>F5</th>
<th>F6</th>
<th>F7</th>
<th>F8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reorder #</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Busy</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>...</td>
<td>no</td>
</tr>
</tbody>
</table>
Figure 3.32 shows the steps of execution for an instruction, as well as the conditions that must be satisfied to proceed to the step and the actions taken. We show the case where mispredicted branches are not resolved until commit. Although speculation seems like a simple addition to dynamic scheduling, a comparison of Figure 3.32 with the comparable figure for Tomasulo’s algorithm (see Figure 3.5 on page 243) shows that speculation adds significant complications to the control. In addition, remember that branch mispredictions are somewhat more complex as well.

There is an important difference in how stores are handled in a speculative processor, versus in Tomasulo’s algorithm. In Tomasulo’s algorithm, a store can update memory when it reaches Write Results (which ensures that the effective address has been calculated) and the data value to store is available. In a speculative processor, a store updates memory only when it reaches the head of the ROB. This difference ensures that memory is not updated until an instruction is no longer speculative.

Figure 3.32 has one significant simplification for stores, which is unneeded in practice. Figure 3.32 requires stores to wait in the write result stage for the register source operand whose value is to be stored; the value is then moved from the Vk field of the store’s reservation station to the Value field of the store’s ROB entry. In reality, however, the value to be stored need not arrive until just before the store commits and can be placed directly into the store’s ROB entry by the sourcing instruction. This is accomplished by having the hardware track when the source value to be stored is available in the store’s ROB entry and searching the ROB on every instruction completion to look for dependent stores. This addition is not complicated but adding it has two effects: we would need to add a field to the ROB and Figure 3.32, which is already in a small font, would no longer fit on one page! Although Figure 3.32 makes this simplification, in our examples, we will allow the store to pass through the write-results stage and simply wait for the value to be ready when it commits.

Like Tomasulo’s algorithm, we must avoid hazards through memory. WAW and WAR hazards through memory are eliminated with speculation, because the actual updating of memory occurs in order, when a store is at the head of the ROB, and hence, no earlier loads or stores can still be pending. RAW hazards through memory are maintained by two restrictions:

1. not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has an Destination field that matches the value of the A field of the load, and

2. maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.

Together, these two restrictions ensure that any load that accesses a memory location written to by an earlier store, cannot perform the memory access until the
**Status** | **Wait until** | **Action or bookkeeping**
--- | --- | ---
**Issue** | All instructions | if (RegisterStat[rs].Busy) /*in-flight instr. writes rs*/
| Reservation station (r) and ROB (b) both available | {h ← RegisterStat[rs].Reorder; 
| | if (ROB[h].Ready) /* Instr completed already */ 
| | {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;} 
| | else {RS[r].Qj ← h;} /* wait for instruction */ 
| | } else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;} 
| | RS[r].Busy ← yes; RS[r].Dest ← b; 
| | ROB[h].Instruction ← opcode; ROB[b].Ready ← no; 
**FP Operations and Stores** | if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/
| | {h ← RegisterStat[rt].Reorder; 
| | if (ROB[b].Ready) /* Instr completed already */ 
| | {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;} 
| | else {RS[r].Qk ← h;} /* wait for instruction */ 
| | } else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;} 
**FP Operations** | RegisterStat[rd].Qi = b; RegisterStat[rd].Busy ← yes; 
| | ROB[b].Dest ← rd; 
**Loads** | RS[r].A ← imm; RegisterStat[rt].Qi = b; 
| | RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt; 
**Stores** | RS[r].A ← imm; 
**Execute** | (RS[r].Qj = 0) and (RS[r].Qk = 0) | Compute results—operands are in Vj andVk 
| **FP Op** | Load step 1 (RS[r].Qj = 0) & there are no stores earlier in the queue | RS[r].A ← RS[r].Vj + RS[r].A; 
| | Load step 2 | Load step 1 done & all stores earlier in ROB have different address | Read from Mem[RS[r].A] 
| | Store | (RS[r].Qj = 0) & store at queue head | ROB[h].Address ← RS[r].Vj + RS[r].A; 
| | Write result | Execution done at r & CDB available. | b ← RS[r].Reorder; RS[r].Busy ← no; 
| | | ∀x (if (RS[x].Qj = b) {RS[x].Vj ← result; RS[x].Qj ← 0}); 
| | | ∀x (if (RS[x].Qk = b) {RS[x].Vk ← result; RS[x].Qk ← 0}); 
| | | ROB[b].Value ← result; ROB[b].Ready ← yes; 
**Store** | Execution done at r & (RS[r].Qk = 0) | ROB[h].Value ← RS[r].Vk; 

**FIGURE 3.32** Steps in the algorithm and what is required for each step. For the issuing instruction, rd is the destination, rs and rt are the sources, and r is the reservation station allocated and b is the assigned ROB entry. RS is the reservation-station data structure. The value returned by a reservation station is called the result. RegisterStat is the register data structure, Regs represents the actual registers, and ROB is the reorder buffer data structure.
store has written the data. Some speculative machines will actually bypass the value from the store to the load directly, when such a RAW hazard occurs.

Although this explanation of speculative execution has focused on floating point, the techniques easily extend to the integer registers and functional units, as we will see in the Putting It All Together section. Indeed, speculation may be more useful in integer programs, since such programs tend to have code where the branch behavior is less predictable. Additionally, these techniques can be extended to work in a multiple-issue processor by allowing multiple instructions to issue and commit every clock. In fact, speculation is probably most interesting in such processors, since less ambitious techniques can probably exploit sufficient ILP within basic blocks when assisted by a compiler.

Multiple Issue with Speculation

A speculative processor can be extended to multiple issue using the same techniques we employed when extending a Tomasulo-based processor in section 3.6. The same techniques for implementing the instruction issue unit can be used: We process multiple instructions per clock assigning reservation stations and reorder buffers to the instructions.

The two challenges of multiple issue with Tomasulo’s algorithm--instruction issue and monitoring the CDBs for instruction completion--become the major challenges for multiple issue with speculate. In addition, to maintain throughput of greater than one instruction per cycle, a speculative processor must be able to handle multiple instruction commits per clock cycle. To show how speculation can improve performance in a multiple issue processor consider the following example using speculation.

**EXAMPLE** Consider the execution of the following loop, which searches an array, on a two issue processor one with dynamic scheduling and one with specu-
Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table as in Figure 3.27 for the first three iterations of this loop for both machines. Assume that up to two instructions of any type can commit per clock.

**Answer**

Figure 3.33 and 3.34 show the performance for a two issue dynamically scheduled processor, without and with speculation. In this case, where a branch is a key potential performance limitation, speculation helps significantly. The third branch in the speculative processor executes in clock cycle 11, while it executes in clock cycle 19 on nonspeculative pipeline. Because the completion rate on the nonspeculative pipeline is falling behind the issue rate rapidly, the nonspeculative pipeline will stall when a few more iterations are issued. The performance of the nonspeculative processor could be improved by allowing load instructions to complete effective address calculation before a branch is decided, but unless speculative memory accesses are allowed, this improvement will gain only one clock per iteration.

The above example clearly shows how speculation can be advantageous when there are data dependent branches, which otherwise would limit performance. This advantage depends, however, on accurate branch prediction. Incorrect speculation will not improve performance, but will, in fact, typically harm performance.

**Design Considerations for Speculative Machines**

In this section we briefly examine a number of important considerations that arise in speculative machines.

**Register renaming versus Reorder Buffers**

One alternative to the use of a ROB is the explicit use of a larger physical set of registers combined with register renaming. This approach builds on the concept of renaming used in Tomasulo’s algorithm, but extends it. In Tomasulo’s algorithm, the values of the *architecturally visible registers* (R0,..., R31 and
3.7 Hardware-Based Speculation

F0,...,F31) are contained, at any point in execution, in some combination of the register set and the reservation stations. With the addition of speculation, register values may also temporarily reside in the ROB. In either case, if the processor does not issue new instructions for a period of time, all existing instructions will commit, and the register values will appear in the register file, which directly corresponds to the architecturally visible registers.

In the register renaming approach, an extended set of physical registers is used to hold both the architecturally visible registers as well as temporary values. Thus, the extended registers replace the function both of the ROB and the reservation stations. During instruction issue, a renaming process maps the names of architectural registers to physical register numbers in the extended register set, allocating a new unused register for the destination. WAW and WAR hazards are avoided by renaming of the destination register, and speculation recovery is handled because a physical register holding an instruction destination does not become the architectural register until the instruction commits. The renaming map is a simple data structure that supplies the physical register number of the register that currently corresponds to the specified architectural register. This structure is similar in structure and function to the register status table in Tomasulo’s algorithm.

<table>
<thead>
<tr>
<th>Iter. #</th>
<th>Instructions</th>
<th>Issues at clock cycle #</th>
<th>Executes at clock cycle #</th>
<th>Memory access at clock cycle #</th>
<th>Write CDB at clock cycle #</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LW R2,0(R1)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>First issue</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R2,R2,#1</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td></td>
<td>Wait for LW</td>
</tr>
<tr>
<td>1</td>
<td>SW 0(R1),R2</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>4</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>1</td>
<td>DADDIU R1,R1,#4</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
<td>Execute directly</td>
</tr>
<tr>
<td>1</td>
<td>BNE R2,R3,LOOP</td>
<td>3</td>
<td>7</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>LW R2,0(R1)</td>
<td>4</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R2,R2,#1</td>
<td>4</td>
<td>11</td>
<td>12</td>
<td></td>
<td>Wait for LW</td>
</tr>
<tr>
<td>2</td>
<td>SW 0(R1),R2</td>
<td>5</td>
<td>9</td>
<td>13</td>
<td>9</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>2</td>
<td>DADDIU R1,R1,#4</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td></td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>2</td>
<td>BNE R2,R3,LOOP</td>
<td>6</td>
<td>13</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>LW R2,0(R1)</td>
<td>7</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R2,R2,#1</td>
<td>7</td>
<td>17</td>
<td>18</td>
<td></td>
<td>Wait for LW</td>
</tr>
<tr>
<td>3</td>
<td>SW 0(R1),R2</td>
<td>8</td>
<td>19</td>
<td>20</td>
<td>9</td>
<td>Wait for DADDIU</td>
</tr>
<tr>
<td>3</td>
<td>DADDIU R1,R1,#4</td>
<td>8</td>
<td>14</td>
<td>15</td>
<td></td>
<td>Wait for BNE</td>
</tr>
<tr>
<td>3</td>
<td>BNZ R2,R3,LOOP</td>
<td>9</td>
<td>19</td>
<td></td>
<td></td>
<td>Wait for DADDIU</td>
</tr>
</tbody>
</table>

**FIGURE 3.33** The time of issue, execution, and writing result for a dual-issue version of our pipeline without speculation. Note that the L.D following the BNE cannot start execution earlier, because it must wait until the branch outcome is determined. This type of program with data dependent branches that cannot be resolved earlier, shows the strength of speculation. Separate functional units for address calculation, ALU operations, and branch condition evaluation allow multiple instructions to execute in the same cycle.
One question you may be asking is: How do we ever know which registers are the architectural registers if they are constantly changing? Most of the time when the program is executing it does not matter. There are clearly cases, however, where another process, such as the operating system, must be able to know exactly where the contents of a certain architectural register resides. To understand how this capability is provided, assume the processor does not issue instructions for some period of time. Then eventually, all instructions in the pipeline will commit, and the mapping between the architecturally visible registers and physical registers will become stable. At that point, a subset of the physical registers contains the architecturally visible registers, and the value of any physical register not associated with an architectural register is unneeded. It is then easy to move the architectural registers to a fixed subset of physical registers so that the values can be communicated to another process.

An advantage of the renaming approach versus the ROB approach is that instruction commit is simplified, since it requires only two simple actions: record that the mapping between an architectural register number and physical register number is no longer speculative, and free up any physical registers being used to hold the “older” value of the architectural register. In a design with reservation stations, a station is freed up when the instruction using it completes execution, and a ROB is freed up when the corresponding instruction commits.

![Figure 3.34](image_url)

**FIGURE 3.34** The time of issue, execution, and writing result for a dual-issue version of our pipeline with speculation. Note that the L.D following the BNE can start execution early, because it is speculative.
With register renaming, deallocating registers is more complex, since before we free up a physical register, we must know that it no longer corresponds to an architectural register, and that no further uses of the physical register are outstanding. A physical register corresponds to an architectural register until the architectural register is rewritten, causing the renaming table to point elsewhere. That is, if no renaming entry points to a particular physical register, then it no longer corresponds to an architectural register. There may, however, still be uses of the physical register outstanding. The processor can determine whether this is the case by examining the source register specifiers of all instructions in the functional unit queues. If a given physical register does not appear as a source and it is not designated as an architectural register, it may be reclaimed and reallocated.

The process of reclamation can be simplified by counting the register source uses as instructions issue and decrementing the count as the instructions fetch their operands. When the count reaches zero, there are no further outstanding uses.

In addition to simplifying instruction commit, a renaming approach means that instruction issue need not examine both the ROB and the register file for an operand, since all results are in the register file. One possibly disconcerting aspect of the renaming approach is that the “real” architectural registers are never fixed but constantly change according to the contents of a renaming map. Although this complicates the design and debugging, it is not inherently problematic, and is an accepted fact in many newer implementations and sometimes even made architecturally visible, as we will see in the IA-64 architecture in the next chapter.

The PowerPC 603/604 series, the MIPS R1000/12000, the Alpha 21264, and the Pentium II, III and 4 all use register renaming, adding from 20 to 80 extra registers. Since all results are allocated a new virtual register until they commit, these extra registers replace a primary function of the ROB and largely determine how many instructions may be in execution (between issue and commit) at one time.

How much to speculate

One of the significant advantages of speculation is its ability to uncover events that would otherwise stall the pipeline early, such as cache misses. This potential advantage, however, comes with a significant potential disadvantage: the processor may speculate that some costly exceptional event occurs and begin processing the event, when in fact, the speculation was incorrect.

To maintain some of the advantage, while minimizing the disadvantages, most pipelines with speculation will allow only low-cost exceptional events (such as a first-level cache miss) to be handled in speculative mode. If an expensive exceptional event occurs, such as a second-level cache miss or a TLB miss, the processor will wait until the instruction causing the event is no longer speculative before handling the event. Although this may slightly degrade the performance of some programs, it avoids significant performance losses in others, especially those that suffer from a high frequency of such events coupled with less than excellent branch prediction.
Speculating through multiple branches
In the examples we have considered so far, it has been possible to resolve a branch before having to speculate on another. Three different situations can benefit from speculating on multiple branches simultaneously: a very high branch frequency, significant clustering of branches, and long delays in functional units. In the first two cases, achieving high performance may mean that multiple branches are speculated, and it may even mean handling more than one branch per clock. Database programs, and other less structured integer computations, often exhibit these properties, making speculation on multiple branches important. Likewise, long delays in functional units can raise the importance of speculating on multiple branches as a way to avoid stalls from the longer pipeline delays.

Speculating on multiple branches slightly complicates the process of speculation recovery, but is straightforward otherwise. A more complex technique is predicting and speculating on more than one branch per cycle. Although no existing processor has done this for general instruction execution as of 2000, we can expect that it may be needed in the future.

Of course, all the techniques described in the next chapter and in this one cannot take advantage of more parallelism than is provided by the application. The question of how much parallelism is available, and under what circumstances, has been hotly debated and is the topic of the next section.

3.8 Studies of the Limitations of ILP
Exploiting ILP to increase performance began with the first pipelined processors in the 1960s. In the 1980s and 1990s, these techniques were key to achieving rapid performance improvements. The question of how much ILP exists is critical to our long-term ability to enhance performance at a rate that exceeds the increase in speed of the base integrated-circuit technology. On a shorter scale, the critical question of what is needed to exploit more ILP is crucial to both computer designers and compiler writers. The data in this section also provide us with a way to examine the value of ideas that we have introduced in this chapter, including memory disambiguation, register renaming, and speculation.

In this section we review one of the studies done of these questions. The historical section (3.15) describes several studies, including the source for the data in this section (which is Wall’s 1993 study). All these studies of available parallelism operate by making a set of assumptions and seeing how much parallelism is available under those assumptions. The data we examine here are from a study that makes the fewest assumptions; in fact, the ultimate hardware model is probably unrealizable. Nonetheless, all such studies assume a certain level of compiler technology and some of these assumptions could affect the results, despite the
use of incredibly ambitious hardware. In addition, new ideas may invalidate the very basic assumptions of this and other studies; for example, value prediction, a technique we discuss at the end of this section, may allow us to overcome the limit of data dependences.

In the future, advances in compiler technology together with significantly new and different hardware techniques may be able to overcome some limitations assumed in these studies; however, it is unlikely that such advances when coupled with realistic hardware will overcome all these limits in the near future. Instead, developing new hardware and software techniques to overcome the limits seen in these studies will continue to be one of the most important challenges in computer design.

### The Hardware Model

To see what the limits of ILP might be, we first need to define an ideal processor. An ideal processor is one where all artificial constraints on ILP are removed. The only limits on ILP in such a processor are those imposed by the actual data flows either through registers or memory.

The assumptions made for an ideal or perfect processor are as follows:

1. **Register renaming**—There are an infinite number of virtual registers available and hence all WAW and WAR hazards are avoided and an unbounded number of instructions can begin execution simultaneously.

2. **Branch prediction**—Branch prediction is perfect. All conditional branches are predicted exactly.

3. **Jump prediction**—All jumps (including jump register used for return and computed jumps) are perfectly predicted. When combined with perfect branch prediction, this is equivalent to having a processor with perfect speculation and an unbounded buffer of instructions available for execution.

4. **Memory-address alias analysis**—All memory addresses are known exactly and a load can be moved before a store provided that the addresses are not identical.

Assumptions 2 and 3 eliminate all control dependences. Likewise, assumptions 1 and 4 eliminate all but the true data dependences. Together, these four assumptions mean that any instruction in the of the program’s execution can be scheduled on the cycle immediately following the execution of the predecessor on which it depends. It is even possible, under these assumptions, for the last dynamically executed instruction in the program to be scheduled on the very first cycle! Thus, this set of assumptions subsumes both control and address speculation and implements them as if they were perfect.

Initially, we examine a processor that can issue an unlimited number of instructions at once looking arbitrarily far ahead in the computation. For all the processor models we examine, there are no restrictions on what types of instruc-
tions can execute in a cycle. For the unlimited-issue case, this means there may be an unlimited number of loads or stores issuing in one clock cycle. In addition, all functional unit latencies are assumed to be one cycle, so that any sequence of dependent instructions can issue on successive cycles. Latencies longer than one cycle would decrease the number of issues per cycle, although not the number of instructions under execution at any point. (The instructions in execution at any point are often referred to as *in-flight*.)

Finally, we assume perfect caches, which is equivalent to saying that all loads and stores always complete in one cycle. This assumption allows our study to focus on fundamental limits to ILP. The resulting data, however, will be very optimistic, because realistic caches would significantly reduce the amount of ILP that could be successfully exploited, even if the rest of the processor were perfect!

Of course, this processor is on the edge of unrealizable. For example, the Alpha 21264 is one of the most advanced superscalar processors announced to date. The 21264 issues up to four instructions per clock and initiates execution on up to six (with significant restrictions on the instruction type, e.g., at most two load/stores), supports a large set of renaming registers (41 integer and 41 floating point, allowing up to 80 instructions in-flight), and uses a large tournament-style branch predictor. After looking at the parallelism available for the perfect processor, we will examine the impact of restricting various features.

To measure the available parallelism, a set of programs were compiled and optimized with the standard MIPS optimizing compilers. The programs were instrumented and executed to produce a trace of the instruction and data references. Every instruction in the trace is then scheduled as early as possible, limited only by the data dependences. Since a trace is used, perfect branch prediction and perfect alias analysis are easy to do. With these mechanisms, instructions may be scheduled much earlier than they would otherwise, moving across large numbers of instructions on which they are not data dependent, including branches, since branches are perfectly predicted.

Figure 3.35 shows the average amount of parallelism available for six of the SPEC92 benchmarks. Throughout this section the parallelism is measured by the average instruction issue rate (remember that all instructions have a one-cycle latency), which is the ideal IPC. Three of these benchmarks (fpppp, doduc, and tomcatv) are floating-point intensive, and the other three are integer programs. Two of the floating-point benchmarks (fpppp and tomcatv) have extensive parallelism, which could be exploited by a vector computer or by a multiprocessor (the structure in fpppp is quite messy, however, since some hand transformations have been done on the code). The doduc program has extensive parallelism, but the parallelism does not occur in simple parallel loops as it does in fpppp and tomcatv. The program li is a LISP interpreter that has many short dependences.

In the next few sections, we restrict various aspects of this processor to show what the effects of various assumptions are before looking at some ambitious but realizable processors.
Limitations on the Window Size and Maximum Issue Count

To build a processor that even comes close to perfect branch prediction and perfect alias analysis requires extensive dynamic analysis, since static compile-time schemes cannot be perfect. Of course, most realistic dynamic schemes will not be perfect, but the use of dynamic schemes will provide the ability to uncover parallelism that cannot be analyzed by static compile-time analysis. Thus, a dynamic processor might be able to more closely match the amount of parallelism uncovered by our ideal processor.

How close could a real dynamically scheduled, speculative processor come to the ideal processor? To gain insight into this question, consider what the perfect processor must do:

1. Look arbitrarily far ahead to find a set of instructions to issue, predicting all branches perfectly.
2. Rename all register uses to avoid WAR and WAW hazards.
3. Determine whether there are any data dependencies among the instructions in the issue packet; if so, rename accordingly.
4. Determine if any memory dependences exist among the issuing instructions and handle them appropriately.
5. Provide enough replicated functional units to allow all the ready instructions to issue.

Obviously, this analysis is quite complicated. For example, to determine whether \( n \) issuing instructions have any register dependences among them, assuming all instructions are register-register and the total number of registers is unbounded, requires...
comparisons. Thus, to detect dependences among the next 2000 instructions—the default size we assume in several figures—requires almost four million comparisons! Even issuing only 50 instructions requires 2450 comparisons. This cost obviously limits the number of instructions that can be considered for issue at once.

In existing and near-term processors, the costs are not quite so high, since we need only detect dependence pairs and the limited number of registers allows different solutions. Furthermore, in a real processor, issue occurs in-order and dependent instructions are handled by a renaming process that accommodates dependent renaming in one clock. Once instructions are issued, the detection of dependences is handled in a distributed fashion by the reservation stations or scoreboard.

The set of instructions that are examined for simultaneous execution is called the window. Each instruction in the window must be kept in the processor and the number of comparisons required every clock is equal to the maximum completion rate times the window size times the number of operands per instruction (today typically 6 x 80 x 2 = 960), since every pending instruction must look at every completing instruction for either of its operands. Thus, the total window size is limited by the required storage, the comparisons, and a limited issue rate, which makes larger window less helpful. To date, the window size has been in the range of 32 to 126, which can require over 2,000 comparisons. The HP PA 8600 reportedly has over 7,000 comparators!

The window size directly limits the number of instructions that begin execution in a given cycle. In practice, real processors will have a more limited number of functional units (e.g., no processor has handled more than two memory references per clock or more than two FP operations), as well as limited numbers of buses and register access ports, which serve as limits on the number of instructions initiated in the same clock. Thus, the maximum number of instructions that may issue, begin execution, or commit in the same clock cycle is usually much smaller than the window size.

Obviously, the number of possible implementation constraints in a multiple issue processor is large, including: issues per clock, functional units and unit latency, register file ports, functional unit queues (which may be fewer than units), issue limits for branches, and limitations on instruction commit. Each of these acts as constraint on the ILP. Rather than try to understand each of these effects, however, we will focus on limiting the size of the window, with the understanding that all other restrictions would further reduce the amount of parallelism that can be exploited.

Figures 3.36 and 3.37 show the effects of restricting the size of the window from which an instruction can execute; the only difference in the two graphs is the format—the data are identical. As we can see in Figure 3.36, the amount of
parallelism uncovered falls sharply with decreasing window size. In 2000, the most advanced processors have window sizes in the range of 64-128, but these window sizes are not strictly comparable to those shown in Figure 3.36 for two reasons. First, the functional units are pipelined, reducing the effective window size compared to the case where all units have single-cycle latency. Second, in real processors the window must also hold any memory references waiting on a cache miss, which are not considered in this model, since it assumes a perfect, single-cycle cache access.

As we can see in Figure 3.37, the integer programs do not contain nearly as much parallelism as the floating-point programs. This result is to be expected. Looking at how the parallelism drops off in Figure 3.37 makes it clear that the parallelism in the floating-point cases is coming from loop-level parallelism. The fact that the amount of parallelism at low window sizes is not that different among the floating-point and integer programs implies a structure where there are dependences within loop bodies, but few dependences between loop iterations in programs such as tomcatv. At small window sizes, the processors simply cannot see the instructions in the next loop iteration that could be issued in parallel with instructions from the current iteration. This case is an example of where better compiler technology (see the next chapter) could uncover higher amounts of ILP.
since it could find the loop-level parallelism and schedule the code to take advantage of it, even with small window sizes.

**FIGURE 3.37** The effect of window size shown by each application by plotting the average number of instruction issues per clock cycle. The most interesting observation is that at modest window sizes, the amount of parallelism found in the integer and floating-point programs is similar. *Artist: please add a data series to this graph. Legend label is 2K, this set of points goes between the Infinite and 512 series. The values of the points to add from top to bottom are: 36, 41, 15, 61, 59, 60. (So that, e.g., there will be a bar labeled 41, with the appropriate height as the second bar in the set corresponding to espresso.*
3.8 Studies of the Limitations of ILP

We know that large window sizes are impractical and inefficient, and the data in Figures 3.36 and 3.37 tell us that issue rates will be considerably reduced with realistic windows, thus we will assume a base window size of 2K entries and a maximum issue capability of 64 instructions per clock for the rest of this analysis. As we will see in the next few sections, when the rest of the processor is not perfect, a 2K window and a 64-issue limitation do not constrain the amount of ILP the processor can exploit.

The Effects of Realistic Branch and Jump Prediction

Our ideal processor assumes that branches can be perfectly predicted: The outcome of any branch in the program is known before the first instruction is executed! Of course, no real processor can ever achieve this. Figures 3.38 and 3.39 show the effects of more realistic prediction schemes in two different formats. Our data is for several different branch-prediction schemes varying from perfect to no predictor. We assume a separate predictor is used for jumps. Jump predictors are important primarily with the most accurate branch predictors, since the branch frequency is higher and the accuracy of the branch predictors dominates.

![Graph](image)

**FIGURE 3.38 The effect of branch-prediction schemes.** This graph shows the impact of going from a perfect model of branch prediction (all branches predicted correctly arbitrarily far ahead) to various dynamic predictors (selective and two-bit), to compile time, profile-based prediction, and finally to using no predictor. The predictors are described precisely in the text. **artist: change label “selective predictor” to “Tournament predictor”**
The five levels of branch prediction shown in these figures are

1. **Perfect**—All branches and jumps are perfectly predicted at the start of execution.

2. **Tournament-based branch predictor**—The prediction scheme uses a correlating two-bit predictor and a noncorrelating two-bit predictor together with a selector, which chooses the best predictor for each branch. The prediction buffer
3. Studies of the Limitations of ILP

contains $2^{13}$ (8K) entries, each consisting of three two-bit fields, two of which are predictors and the third is a selector. The correlating predictor is indexed using the exclusive-or of the branch address and the global branch history. The noncorrelating predictor is the standard two-bit predictor indexed by the branch address. The selector table is also indexed by the branch address and specifies whether the correlating or noncorrelating predictor should be used. The selector is incremented or decremented just as we would for a standard two-bit predictor. This predictor, which uses a total of 48K bits, outperforms both the correlating and noncorrelating predictors, achieving an average accuracy of 97% for these six SPEC benchmarks; this predictor is comparable in strategy and somewhat larger than the best predictors in use in 2000. Jump prediction is done with a pair of 2K-entry predictors, one organized as a circular buffer for predicting returns and one organized as a standard predictor and used for computed jumps (as in case statement or computed gotos). These jump predictors are nearly perfect.

3. **Standard two-bit predictor with 512 two-bit entries**—In addition, we assume a 16-entry buffer to predict returns.

4. **Static**—A static predictor uses the profile history of the program and predicts that the branch is always taken or always not taken based on the profile.

5. **None**—No branch prediction is used, though jumps are still predicted. Parallelism is largely limited to within a basic block.

Since we do not charge additional cycles for a mispredicted branch, the only effect of varying the branch prediction is to vary the amount of parallelism that can be exploited across basic blocks by speculation. Figure 3.40 shows the accuracy of the three realistic predictors for the conditional branches for the subset of SPEC92 benchmarks we include here. By comparison, Figure 3.61 on page 341 shows the size and type of branch predictor in recent high performance processors.

Figure 3.39 shows that the branch behavior of two of the floating-point programs is much simpler than the other programs, primarily because these two programs have many fewer branches and the few branches that exist are more predictable. This property allows significant amounts of parallelism to be exploited with realistic prediction schemes. In contrast, for all the integer programs and for doduc, the FP benchmark with the least loop-level parallelism, even the difference between perfect branch prediction and the ambitious selective predictor is dramatic. Like the window size data, these figures tell us that to achieve significant amounts of parallelism in integer programs, the processor must select and execute instructions that are widely separated. When branch prediction is not highly accurate, the mispredicted branches become a barrier to finding the parallelism.
As we have seen, branch prediction is critical, especially with a window size of 2K instructions and an issue limit of 64. For the rest of the studies, in addition to the window and issue limit, we assume as a base a more ambitious tournament predictor that uses two levels of prediction and a total of 8K entries. This predictor, which requires more than 150K bits of storage (roughly four times the largest predictor to date), slightly outperforms the selective predictor described above (by about 0.5–1%). We also assume a pair of 2K jump and return predictors, as described above.

The Effects of Finite Registers

Our ideal processor eliminates all name dependences among register references using an infinite set of physical registers. To date, the Alpha 21264 has provided the largest number of extended registers: 41 integer and 41 FP registers, in addition to 32 integer and 32 floating point architectural registers. Figures 3.41 and 3.42 show the effect of reducing the number of registers available for renaming, again using the same data in two different forms. Both the FP and GP registers are increased by the number of registers shown on the axis or in the legend.

At first, the results in these figures might seem somewhat surprising: you might expect that name dependences should only slightly reduce the parallelism avail-
able. Remember though, exploiting large amounts of parallelism requires evaluating many independent threads of execution. Thus, many registers are needed to hold live variables from these threads. Figure 3.41 shows that the impact of having only a finite number of registers is significant if extensive parallelism exists. Although these graphs show a large impact on the floating-point programs, the impact on the integer programs is small primarily because the limitations in window size and branch prediction have limited the ILP substantially, making renaming less valuable. In addition, notice that the reduction in available parallelism is significant even if 64 additional integer and 64 additional FP registers are available for renaming, which is more than the number of extra registers available on any existing processor as of 2000.

Although register renaming is obviously critical to performance, an infinite number of registers is obviously not practical. Thus, for the next section, we assume that there are 256 integer and 256 FP registers available for renaming—far more than any anticipated processor has.

FIGURE 3.41 The effect of finite numbers of registers available for renaming. Both the number of FP registers and the number of GP registers are increased by the number shown on the x axis. The effect is most dramatic on the FP programs, although having only 32 extra GP and 32 extra FP registers has a significant impact on all the programs. As stated earlier, we assume a window size of 2K entries and a maximum issue width of 64 instructions. None implies no extra registers available.
The Effects of Imperfect Alias Analysis

Our optimal model assumes that it can perfectly analyze all memory dependences, as well as eliminate all register name dependences. Of course, perfect alias analysis is not possible in practice: The analysis cannot be perfect at com-
3.8 Studies of the Limitations of ILP

pile time, and it requires a potentially unbounded number of comparisons at runtime (since the number of simultaneous memory references is unconstrained). Figures 3.43 and 3.44 show the impact of three other models of memory alias analysis, in addition to perfect analysis. The three models are:

1. **Global/stack perfect**—This model does perfect predictions for global and stack references and assumes all heap references conflict. This model represents an idealized version of the best compiler-based analysis schemes currently in production. Recent and ongoing research on alias analysis for pointers should improve the handling of pointers to the heap in the future.

2. **Inspection**—This model examines the accesses to see if they can be determined not to interfere at compile time. For example, if an access uses R10 as a base register with an offset of 20, then another access that uses R10 as a base register with an offset of 100 cannot interfere. In addition, addresses based on registers that point to different allocation areas (such as the global area and the stack area) are assumed never to alias. This analysis is similar to that performed by many existing commercial compilers, though newer compilers can
do better, at least for loop-oriented programs.

3. *None*—All memory references are assumed to conflict.

As one might expect, for the FORTRAN programs (where no heap references exist), there is no difference between perfect and global/stack perfect analysis. The global/stack perfect analysis is optimistic, since no compiler could ever find all array dependences exactly. The fact that perfect analysis of global and stack references is still a factor of two better than inspection indicates that either sophisticated compiler analysis or dynamic analysis on the fly will be required to obtain much parallelism. In practice, dynamically scheduled processors rely on dynamic memory disambiguation and are limited by three factors:
3.9 Limitations on ILP for Realizable Processors

1. To implement perfect dynamic disambiguation for a given load, we must know the memory addresses of all earlier stores that not yet committed, since a load may have a dependence through memory on a store. One technique for reducing this limitation on in-order address calculation is memory address speculation. With memory address speculation, the processor either assumes that no such memory dependences exist or uses a hardware prediction mechanism to predict if a dependence exists, stalling the load if a dependence is predicted. Of course, the processor can be wrong about the absence of the dependence, so we need a mechanism to discover if a dependence truly exists and to recover if so. To discover if a dependence exists, the processor examines the destination address of each completing store that is earlier in program order than the given load. If a dependence that should have been enforced occurs, the processor uses the speculative restart mechanism to redo the load and the following instructions. (We will see how this type of address speculation can be supported with instruction set extensions in the next chapter.)

2. Only a small number of memory references can be disambiguated per clock cycle.

3. The number of the load/store buffers determines how much earlier or later in the instruction stream a load or store may be moved.

Both the number of simultaneous disambiguations and the number of the load/store buffers will affect the clock cycle time.

3.9 Limitations on ILP for Realizable Processors

In this section we look at the performance of processors ambitious levels of hardware support equal to or better than what is likely in the next five years. In particular we assume the following fixed attributes:

1. Up to 64 instruction issues per clock with no issue restrictions. As we discuss later, the practical implications of very wide issue widths on clock rate, logic complexity, and power may be the most important limitation on exploiting ILP.

2. A tournament predictor with 1K entries and a 16-entry return predictor. This predictor is fairly comparable to the best predictors in 2000; the predictor is not a primary bottleneck.

3. Perfect disambiguation of memory references done dynamically—this is ambitious but perhaps attainable for small window sizes (and hence small issue rates and load/store buffers) or through a memory dependence predictor.

4. Register renaming with 64 additional integer and 64 additional FP registers,
exceeding largest number available on any processor in 2001 (41 and 41 in the Alpha 21264), but probably easily reachable within two or three years.

Figures 3.45 and 3.46 show the result for this configuration as we vary the window size. This configuration is more complex and expensive than any existing implementations, especially in terms of the number of instruction issues, which is more than ten times larger than the largest number of issues available on any processor in 2001. Nonetheless, it gives a useful bound on what future implementations might yield. The data in these figures is likely to be very optimistic for another reason. There are no issue restrictions among the 64 instructions: they may all be memory references. No one would even contemplate this capability in a processor in the near future. Unfortunately, it is quite difficult to bound the performance of a processor with reasonable issue restrictions; not only is the space of possibilities quite large, but the existence of issue restrictions requires that the parallelism be evaluated with an accurate instruction scheduler, making the cost of studying processors with large numbers of issues very expensive.

In addition, remember that in interpreting these results, cache misses and non-unit latencies have not been taken into account, and both these effects will have significant impact (see the Exercises).

Figure 3.45 shows the parallelism versus window size. The most startling observation is that with the realistic processor constraints listed above, the effect of the window size for the integer programs is not so severe as for FP programs. This result points to the key difference between these two types of programs. The availability of loop-level parallelism in two of the FP programs means that the amount of ILP that can be exploited is higher, but that for integer programs other factors—such as branch prediction, register renaming, and less parallelism to start with—are all important limitations. This observation is critical, because of the increased emphasis on integer performance in the last few years. As we will see in the next section, for a realistic processor in 2000, the actual performance levels are much lower than those shown in Figure 3.45.

Given the difficulty of increasing the instruction rates with realistic hardware designs, designers face a challenge in deciding how best to use the limited resources available on a integrated circuit. One of the most interesting trade-offs is between simpler processors with larger caches and higher clock rates versus more emphasis on instruction-level parallelism with a slower clock and smaller caches. The following Example illustrates the challenges.

**EXAMPLE** Consider the following three hypothetical, but not atypical, processors, which we run with the SPEC gcc benchmark:

1. A simple MIPS two-issue static pipe running at a clock rate of 1 GHz
3.9 Limitations on ILP for Realizable Processors

and achieving a pipeline CPI of 1.0. This processor has a cache system that yields 0.01 misses per instruction.

2. A deeply pipelined version of MIPS with slightly smaller caches and a 1.2 GHz clock rate. The pipeline CPI of the processor is 1.2, and the smaller caches yield 0.015 misses per instruction on average.

3. A speculative superscalar with a 64-entry window. It achieves one-half of the ideal issue rate measured for this window size. (Use the data in Figure 3.45 on page 311.) This processor has the smallest caches, which leads to 0.02 misses per instruction, but it hides 10% of the miss penalty on every miss by dynamic scheduling. This processor has a 800-MHz clock.

Assume that the main memory time (which sets the miss penalty) is 100 ns. Determine the relative performance of these three processors.

ANSWER First, we use the miss penalty and miss rate information to compute the contribution to CPI from cache misses for each configuration. We do this with the following formula:
We need to compute the miss penalties for each system:

Cache CPI = Misses per instruction × Miss penalty

We need to compute the miss penalties for each system:
The clock cycle times for the processors are 1 ns, 0.83 ns, and 1.25 ns, respectively. Hence, the miss penalties are

\[
\text{Miss penalty}_1 = \frac{100 \text{ ns}}{1 \text{ ns}} = 100 \text{ cycles}
\]

\[
\text{Miss penalty}_2 = \frac{100 \text{ ns}}{0.83 \text{ ns}} = 120 \text{ cycles}
\]

\[
\text{Miss penalty}_3 = \frac{0.9 \times 100 \text{ ns}}{1.25 \text{ ns}} = 72 \text{ cycles}
\]

Applying this for each cache:

- Cache CPI$_1 = 0.01 \times 100 = 1.0$
- Cache CPI$_2 = 0.015 \times 120 = 1.8$
- Cache CPI$_3 = 0.02 \times 72 = 1.44$

We know the pipeline CPI contribution for everything but processor 3; its pipeline CPI is given by

\[
\text{Pipeline CPI}_3 = \frac{1}{\text{Issue rate}} = \frac{1}{9 \times 0.5} = \frac{1}{4.5} = 0.22
\]

Now we can find the CPI for each processor by adding the pipeline and cache CPI contributions.

- CPI$_1 = 1.0 + 1.0 = 2.0$
- CPI$_2 = 1.2 + 1.8 = 3.0$
- CPI$_3 = 0.22 + 1.44 = 1.66$

Since this is the same architecture we can compare instruction execution rates to determine relative performance:

\[
\text{Instruction execution rate}_1 = \frac{1000 \text{ MHz}}{2} = 500 \text{ MIPS}
\]

\[
\text{Instruction execution rate}_2 = \frac{1200 \text{ MHz}}{3.0} = 400 \text{ MIPS}
\]

\[
\text{Instruction execution rate}_3 = \frac{800 \text{ MHz}}{1.66} = 482 \text{ MIPS}
\]

In this example, the moderate issue processor looks best. Of course, the designer building either system 2 or system 3 will probably be alarmed by
the large fraction of the system performance lost to cache misses. In the Chapter 5 we'll see the most common solution to this problem: adding another level of caches.

Beyond the limits of this study

Like any limit study, the study we have examined in this section has its own limitations. We divide these into two classes: limitations that arise even for the perfect speculative processor and limitations that arise for one or more realistic models. Of course, all the limitations in the first class apply to the second. The most important limitations that apply even to the perfect model are:

1. **WAR and WAW hazards through memory**: the study eliminated WAW and WAR hazards through register renaming, but not in memory usage. Although, at first glance it might appear that such circumstances are rare (especially WAW hazards), they arise due to the allocation of stack frames. A called procedure reuses the memory locations of a previous procedure on the stack and this can lead to WAW and WAR hazards that are unnecessarily limiting. Austin and Sohi’s 1992 paper examines this issue.

2. **Unnecessary dependences**: with infinite numbers of registers, all but true register data dependences are removed. There are, however, dependences arising from either recurrences or code generation conventions that introduce unnecessary true data dependences. One example of these is the dependence on the control variable in a simple do-loop: since the control variable is incremented on every loop iteration, the loop contains at least one dependence. As we show in the next chapter, loop unrolling and aggressive algebraic optimization can remove such dependent computation. Wall’s study includes a limited amount of such optimizations, but applying them more aggressively could lead to increased amounts of ILP. In addition, certain code generation conventions introduce unneeded dependences, in particular the use of return address registers and a register for the stack pointer (which is incremented and decremented in the call/return sequence). Wall removes the effect of the return address register, but the use of a stack pointer in the linkage convention can cause “unnecessary” dependences. Postiff, Greene, Tyson, and Mudge explored the advantages of removing this constraint in a 1999 paper.

3. **Overcoming the data flow limit**: a recent proposed idea to boost ILP, which goes beyond the capability of the study above, is *value prediction*. Value prediction consists of predicting data values and speculating on the prediction. There are two obvious uses of this scheme: predicting data values and speculating on the result and predicting address values for memory alias elimination. The latter affects parallelism only under less than perfect circumstances,
as we discuss shortly.

Value prediction has possibly the most potential for increasing ILP. *Data value prediction and speculation* predicts data values and uses them in destination instructions speculatively. Such speculation allows multiple dependent instructions to be executed in the same clock cycle, thus increasing the potential ILP. To be effective, however, data values must be predicted very accurately, since they will be used by consuming instructions, just as if they were correctly computed. Thus, inaccurate prediction will lead to incorrect speculation and recovery, just as when branches are mispredicted.

One insight that gives some hope is that certain instructions produce the same values with high frequency, so it may be possible to selectively predict values for certain instructions with high accuracy. Obviously, perfect data value prediction would lead to infinite parallelism, since every value of every instruction could be predicted a priori.

Thus, studying the effect of value prediction in true limit studies is difficult and has not yet been done. Several studies have examined the role of value prediction in exploiting ILP in more realistic processors (e.g., Lipasti, Wilkerson, and Shen in 1996). The extent to which general value prediction will be used in real processors remains unclear at the present.

For a less than perfect processor, there are several ideas, which have been proposed, that could expose more ILP. We mention the two most important here:

1. *Address value prediction and speculation* predicts memory address values and speculates by reordering loads and stores. This technique eliminates the need to compute effective addresses in-order to determine whether memory references can be reordered, and could provide better aliasing analysis than any practical scheme. Because we need not actually predict data values, but only if effective addresses are identical, this type of prediction can be accomplished by simpler techniques. Recent processors include limited versions of this technique and it can be expected that future implementations of address value prediction may yield an approximation to perfect alias analysis, allowing processors to eliminate this limit to exploiting ILP.

2. Speculating on multiple paths: this idea was discussed by Lam and Wilson in 1992 and explored in the study covered in this section. By speculating on multiple paths, the cost of incorrect recovery is reduced and more parallelism can be uncovered. It only makes sense to evaluate this scheme for a limited number of branches, because the hardware resources required grow exponentially. Wall’s 1993 study provides data for speculating in both directions on up to eight branches. Whether such schemes ever become practical, or whether it will always be better to devote the equivalent silicon area to better branch predictors remains to be seen. In Chapter 8, we discuss thread-level parallelism and the use of speculative threads.

It is critical to understand that none of the limits in this section are fundamental in the sense that overcoming them requires a change in the laws of physics!
Instead, they are practical limitations that imply the existence of some formidable barriers to exploiting additional ILP. These limitations—whether they be window size, alias detection, or branch prediction—represent challenges for designers and researchers to overcome! As we discuss in the concluding remarks, there are a variety of other practical issues that may actually be the more serious limits to exploiting ILP in future processors.

### 3.10 Putting It All Together: The P6 Microarchitecture

The Intel P6 microarchitecture forms the basis for the Pentium Pro, Pentium II, and the Pentium III. In addition to some specialized instruction set extensions (MMX and SSE), these three processors differ in clock rate, cache architecture, and memory interface and is summarized in Figure 3.47.

<table>
<thead>
<tr>
<th>Processor</th>
<th>First ship date</th>
<th>Clock rate range</th>
<th>L1 cache</th>
<th>L2 cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium Pro</td>
<td>1995</td>
<td>100–200 MHz</td>
<td>8KB instr. + 8KB data</td>
<td>256 KB–1,024 KB</td>
</tr>
<tr>
<td>Pentium II</td>
<td>1998</td>
<td>233–450 MHz</td>
<td>16KB instr. + 16KB data</td>
<td>256 KB–512 KB</td>
</tr>
<tr>
<td>Pentium II Xenon</td>
<td>1999</td>
<td>400–450 MHz</td>
<td>16KB instr. + 16KB data</td>
<td>512 KB–2 MB</td>
</tr>
<tr>
<td>Celeron</td>
<td>1999</td>
<td>500–900 MHz</td>
<td>16KB instr. + 16KB data</td>
<td>128 KB</td>
</tr>
<tr>
<td>Pentium III</td>
<td>1999</td>
<td>450–1,100 MHz</td>
<td>16KB instr. + 16KB data</td>
<td>256 KB–512 KB</td>
</tr>
<tr>
<td>Pentium III Xenon</td>
<td>2000</td>
<td>700–900 MHz</td>
<td>16KB instr. + 16KB data</td>
<td>1 MB–2 MB</td>
</tr>
</tbody>
</table>

**FIGURE 3.47 The Intel processors based on the P6 microarchitecture and their important differences.** In the Pentium Pro, the processor and specialized cache SRAMs were integrated into a multichip module. In the Pentium II standard SRAMs are used. In the Pentium III, there is either on-chip 256 KB L2 cache or an off-chip 512 KB cache. The Xenon version are intended for server applications; they use an off-chip L2 and support multiprocessing. The Pentium II added the MMX instruction extension, while the Pentium III added the SSE extensions.

The P6 microarchitecture is a dynamically scheduled processor that translates each IA-32 instruction to a series of micro-operations (uops) that are executed by the pipeline; the uops are similar to typical RISC instructions. Up to three IA-32 instructions are fetched, decoded, and translated into uops every clock cycle. If an IA-32 instruction requires more than four uops, it is implemented by a micro-coded sequence that generates the necessary uops in multiple clock cycles. The maximum number of uops that may be generated per clock cycle is six, with four allocated to the first IA-32 instruction, and one uop slot to each of the remaining two IA-32 instructions.

The uops are executed by an out-of-order speculative pipeline using register renaming and a ROB. This pipeline is very similar to that in section 3.7, except that the functional unit capability and the sizes of buffers are different. Up to three uops per clock can be renamed and dispatched to the reservation stations; instruction commit can also complete up to three uops per clock. The pipeline is structured in 14 stages composed of the following:
8 stages are used for in-order instruction fetch, decode, and dispatch. The next instruction is selected during fetch using a 512-entry, two-level branch predictor. The decode and issue stages including register renaming (using 40 virtual registers) and dispatch to one of 20 reservation stations and to one of 40 entries in the ROB.

3 stages are used for out-of-order execution in one of five separate functional units (integer unit, FP unit, branch unit, memory address unit, and memory access unit). The execution pipeline is from 1 cycle (for simple integer ALU operations) to 32 cycles for FP divide. The issue rate and latency of some typical operations appears in Figure 3.48.

3 stages are used for instruction commit.

3.10 Putting It All Together: The P6 Microarchitecture

<table>
<thead>
<tr>
<th>Instruction name</th>
<th>Pipeline stages</th>
<th>Repeat rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer ALU</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Integer load</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Integer multiply</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>FP add</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>FP multiply</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>FP divide (64-bit)</td>
<td>32</td>
<td>32</td>
</tr>
</tbody>
</table>

FIGURE 3.48 The latency and repeat rate for common uops in the P6 microarchitecture. A repeat rate of 1 means that the unit is fully pipelined, and a repeat rate of 2 means that operations can start every other cycle.

Figure 3.49 shows a high-level picture of the pipeline, the throughput of each stage, and the capacity of buffers between stages. A stage will not achieve its throughput if either the input buffer cannot supply enough operands or the output buffer lacks capacity. In addition, internal restrictions or dynamic events (such as a cache miss) can cause a stall within all the units. For example, an instruction cache miss will prevent the instruction fetch stage from generating 16 bytes of instructions; similarly, three instructions can be decoded only under certain restrictions in how they map to uops.

Performance of the Pentium Pro Implementation

This section looks at some performance measurements for the Pentium Pro implementation. The Pentium Pro has the smallest set of primary caches among the P6 based microprocessors; it has, however, a high bandwidth interface to the secondary caches. Thus, while we would expect more performance to be lost to cache misses than on the Pentium II, the relatively faster and higher bandwidth secondary caches should reduce this effect somewhat. The measurements in this section use a 200 MHz Pentium Pro with a 256KB secondary cache and a 66 MHz main memory bus. The data for this section comes from a study by Bhandarkar and Ding [1997] that uses SPEC CPU95 as the benchmark set.
Understanding the performance of a dynamically-scheduled processor is complex. To see why, consider first that the actual CPI will be significantly greater than the ideal CPI, which in the case of the P6 architecture is 0.33. If the effective CPI is, for example, 0.66, then the processor can fall behind, achieving an CPI of 1, during some part of the execution and subsequently catch up by issuing and graduating two instructions per clock. Furthermore, consider how stalls actually occur in dynamically-scheduled, speculative processors. Since cache misses are overlapped, branches outcomes are speculated, and data dependences are dynamically scheduled around, what does a stall actually mean? In the limit, stalls occur when the processor fails to commit its full complement of instructions in a clock cycle.

Of course, the lack of instructions to complete means that somewhere earlier in the pipeline, some instructions failed to make progress (or in the limit, failed to even issue). This blockage can occur for a combination of several reasons in the Pentium Pro:

1. Less than a three IA-32 instructions could be fetched, due to instruction cache misses.
2. Less than three instructions could issue, because one of the three IA-32 instructions generated more than the allocated number of uops (4 for the first instruction and 1 for each of other two).
3. Not all the microoperations generated in a clock cycle could issue because of a shortage of reservation stations or reorder buffers.
4. A data dependence led to a stall because every reservations station or the reorder buffer was filled with instructions that are dependent.
5. A data cache misses led to a stall because every reservation station or the reorder buffer was filled with instructions waiting for a cache miss.

**Figure 3.49** The P6 processor pipeline showing the throughput of each stage and the total buffering provided between stages. The buffering provided is either as bytes (before instruction decoding), as uops (after decoding and translation), as reservation station entries (after issue), or as reorder buffer entries (after execution). There are five execution units, each of which can potentially initiate a new uop every cycle (though some are not fully pipelined as shown in Figure 3.48). Recall that during renaming an instruction reserves a reorder buffer entry, so that stalls can occur during renaming/issue when the reorder buffer is full. Notice that the instruction fetch unit can fill the entire prefetch buffer in one cycle; if the buffer is partially full, fewer bytes will be fetched.
6. Branch mispredicts cause stalls directly, since the pipeline will need to be flushed and refilled. A mispredict can also cause a stall that arises from interference between speculated instructions that will be canceled and instructions that will be completed.

Because of the ability to overlap potential stall cycles from multiple sources, it is difficult to assign the cost of a stall cycle to any single cause. Instead, we will look at the contributions to stalls and conclude by showing that the actual CPI is less than what would be observed if no overlap of stalls were possible.

**FIGURE 3.50** The number of instructions decoded each clock varies widely and depends upon a variety of facts including the instruction cache miss rate, the instruction decode rate, and the downstream execution rate. On average for these benchmarks, 0.87 instructions are decoded per cycle.

**Stalls in the Decode Cycle**
To start, let’s look at the rate at which instructions are fetched and issued. Although the processor attempts to fetch three instructions every cycle, it cannot maintain this rate if the instruction cache generates a miss, if one of the instruc-
tions requires more than the number of microoperations available to it or if the six-entry uop issue buffer is full. Figure 3.50 shows the fraction of time in which 0, 1, 2, or 3 IA-32 instructions are decoded.

Figure 3.51 breaks out the stalls at decode time according to whether they are due to instruction cache stalls, which lead to less than three instructions available to decode, or resource capacity limitations, which means that a lack of reservation station or reorder buffers prevents a uop from issuing. Failure to issue a uop, eventually leads to a full uop buffer (recall that it has six entries), which then blocks instruction decode.

**FIGURE 3.51** Stall cycles per instruction at decode time and the breakdown due to instruction stream stalls, which occur because of instruction cache misses, or resource capacity stalls, which occur because of a lack of reservation stations or reorder buffer entries. SPEC CPU95 is used as the benchmark suite, for this, and the rest of the measurements in this section.
The instruction cache miss rate for the SPEC95 FP benchmarks is small, and, for most of the FP benchmarks, resource capacity is the primary cause of decode stalls. The resource limitation arises because of lack of progress further down the pipeline, due either to large numbers of dependent operations or to long latency operations; the latter is a limitation for floating point programs, in particular. For example, the programs su2cor and hydro2d, which both have large numbers of resource stalls, also have long running, dependent floating-point calculations.

Another possible reason for the reduction in decode throughput could be that the expansion of IA-32 instructions into uops causes the uop buffer to fill. This would be the case if the number of uops per IA-32 instruction were large. Figure 3.52 shows, however, that most IA-32 instructions map to a single uop, and that on average there are 1.37 microoperations per IA-32 instruction (which means that the CPI for the processor is 1.37 times higher than the CPI of the microoperations). Surprisingly, the integer programs take slightly more microoperations per IA-32 instruction on average than the floating-point programs!

![Graph showing microoperations per IA-32 instruction](image)

**FIGURE 3.52** The number of microoperations per IA-32 instruction. Other than fpppp, the integer programs typically require more uops. Most instructions will take only one uop, and, thus, the uop buffer fills primarily because of delays in the execution unit.

Data Cache Behavior

Figure 3.53 shows the number of first level (L1) and second level (L2) cache misses per thousand instructions. The L2 misses, although smaller in number, cost more than five times as much as L1 misses, and thus, dominate in some applications. Instruction cache misses are a minor effect in most of the programs. Although the speculative, out-of-order pipeline may be effective at hiding stalls
due to L1 data misses, it cannot hide the long latency L2 cache misses, and L2 miss rates and effective CPI track similarly.

Branch Performance and Speculation Costs

Branch target addresses are predicted with a 512-entry BTB, based on the two-level adaptive scheme of Yeh and Patt, which is similar to the predictor described on page 258. If the BTB does not hit, a static prediction is used: backward branches are predicted taken (and have a one cycle penalty if correctly predicted) and forward branches are predicted not taken (and have no penalty if correctly predicted). Branch mispredicts have both a direct performance penalty, which is between 10-15 cycles, and an indirect penalty due to the overhead of incorrectly
speculated instructions, which is essentially impossible to measure. (Sometimes misspeculated instructions can result in a performance advantage, but this is likely to be rare.) Figure 3.54 shows the fraction of branches mispredicted either because of BTB misses or because of incorrect predictions. On average about 20% of the branches either miss or are mispredicted and use the simple static predictor rule.

To understand the secondary effects arising from speculation that will be canceled, Figure 3.53 plots the average number of speculated uops that do not commit. On average about 1.2 times as many uops issue as commit. By factoring in the branch frequency and the mispredict rates, we find that, on average, each mispredicted branch issues 20 uops that will later be canceled. Unfortunately, accessing the exact costs of incorrectly speculated operations is virtually impossible, since they may cost nothing (if they do not block the progress of other instructions) or may be very costly.

Putting the Pieces Together: Overall Performance of the P6 Pipeline
Overall performance depends on the rate at which instructions actually complete and commit. Figure 3.56 shows the fraction of the time that 0, 1, 2, or 3 uops
commit. On average, one uop commits per cycle, but, as Figure 3.56 shows, 23% of the time 3 uops commit in a cycle. This distribution demonstrates the ability of a dynamically-scheduled pipeline to fall behind (on 55% of the cycles, no uops commit) and later catch up (31% of the cycles have 2 or 3 uops committing).

Figure 3.57 sums up all the possible issue and stall cycles per IA-32 instruction and compares it to the actual measured CPI on the processor. The uop cycles in Figure 3.57 are the number of cycles per instruction assuming that the processor sustains three uops per cycle and accounting for the number of uops required per IA-32 instruction for that benchmark. The sum of the issue cycles plus stalls exceeds the actual measured CPI by an average of 1.37, varying from 1.0 to 1.75. This difference arises from the ability of the dynamically-scheduled pipeline to overlap and hide different classes of stalls arising in different types of programs. The average CPI is 1.15 for the SPECint programs and 2.0 for the SPECFP programs. The P6 microarchitecture is clearly designed to focus on integer programs.

The Pentium III versus the Pentium 4

The microarchitecture of the Pentium 4, which is called NetBurst, is similar to that of the Pentium III (called the P6 microarchitecture): both fetch up to three IA-32 instructions per cycle, decode them into micro-ops, and send the uops to an...
3.10 Putting It All Together: The P6 Microarchitecture

out-of-order execution engine that can graduate up to three uops per cycle. There
are, however, many differences that are designed to allow the NetBurst microarchi-
cecture to operate at a significantly higher clock rate than the P6 microarchi-
tecture and to help maintain or close the peak to sustained execution throughput.
Among the most important of these are:

- A much deeper pipeline: P6 requires about 10 clock cycles from the time a sim-
  ple add instruction is fetched until the availability of its results. In comparison,
  NetBurst takes about 20 cycles, including 2 cycles reserved simply to drive re-
  sults across the chip!

- NetBurst uses register renaming (like the MIPS R10K and the Alpha 21264)
  rather than the reorder buffer, which is used in P6. Use of register renaming al-
  lows many more outstanding results (potentially up to 128) in NetBurst versus
  the 40 that are permitted in P6.

- There are seven integer execution units in NetBurst versus five in P6. The ad-
  ditions are an additional integer ALU and an additional address computation
  unit.

FIGURE 3.56 The breakdown in how often 0, 1, 2, or 3 uops commit in a cycle. The average number of uop comple-
tions per cycle is distributed as: 0 completions 55% of the cycles, 1 completion 13% of the cycles, 2 completions 8% of
the cycles, and 3 completions 23% of the cycles,
An aggressive ALU (operating at twice the clock rate) and an aggressive data cache lead to lower latencies for the basic ALU operations (effectively one-half a clock cycle in NetBurst versus one in P6) and for data loads (effectively two cycles in NetBurst versus three in P6). These high-speed functional units are critical to lowering the potential increase in stalls from the very deep pipeline.

NetBurst uses a sophisticated trace cache (see Chapter 5) to improve instruction fetch performance, while P6 uses a conventional prefetch buffer and instruction cache.

Netburst has a branch target buffer that is eight times larger and has an improved prediction algorithm.
NetBurst has a level 1 data cache that is 8KB compared to P6’s 16KB L1 data cache. NetBurst’s larger level two cache (256KB) with higher bandwidth should offset this disadvantage.

NetBurst implements the new SSE2 floating point instructions that allow two floating operations per instruction; these operations are structured as a 128-bit SIMD or short-vector structure. As we saw in Chapter 1 this gives Pentium 4 a considerable advantage over Pentium III on floating point code.

A Brief Performance Comparison of the Pentium III and Pentium 4
As we saw in Figure 1.28 on page 60 the Pentium 4 at 1.7 Ghz outperforms the Pentium III at 1 GHz by a factor of 1.26 for SPEC CINT2000 and 1.8 for SPEC CFP2000. Figure 3.58 shows the performance of the Pentium III and Pentium 4 on four of the SPEC benchmarks that are in both SPEC95 and SPEC2000. The floating point benchmarks clearly take advantage of the new instruction set extensions and yield an advantage of 1.6–1.7 above clock rate scaling.

![Performance Comparison Chart](chart.png)

**FIGURE 3.58** The performance of the Pentium 4 for four SPEC2000 benchmarks (two integer: gcc and vortex, and two floating point: applu and mgrid) exceeds the Pentium III by a factor of between 1.2 and 2.9. This exceeds the purely clock speed advantage for the floating point benchmarks and is less than the clock speed advantage for the integer programs.

For the two integer benchmarks, the situation is somewhat different. In both cases the Pentium 4 delivers less than linear scaling with the increase in clock rate. If we assume the instruction counts are identical for integer codes on the two
processors, then the CPI for the two integer benchmarks is higher on the Pentium 4 (by a factor of 1.1 for gcc and a factor of 1.5 for vortex). Looking at the data for the Pentium Pro, we can see that the these benchmarks have relatively low level-2 miss rates and that they hide much of their level-1 miss penalty through dynamic scheduling and speculation. Thus, it is likely that the deeper pipeline and larger pipeline stall penalties on the Pentium 4 lead to a higher CPI for these two programs and reduce some of the gain from the high clock rate.

One interesting question is: why did the designers at Intel decide on the approach they took for the Pentium 4? On the surface, the alternative of doubling the issue rate of the Pentium III, as opposed to doubling the pipeline depth and the clock rate, looks at least as attractive. Of course, there are numerous changes between the two architectures, making an exact analysis of the tradeoffs difficult. Furthermore, because of the changes in the floating point instruction set, a comparison of the two pipeline organizations needs to focus on integer performance.

There are two sources of performance loss that arise if we compare the deeper pipeline of the Pentium 4 with that of the Pentium III. The first is the increase in clock overhead that occurs due to increased clock skew and jitter. This overhead is given by the difference between the ideal clock speed and the achieved clock speed. In comparable technologies, the Pentium 4 clock rate is between 1.7 and 1.8 times higher than the Pentium III clock rate. This range represents between 85% and 90% of the ideal clock rate, which is 2 times higher.

The second source of performance loss is the increase in CPI that arises from the deeper pipeline. We can estimate this by taking the ratio in clock rate versus the ratio in achieved overall performance. Using SPECInt as the performance measure and comparing a 1 GHz Pentium III to a 1.7 GHz Pentium 4, the performance ratio is 1.26. This tells us that the CPI for SPECInt on the Pentium 4 must be $1.7 / 1.26 = 1.34$ times higher., or alternatively that the Pentium 4 is about 1.26/1.7 = 74% of the efficiency of the Pentium III. Of course, some of this loss is in the memory system, rather than in the pipeline.

The key question is whether doubling the issue width would result in a greater than 1.26 times overall performance gain. This is a very difficult question to answer, since we must account for the improvement in pipeline CPI, the relative increase in cost of memory stalls, and the potential clock rate impact of a processor with twice the issue width. It is unlikely, looking at the data in Section 3.9, that doubling the issue rate will achieve better than a factor of 1.5 improvement in ideal instruction throughput. When combined with the potential impact on clock rate and the memory system costs, it appears that the choice of the Intel Pentium 4 designers to favor a deeper pipeline rather than wider issue, is at least a reasonable design choice.
Throughout this chapter, our discussion has focused on exploiting parallelism in programs by finding and using the parallelism among instructions within the program. Although this approach has the great advantage that it is reasonably transparent to the programmer, as we have seen ILP can be quite limited or hard to exploit in some applications. Furthermore, there may be significant parallelism occurring naturally at a higher level in the application that cannot be exploited with the approaches discussed in this chapter. For example, an online transaction processing system has natural parallelism among the multiple queries and updates that are presented by requests. These queries and updates can be processed mostly in parallel, since they are largely independent of one another. Similarly, embedded applications often have natural high-level parallelism. For example, a processor in a network router can exploit parallelism among independent packets.

This higher level parallelism is called *thread level parallelism* because it is logically structured as separate threads of execution. A *thread* is a separate process with its own instructions and data. A thread may represent a process that is part of a parallel program consisting of multiple processes, or it may represent an independent program on its own. Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute. Unlike instruction level parallelism, which exploits implicit parallel operations within a loop or straight-line code segment, thread level parallelism is explicitly represented by the use of multiple threads of execution that are inherently parallel.

Thread level parallelism is important alternative to instruction level parallelism primarily because it could be more cost-effective to exploit than instruction level parallelism. There are many important applications where thread level parallelism occurs naturally, as it does in many server applications. In other cases, the software is being written from scratch and expressing the inherent parallelism is easy, as is true in some embedded applications. Chapter 6 explores multiprocessors and the support they provide for thread level parallelism.

The investment required to program applications to expose thread-level parallelism, makes it costly to switch the large established base of software to multiprocessors. This is especially true for desktop applications, where the natural parallelism that is present in many server environments, is harder to find. Thus, despite the potentially greater efficiency of exploiting thread-level parallelism, it is likely that ILP-based approaches will continue to be the primary focus for desktop-oriented processors.
3.12 Crosscutting Issues: Using an ILP Datapath to Exploit TLP

Thread-level and instruction-level parallelism exploit two different kinds of parallel structure in a program. One natural question to ask is whether it is possible for a processor oriented at instruction level parallelism to exploit thread level parallelism.

The motivation for this question comes from the observation that a datapath designed to exploit higher amounts of ILP, will find that functional units are often idle because of either stalls or dependences in the code. Could the parallelism among threads to be used a source of independent instructions that might be used to keep the processor busy during stalls? Could this thread-level parallelism be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?

Multithreading, and a variant called simultaneous multithreading, take advantage of these insights by using thread level parallelism either as the primary form of parallelism exploitation—for example, on top of a simple pipelined processor—or as a method that works in conjunction with ILP mechanisms. In both cases, multiple threads are being executed with in single processor by duplicating the thread-specific state (program counter, registers, and so on.) and sharing the other processor resources by multiplexing them among the threads. Since multithreading is a method for exploiting thread level parallelism, we discuss it in more depth in Chapter 6.

3.13 Fallacies and Pitfalls

Our first fallacy is a two part one that indicates that simple rules do not hold and that the choice of benchmarks plays a major role.

Fallacies:

- Processors with lower CPIs will always be faster.
- Processors with faster clock rates will always be faster.

Although a lower CPI is certainly better, sophisticated pipelines typically have slower clock rates than processors with simple pipelines. In applications with limited ILP or where the parallelism cannot be exploited by the hardware resources, the faster clock rate often wins. But, when significant ILP exists, a processor that exploits lots of ILP may be better.

The IBM Power III processor is designed for high-performance FP and capable of sustaining four instructions per clock, including two FP and two load-store
instructions. It offers a 400 MHz clock rate in 2000, capable and achieves a SPEC CINT2000 peak rating of 249 and a SPEC CFP2000 peak rating of 344. The Pentium III has a comparably aggressive integer pipeline but has less aggressive FP units. An 800 MHz Pentium III in 2000 achieves a SPEC CINT 2000 peak rating of 344 and a SPEC CFP2000 peak rating of 237.

Thus, the faster clock rate of the Pentium III (800 MHz vs. 400 MHz) leads to an integer rating that is 1.38 times higher than the Power III, but the more aggressive FP pipeline of the Power III (and a better instruction set for floating point) leads to a lower CPI. If we assume comparable instruction counts, the Power III CPI must be almost 3x better than that of the Pentium III for the SPECFP 2000 benchmarks, leading to an overall performance advantage of 1.45.

Of course, this fallacy is nothing more than a restatement of a pitfall from Chapter 2 (see page XXX) about comparing processors using only one part of the performance equation.

Pitfall: Emphasizing an improvement in CPI by increasing issue rate while sacrificing clock rate can lead to lower performance.

The TI SuperSPARC design is a flexible multiple-issue processor capable of issuing up to three instructions per cycle. It had a 1994 clock rate of 60 MHz. The HP PA 7100 processor is a simple dual-issue processor (integer and FP combination) with a 99-MHz clock rate in 1994. The HP processor is faster on all the SPEC92 benchmarks except two of the integer benchmarks and one FP benchmark, as shown in Figure 3.59. On average, the two processors are close on integer, but the HP processor is about 1.5 times faster on the FP benchmarks. Of course, differences in compiler technology, detailed tradeoffs in the processor (including the cache size and memory organization), and the implementation technology, could all contribute to the performance differences.

The potential of multiple-issue techniques has caused many designers to focus on improving CPI while possibly not focusing adequately on the trade-off in cycle time incurred when implementing these sophisticated techniques. This inclination arises at least partially because it is easier with good simulation tools to evaluate the impact of enhancements that affect CPI than it is to evaluate the cycle time impact.

There are two factors that lead to this outcome. First, it is difficult to know the clock rate impact of an approach until the design is well underway, and then it may be too late to make large changes in the organization. Second, the design simulation tools available for determining and improving CPI are generally better than those available for determining and improving cycle time.

In understanding the complex interaction between cycle time and various organizational approaches, the experience of the designers seems to be one of the most valuable factors. With ever more complex designs, however, even the best designers find it hard to understand the complex tradeoffs between clock rate and other organizational decisions. At the end of Section 3.10, we will see the oppo-
site problem: how emphasizing a high clock rate, obtained through a deeper pipeline, can lead to degraded CPI and a lower performance gain than might be expected based solely on the higher clock rate.

Pitfall: Improving only one aspect of a multiple-issue processor and expecting overall performance improvement.

This pitfall is simply a restatement of Amdahl’s Law. A designer might simply look at a design, see a poor branch prediction mechanism and improve it, expecting to see significant performance improvements. The difficulty is that many factors limit the performance of multiple-issue machines, and improving one aspect of a processor often exposes some other aspect that previously did not limit performance.

We can see examples of this in the data on ILP. For example, looking just at the effect of branch prediction in Figure 3.39 on page 302, we can see that going from a standard two-bit predictor to a tournament predictor significantly improves the parallelism in espresso (from an issue rate of 7 to an issue rate of 12). If the processor provides only 32 registers for renaming, however, the amount of parallelism is limited to 5 issues per clock cycle, even with a branch prediction scheme better than either alternative.

Pitfalls: Sometimes bigger and dumber is better.
Advanced pipelines have focused on novel and increasingly sophisticated schemes for improving CPI. The 21264 uses a sophisticated tournament predictor with a total of 29 Kbits (see page 258), while the earlier 21164 uses a simple 2-bit predictor with 2K entries (or a total of 4 Kbits). For the SPEC95 benchmarks, the more sophisticated branch predictor of the 21264 outperforms the simpler 2-bit scheme on all but one benchmark. On average, for SPECInt95, the 21264 has 11.5 mispredictions per 1000 instructions committed while the 21164 has about 16.5 mispredictions.

Somewhat surprisingly, the simpler 2-bit scheme works better for the transaction processing workload than the sophisticated 21264 scheme (17 mispredictions vs. 19 per 1000 completed instructions)! How can a predictor with less than 1/7 the number of bits and a much simpler scheme actually work better? The answer lies in the structure of the workload. The transaction processing workload has a very large code size (more than an order of magnitude larger than any SPEC95 benchmark) with a large branch frequency. The ability of the 21164 predictor to hold twice as many branch predictions based on purely local behavior (2K vs. the 1K local predictor in the 21264) seems to provide a slight advantage.

This pitfall also reminds us that different applications can produce different behaviors. As processors become more sophisticated including specific microarchitectural features aimed at some particular program behavior, it is likely that different applications will see more divergent behavior.

The tremendous interest in multiple-issue organizations came about because of an interest in improving performance without affecting the standard uniprocessor programming model. Although taking advantage of ILP is conceptually simple, the design problems are amazingly complex in practice. It is extremely difficult to achieve the performance you might expect from a simple first-level analysis.

The trade-offs between increasing clock speed and decreasing CPI through multiple issue are extremely hard to quantify. In the 1995 edition of this book, we stated: “Although you might expect that it is possible to build an advanced multiple-issue processor with a high clock rate, a factor of 1.5 to 2 in clock rate has consistently separated the highest clock rate processors and the most sophisticated multiple-issue processors. It is simply too early to tell whether this difference is due to fundamental implementation trade-offs, or to the difficulty of dealing with the complexities in multiple-issue processors, or simply a lack of experience in implementing such processors.”

Given the availability of the Alpha 21264 at 800 MHz, the Pentium III at 1.1 GHz, the AMD Athlon at 1.3 GHz, and the Pentium 4 at 2 GHz, it is clear that the limitation was primarily our understanding of how to build such processors. It is
also likely that this the first generation of CAD tools used for more than two million logic transistors was a limitation.

One insight that was clear in 1995 and remains clear in 2000 is that the peak to sustained performance ratios for multiple-issue processors are often quite large and typically grow as the issue rate grows. Thus, increasing the clock rate by \(X\) is almost always a better choice than increasing the issue width by \(X\), though often the clock rate increase may rely largely on deeper pipelining, substantially narrowing the advantage. This insight probably played a role in motivating Intel to pursue a deeper pipeline for the Pentium 4, rather than trying to increase the issue width. Recall, however, the fundamental observation we made in Chapter 1 about the improvement in semiconductor technologies: the number of transistors available grows faster than the speed of the transistors. Thus, a strategy that focuses only on deeper pipelining may not be the best use of the technology in the long run.

Rather than embracing dramatic new approaches in microarchitecture, the last five years have focused on raising the clock rates of multiple issue machines and narrowing the gap between peak and sustained performance. The dynamically-scheduled, multiple-issue processors announced in the last two years (the Alpha 21264, the Pentium III and 4, and the AMD Athlon) have same basic structure and similar sustained issue rates (three to four instructions per clock) as the first dynamically-scheduled, multiple-issue processors announced in 1995! But, the clock rates are 4 to 8 times higher, the caches are 2 to 4 times bigger, there are 2 to 4 times as many renaming registers, and twice as many load/store units! The result is performance that is 6 to 10 times higher.

All the leading edge desktop and server processors are large, complex chips with more than 15 million transistors per processor. Notwithstanding, a simple two-way superscalar that issues FP instructions in parallel with integer instructions, or dual issues integer instructions (but not memory references) can probably be built with little impact on clock rate and with a tiny die size (in comparison to today’s processes). Such a processor should perform well with a higher sustained to peak ratio than the high-end wide-issue processors and can be amazingly cost-effective. As a result, the high-end of the embedded space has recently moved to multiple-issue processors!

Whether approaches based primarily on faster clock rates, simpler hardware, and more static scheduling or approaches using more sophisticated hardware to achieve lower CPI will win out is difficult to say and may depend on the benchmarks.

**Practical Limitations on Exploiting More ILP**

Independent of the method used to exploit ILP, there are potential limitations that arise from employing more transistors. When the number of transistors employed is increased, the clock period is often determined by wire delays encountered both in distributing the clock and in the communication path of critical signals,
such as those that signal exceptions. These delays make it more difficult to em-
ploy increased numbers of transistors to exploit more ILP, while also increasing
the clock rate. These problems are sometimes overcome by adding additional
stages, which are reserved just for communicating signals across longer wires.
The Pentium 4 does this. These increased clock stages, however, can lead to more
stalls and a higher CPI, since they increase pipeline latency. We saw exactly this
phenomenon when comparing the Pentium 4 to the Pentium III.

Although the limitations explored in Section 3.8 act as significant barriers to
exploiting more ILP, it may be that more basic challenges would prevent the effi-
cient exploitation of additional ILP, even if it could be uncovered. For example,
doubling the issue rates above the current rates of four instructions per clock will
probably require a processor to sustain three or four memory accesses per cycle
and probably resolve two or three branches per cycle. In addition, supplying eight
instructions per cycle will probably require fetching sixteen, speculating through
multiple branches, and accessing roughly twenty registers per cycle. None of this
is impossible, but whether it can be done while simultaneously maintaining clock
rates exceeding 2 GHz is an open question and will surely be a significant chal-
lenge for any design team!

Equal in importance to the CPI versus clock rate trade-off, are realistic limita-
tions on power. Recall that dynamic power is proportional to the product of the
number of switching transistors and the switching rate. A microprocessor trying
to achieve both a low CPI and a high CR fights both of these factors. Achieving
an improved CPI means more instructions in flight and more transistors switch-
ing every clock cycle.

Two factors make it likely that the switching transistor count grows faster than
performance. The first is the gap between peak issue rates and sustained perfor-
mance, which continues to grow. Since the number of transistors switching is
likely to be proportional to the peak issue rate and the performance is propor-
tional to the sustained rate, the growing performance gap translates to increasing
transistors switches per unit of performance. Second, issuing multiple instruc-
tions incurs some overhead in logic that grows as the issue rate grows. This logic
is responsible for instruction issue analysis, including dependence checking, reg-
ister renaming, and similar functions. The combined result is that, without volt-
age reductions to decrease power, lower CPIs are likely to lead to lower ratios of
performance per watt.

A similar conundrum applies to attempts to increase clock rate. Of course, in-
creasing the clock rate will increase transistor switching frequency and directly
increase power consumption. As we saw in the Pentium 4 discussion, a deeper
pipeline structure can be used to achieve a clock rate increase that exceeds what
could be obtained just from improvements in transistor speed. Deeper pipelines,
however, incur additional power penalties, resulting from several sources. The
most important of these is the simple observation that a deeper pipeline means
more operations are in flight every clock cycle, which means more transistors are
switching, which means more power!
What is key to understand is the extent to which this potential growth in power caused by an increase in both the switching frequency and number of transistors switching is offset by a reduction in the operating voltage. Although these relationships is complex to understand, we can look at the results empirically and draw some conclusions.

The Pentium III and Pentium 4 provide an opportunity to examine this issue. As discussed on page 324, the Pentium 4 has a much deeper pipeline and can exploit more ILP than the Pentium III, although its basic peak issue rate is the same. The operating voltage of the Pentium 4 at 1.7 GHz is slightly higher than a 1 GHz Pentium III: 1.75V versus 1.70V. The power difference, however, is much larger: the 1.7 GHz Pentium 4 consumes 64 W typical, while the 1 GHz Pentium III consumes only 30 W by comparison. Figure 3.60 shows the effective performance of a 1.7 GHz Pentium 4 per watt relative to the performance per watt of a 1 GHz Pentium III using the same benchmarks presented in Figure 1.28 on page 60. Clearly, while the Pentium 4 is faster, its higher clock rate, deeper pipeline and higher sustained execution rate, make it significantly less power efficient. Whether the decreased power efficiency between the Pentium III and Pentium 4 are deep issues and unlikely to be overcome, or to whether they are artifacts of the two implementations is a key question that will probably be settled in future implementations. What is clear is that neither deeper pipelines nor wider issue rates can circumvent the need to consume more power to improve performance.

FIGURE 3.60 The relative performance per Watt of the Pentium 4 is 15% to 40% less than the Pentium III on these five sets of benchmarks.
More generally, the question of how best to exploit parallelism remains open. Clearly ILP will continue to play a big role because of its smaller impact on programmers and applications when compared to an explicitly parallel model using multiple threads and parallel processors. What sort of parallelism computer architects will employ as they try to achieve higher performance levels, and what type of parallelism programmers will accept are hard to predict. Likewise, it is unclear whether vectors will play a larger role in processors designed for multimedia and DSP applications, or whether such processors will rely on limited SIMD and ILP approaches. We will return to these questions in the next chapter as well as in Chapter 6.

3.15 Historical Perspective and References

This section describes some of the major advances in dynamically scheduled pipelines and ends with some of the recent literature on multiple-issue processors. Ideas such as data flow computation derived from observations that programs were limited by data dependence. The history of basic pipelining and the CDC 6600, the first dynamically scheduled processor, are contained in Appendix A.

The IBM 360 Model 91: A Landmark Computer

The IBM 360/91 introduced many new concepts, including tagging of data, register renaming, dynamic detection of memory hazards, and generalized forwarding. Tomasulo’s algorithm is described in his 1967 paper. Anderson, Sparacio, and Tomasulo [1967] describe other aspects of the processor, including the use of branch prediction. Many of the ideas in the 360/91 faded from use for nearly 25 years before being broadly resurrected in the 1990s. Unfortunately, the 360/91 was not successful and only a handful were sold. The complexity of the design made it late to the market and allowed the Model 85, which was the first IBM processor with a cache, to outperform the 91.

Branch Prediction Schemes

The two-bit dynamic hardware branch prediction scheme was described by J. E. Smith [1981]. Ditzel and McLellan [1987] describe a novel branch-target buffer for CRISP, which implements branch folding. McFarling and Hennessy [1986] did a quantitative comparison of a variety of compile-time and runtime branch prediction schemes. The correlating predictor we examine was described by Pan, So, and Rameh [1992]. Yeh and Patt [1992, 1993] generalized the correlation idea and described multilevel predictors that use branch histories for each branch, similar to the local history predictor used in the 21264. McFarling’s tournament prediction scheme, which he refers to a combined predictor, is described in his...
1993 technical report. There are a variety of more recent papers on branch prediction based on variations in the multilevel and correlating predictor ideas. Kaeli and Emma [1991] describe return address prediction.

The Development of Multiple-Issue Processors

The concept of multiple-issue designs has been around for a while, though much of the work in the 1970s focused on statically scheduled approaches, which we discuss in the next chapter. IBM did pioneering work on multiple issue. In the 1960s, a project called ACS was underway in California. It included multiple-issue concepts, a proposal for dynamic scheduling (although with a simpler mechanism than Tomasulo’s scheme, which used back-up registers), and fetching down both branch paths. The project originally started as a new architecture to follow Stretch and surpass the CDC 6600/6800. ACS started in New York but was moved to California, later changed to be S/360 compatible, and eventually canceled. John Cocke was one of the intellectual forces behind the team that included a number of IBM veterans and younger contributors many of whom went on to other important roles in IBM and elsewhere: Jack Bertram, Ed Sussenguth, Gene Amdahl, Herb Schorr, Fran Allen, Lynn Conway, and Phil Dauber among others. While the compiler team published many of their ideas and had great influence outside IBM, the architecture ideas were not widely disseminated at that time. The most complete accessible documentation of this important project is at: http://www.cs.clemson.edu/~mark/acs.html, which includes interviews with the ACS veterans and pointers to other sources. Sussenguth [1999] is a good overview of ACS.

More than 10 years after ACS was cancelled, John Cocke made a new proposal for a superscalar processor that dynamically made issue decisions; he described the key ideas in several talks in the mid 1980s and coined the name superscalar. He called the design America; it is described by Agerwala and Cocke [1987]. The IBM Power-1 architecture (the RS/6000 line) is based on these ideas (see Bakoglu et al. [1989]).

J. E. Smith [1984] and his colleagues at Wisconsin proposed the decoupled approach that included multiple issue with limited dynamic pipeline scheduling. A key feature of this processor is the use of queues to maintain order among a class of instructions (such as memory references) while allowing it to slip behind or ahead of another class of instructions. The Astronautics ZS-1 described by Smith et al. [1987] embodies this approach with queues to connect the load-store unit and the operation units. The Power-2 design uses queues in a similar fashion. J. E. Smith [1989] also describes the advantages of dynamic scheduling and compares that approach to static scheduling.

The concept of speculation has its roots in the original 360/91, which performed a very limited form of speculation. The approach used in recent processors combines the dynamic scheduling techniques of the 360/91 with a buffer to allow in-order commit. J. E. Smith and Pleszkun [1988] explored the use of buff-
er to maintain precise interrupts and described the concept of a reorder buffer. Sohi [1990] describes adding renaming and dynamic scheduling, making it possible to use the mechanism for speculation. Patt and his colleagues were early proponents of aggressive reordering and speculation. They focused on checkpoint and restart mechanisms and pioneered an approach called HPSm, which is also an extension of Tomasulo’s algorithm [Hwu and Patt 1986].

The use of speculation as a technique in multiple-issue processors was evaluated by Smith, Johnson, and Horowitz [1989] using the reorder buffer technique; their goal was to study available ILP in nonscientific code using speculation and multiple issue. In a subsequent book, M. Johnson [1990] describes the design of a speculative superscalar processor. Johnson later led the AMD K-5 design, one of the first speculative superscalars.

**Studies of ILP and Ideas to Increase ILP**

A series of early papers, including Tjaden and Flynn [1970] and Riseman and Foster [1972], concluded that only small amounts of parallelism could be available at the instruction level without investing an enormous amount of hardware. These papers dampened the appeal of multiple instruction issue for more than ten years. Nicolau and Fisher [1984] published a paper based on their work with trace scheduling and asserted the presence of large amounts of potential ILP in scientific programs.

Since then there have been many studies of the available ILP. Such studies have been criticized since they presume some level of both hardware support and compiler technology. Nonetheless, the studies are useful to set expectations as well as to understand the sources of the limitations. Wall has participated in several such studies, including Jouppi and Wall [1989], Wall [1991], and Wall [1993]. Although the early studies were criticized as being conservative (e.g., they didn’t include speculation), the last study is by far the most ambitious study of ILP to date and the basis for the data in section 3.10. Sohi and Vajapeyam [1989] give measurements of available parallelism for wide-instruction-word processors. Smith, Johnson, and Horowitz [1989] also used a speculative superscalar processor to study ILP limits. At the time of their study, they anticipated that the processor they specified was an upper bound on reasonable designs. Recent and upcoming processors, however, are likely to be at least as ambitious as their processor.

Lam and Wilson [1992] looked at the limitations imposed by speculation and shown that additional gains are possible by allowing processors to speculate in multiple directions, which requires more than one PC. (Such schemes cannot exceed what perfect speculation accomplishes, but they help close the gap between realistic prediction schemes and perfect prediction.) Wall’s 1993 study includes a limited evaluation of this approach (up to 8 branches are explored).
Going Beyond the Data Flow Limit

One other approach that has been explored in the literature is the use of value prediction. Value prediction can allow speculation based on data values. There have been a number of studies of the use of value prediction. Lipasti and Shen published two papers in 1996 evaluating the concept of value prediction and its potential impact on ILP exploitation. Sodani and Sohi [1997] approaches the same problem from the viewpoint of reusing the values produced by instructions. Moshovos, Breach, Vijaykumar and Sohi [1997] show that by deciding when to speculate on values, by tracking whether such speculation has been accurate in the past, is important to achieving performance gains with value speculation. Moshovos and Sohi [1997] and Chrysos and Emer [1998] focus on predicting memory dependences and using this information to eliminate the dependence through memory. Gonzalez and Gozalez [1998], Babbay and Mendelson [1998], and Calder, Reinman and Tullsen [1999] are more recent studies of the use of value prediction. This area is currently highly active with new results being published in every conference.

Recent Advanced Microprocessors

The years 1994–95 saw the announcement of wide superscalar processors (3 or more issues per clock) by every major processor vendor: Intel Pentium Pro and Pentium II (these processors share the same core pipeline architecture, described by Cowell and Steck [1995]), AMD K5, K6, and Althon, Sun UltraSPARC (see Lauterbach and Horel [1999]), Alpha 21164 (see Edmonston et. al [1995]) and 21264 (see Kessler [2000]), MIPS R10000 and R12000 (see Yeager [1996]), PowerPC 603, 604, 620 (see Diep, Nelson, and Shen [1995]), and HP 8000 (Kumar [1997]). The latter part of the decade (1996-2000), saw second generations of much of these processors (Pentium III, AMD Athlon, Alpha 21264, among others). The second generation, although similar in issue rate, could sustain a lower CPI, provided much higher clock rates, all included dynamic scheduling, and almost universally supported speculation. In practice, many factors, including the implementation technology, the memory hierarchy, the skill of the designers, and the type of applications benchmarked, all play a role in determining which approach is best. Figure 3.61 shows the most interesting processors of the past five years, their characteristics.
### FIGURE 3.61 Recent high-performance processors and their characteristics

The window size column shows the size of the buffer available for instructions, and, hence, the maximum number of instructions in flight. Both the Pentium III and the Athlon schedule microoperations and the window is the maximum number of microoperations in execution. The IBM, HP, and UltraSPARC processors support dynamic issue, but not speculation. To read more about these processors the following references are useful: *IBM Journal of Research and Development* (contains issues on Power and PowerPC designs), the *Digital Technical Journal* (contains issues on various Alpha processors), and *Proceedings of the Hot Chips Symposium* (annual meeting at Stanford, which reviews the newest microprocessors), the International Solid State Circuits Conference, and the annual Microprocessor Forum meetings, and the annual International Symposium on Computer Architecture. Much of this data in this table came from Microprocessor Report online April 30, 2001.

<table>
<thead>
<tr>
<th>Processor</th>
<th>System</th>
<th>Max. current (MHz)</th>
<th>Power (W)</th>
<th>Transistors (M)</th>
<th>Window size</th>
<th>Rename registers (int/FP)</th>
<th>Issue rate: Maximum/Memory / Integer / FP / Branch</th>
<th>Branch Predict Buffer</th>
<th>Pipe-stages (int/ load)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS R14000</td>
<td>2000</td>
<td>400</td>
<td>25</td>
<td>7</td>
<td>48</td>
<td>32/32</td>
<td>4/1/2/2/1</td>
<td>2K x 2</td>
<td>6</td>
</tr>
<tr>
<td>Ultra SPARC III</td>
<td>2001</td>
<td>900</td>
<td>65</td>
<td>29</td>
<td>N.A.</td>
<td>None</td>
<td>4/1/4/3/1</td>
<td>16K x 2</td>
<td>14/15</td>
</tr>
<tr>
<td>Pentium III</td>
<td>2000</td>
<td>1000</td>
<td>30</td>
<td>24</td>
<td>40</td>
<td>Total: 40</td>
<td>3/2/2/1/1</td>
<td>512 entries</td>
<td>12/14</td>
</tr>
<tr>
<td>Pentium 4</td>
<td>2001</td>
<td>1700</td>
<td>64</td>
<td>42</td>
<td>126</td>
<td>Total:128</td>
<td>3/2/3/2/1</td>
<td>4K x 2</td>
<td>22/24</td>
</tr>
<tr>
<td>HP PA 8600</td>
<td>2001</td>
<td>552</td>
<td>60</td>
<td>130</td>
<td>56</td>
<td>Total: 56</td>
<td>4/2/2/2/1</td>
<td>2K x 2</td>
<td>7/9</td>
</tr>
<tr>
<td>Alpha 21264B</td>
<td>2001</td>
<td>833</td>
<td>75</td>
<td>15</td>
<td>80</td>
<td>41/41</td>
<td>4/2/4/2/1</td>
<td>multilevel</td>
<td>7/9</td>
</tr>
<tr>
<td>Power PC 7400 (G4)</td>
<td>2000</td>
<td>450</td>
<td>5</td>
<td>7</td>
<td>5</td>
<td>6/6</td>
<td>3/1/2/1/1</td>
<td>512 x 2</td>
<td>4/5</td>
</tr>
<tr>
<td>AMD Athlon</td>
<td>2001</td>
<td>1330</td>
<td>76</td>
<td>37</td>
<td>72</td>
<td>36/36</td>
<td>3/2/3/3/1</td>
<td>4K x 9</td>
<td>9/11</td>
</tr>
<tr>
<td>IBM Power 3-II</td>
<td>2000</td>
<td>450</td>
<td>36</td>
<td>23</td>
<td>32</td>
<td>16/24</td>
<td>4/2/2/2/2</td>
<td>2K x 2</td>
<td>7/8</td>
</tr>
</tbody>
</table>

### References


BABBAY F. AND A. MENDELSON [1998]. “Using Value Prediction to Increase the Power of Speculative Execution Hardware.” *ACM Transactions on Computer Systems*, vol. 16, No. 3 (August), pages


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


THORNTON, J. E. [1970]. Design of a Computer, the Control Data 6600, Scott, Foresman, Glenview,
3.15 Historical Perspective and References

III.


EXERCISES

3.1 Exercise from Dave (not fully thought out, but a good direction): Given a table like that in Figures 3.25 on page 275 or 3.26 on page 276 and some of the following deduce the rest of the following:

a. the original code

b. the number of functional units

c. the number of instructions issued per clock

d. the functional units

3.2 [10] <3.1> For the following code fragment, list the control dependences. For each control dependence, tell whether the statement can be scheduled before the if statement based on the data references. Assume that all data references are shown, that all values are defined before use, and that only b and c are used again after this segment. You may ignore any possible exceptions.

```c
if (a>c) {
    d = d + 5;
    a = b + d + e;
}
else {
    e = e + 2;
    f = f + 2;
```
A good exercise but requires describing how scoreboards work. There are a number of problems based on scoreboards, which may be salvagable by one of the following: introducing scoreboards (maybe not worth it), removing part of the renaming capability (WAW or WAR) and asking about the result, recasting the problem to ask how Tomasulo avoids the problem.

3.3 [20] <3.2> It is critical that the scoreboard be able to distinguish RAW and WAR hazards, since a WAR hazard requires stalling the instruction doing the writing until the instruction reading an operand initiates execution, but a RAW hazard requires delaying the reading instruction until the writing instruction finishes—just the opposite. For example, consider the sequence:

\[
\begin{align*}
\text{MUL.D } & \ F0, F6, F4 \\
\text{SUB.D } & \ F8, F0, F2 \\
\text{ADD.D } & \ F2, F10, F2
\end{align*}
\]

The SUB.D depends on the MUL.D (a RAW hazard) and thus the MUL.D must be allowed to complete before the SUB.D; if the MUL.D were stalled for the SUB.D due to the inability to distinguish between RAW and WAR hazards, the processor will deadlock. This sequence contains a WAR hazard between the ADD.D and the SUB.D, and the ADD.D cannot be allowed to complete until the SUB.D begins execution. The difficulty lies in distinguishing the RAW hazard between MUL.D and SUB.D, and the WAR hazard between the SUB.D and ADD.D.

Describe how the scoreboard for a processor with two multiply units and two add units avoids this problem and show the scoreboard values for the above sequence assuming the ADD.D is the only instruction that has completed execution (though it has not written its result). (Hint: Think about how WAW hazards are prevented and what this implies about active instruction sequences.)

A good exercise but requires reworking (e.g., show how even with 1 issue/clock, a single cdb can be problem) to save it?

3.4 [12] <3.2> A shortcoming of the scoreboard approach occurs when multiple functional units that share input buses are waiting for a single result. The units cannot start simultaneously, but must serialize. This property is not true in Tomasulo’s algorithm. Give a code sequence that uses no more than 10 instructions and shows this problem. Assume the hardware configuration from Figure 4.3, for the scoreboard, and Figure 3.2, for Tomasulo’s scheme. Use the FP latencies from Figure 4.2 (page 224). Indicate where the Tomasulo approach can continue, but the scoreboard approach must stall.

A good exercise but requires reworking (e.g., show how even with 1 issue/clock, a single cdb can be problem) to save it?

3.5 [15] <3.2> Tomasulo’s algorithm also has a disadvantage versus the scoreboard: only one result can complete per clock, due to the CDB. Use the hardware configuration from Figures 4.3 and 3.2 and the FP latencies from Figure 4.2 (page 224). Find a code sequence
of no more than 10 instructions where the scoreboard does not stall, but Tomasulo’s algorithm must due to CDB contention. Indicate where this occurs in your sequence.

**Maybe also try a version of this with multiple issue?**

3.6 [45] <3.2> One benefit of a dynamically scheduled processor is its ability to tolerate changes in latency or issue capability without requiring recompilation. This capability was a primary motivation behind the 360/91 implementation. The purpose of this programming assignment is to evaluate this effect. Implement a version of Tomasulo’s algorithm for MIPS to issue one instruction per clock; your implementation should also be capable of inorder issue. Assume fully pipelined functional units and the latencies shown in Figure 3.62.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>7</td>
</tr>
<tr>
<td>Branch</td>
<td>9</td>
</tr>
<tr>
<td>Load-store</td>
<td>11</td>
</tr>
<tr>
<td>FP add</td>
<td>13</td>
</tr>
<tr>
<td>FP mul</td>
<td>15</td>
</tr>
<tr>
<td>FP divide</td>
<td>17</td>
</tr>
</tbody>
</table>

**FIGURE 3.62  Latencies for functional units.**

A one-cycle latency means that the unit and the result are available for the next instruction. Assume the processor takes a one-cycle stall for branches, in addition to any data-dependent stalls shown in the above table. Choose 5–10 small FP benchmarks (with loops) to run; compare the performance with and without dynamic scheduling. Try scheduling the loops by hand and see how close you can get with the statically scheduled processor to the dynamically scheduled results.

Change the processor to the configuration shown in Figure 3.63.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>19</td>
</tr>
<tr>
<td>Branch</td>
<td>21</td>
</tr>
<tr>
<td>Load-store</td>
<td>23</td>
</tr>
<tr>
<td>FP add</td>
<td>25</td>
</tr>
<tr>
<td>FP mul</td>
<td>27</td>
</tr>
<tr>
<td>FP divide</td>
<td>29</td>
</tr>
</tbody>
</table>

**FIGURE 3.63  Latencies for functional units, configuration 2.**

Rerun the loops and compare the performance of the dynamically scheduled processor and
the statically scheduled processor.

3.7 [15] Suppose we have a deeply pipelined processor, for which we implement a branch-target buffer for the conditional branches only. Assume that the misprediction penalty is always 4 cycles and the buffer miss penalty is always 3 cycles. Assume 90% hit rate and 90% accuracy, and 15% branch frequency. How much faster is the processor with the branch-target buffer versus a processor that has a fixed 2-cycle branch penalty? Assume a base CPI without branch stalls of 1.

3.8 [10] Determine the improvement from branch folding for unconditional branches. Assume a 90% hit rate, a base CPI without unconditional branch stalls of 1, and an unconditional branch frequency of 5%. How much improvement is gained by this enhancement versus a processor whose effective CPI is 1.1?

3.9 [30] Implement a simulator to evaluate the performance of a branch-prediction buffer that does not store branches that are predicted as untaken. Consider the following prediction schemes: a one-bit predictor storing only predicted taken branches, a two-bit predictor storing all the branches, a scheme with a target buffer that stores only predicted taken branches and a two-bit prediction buffer. Explore different sizes for the buffers keeping the total number of bits (assuming 32-bit addresses) the same for all schemes. Determine what the branch penalties are, using Figure 3.21 as a guideline. How do the different schemes compare both in prediction accuracy and in branch cost?

3.10 [30] Implement a simulator to evaluate various branch prediction schemes. You can use the instruction portion of a set of cache traces to simulate the branch-prediction buffer. Pick a set of table sizes (e.g., 1K bits, 2K bits, 8K bits, and 16K bits). Determine the performance of both (0,2) and (2,2) predictors for the various table sizes. Also compare the performance of the degenerate predictor that uses no branch address information for these table sizes. Determine how large the table must be for the degenerate predictor to perform as well as a (0,2) predictor with 256 entries.

*this is an interesting exercise to do in several forms: tomsulo, multiple issue with tomasulo and even speculation. Needs some regorking. may want to ask them to create tables like those in the text (Figures 3.25 on page 275 and 3.26 on page 276)*

3.11 [20/22/22/22/22/25/25/25/20/22/22] In this Exercise, we will look at how a common vector loop runs on a variety of pipelined versions of MIPS. The loop is the so-called SAXPY loop (discussed extensively in Appendix B) and the central operation in Gaussian elimination. The loop implements the vector operation \( Y = a \times X + Y \) for a vector of length 100. Here is the MIPS code for the loop:

```mips
foo:  L.D   F2,0(R1) ; load X(i)
      MUL.D  F4,F2,F0 ; multiply a*X(i)
      L.D   F6,0(R2) ; load Y(i)
      ADD.D  F6,F4,F6 ; add a*X(i) + Y(i)
      S.D   F6,0(R2) ; store Y(i)
      DADDUI R1,R1,#8 ; increment X index
      DADDUI R2,R2,#8 ; increment Y index
      DSGTUI R3,R1,done ; test if done
      BEQZ R3,foo ; loop if not done
```

For (a)–(e), assume that the integer operations issue and complete in one clock cycle (in-
including loads) and that their results are fully bypassed. Ignore the branch delay. You will use the FP latencies shown in Figure 4.2 (page 224). Assume that the FP unit is fully pipelined.

a. [20] <3.1> For this problem use the standard single-issue MIPS pipeline with the pipeline latencies from Figure 4.2. Show the number of stall cycles for each instruction and what clock cycle each instruction begins execution (i.e., enters its first EX cycle) on the first iteration of the loop. How many clock cycles does each loop iteration take?

b. [22] <3.2> Use the MIPS code for SAXPY above and a fully pipelined FPU with the latencies of Figure 4.2. Assume Tomasulo’s algorithm for the hardware with one integer unit taking one execution cycle (a latency of 0 cycles to use) for all integer operations. Show the state of the reservation stations and register-status tables (as in Figure 3.3) when the SGTI writes its result on the CDB. Do not include the branch.

c. [22] <3.2> Using the MIPS code for SAXPY above, assume a scoreboard with the FP functional units described in Figure 4.3, plus one integer functional unit (also used for load-store). Assume the latencies shown in Figure 3.64. Show the state of the scoreboard (as in Figure 4.4) when the branch issues for the second time. Assume the branch was correctly predicted taken and took one cycle. How many clock cycles does each loop iteration take? You may ignore any register port/bus conflicts.

d. [25] <3.2> Use the MIPS code for SAXPY above. Assume Tomasulo’s algorithm for the hardware using one fully pipelined FP unit and one integer unit. Assume the latencies shown in Figure 3.64.

Show the state of the reservation stations and register status tables (as in Figure 3.3) when the branch is executed for the second time. Assume the branch was correctly predicted as taken. How many clock cycles does each loop iteration take?

e. [25] <3.1.3.6> Assume a superscalar architecture with Tomasulo’s algorithm for scheduling that can issue any two independent operations in a clock cycle (including two integer operations). Unwind the MIPS code for SAXPY to make four copies of the body and schedule it assuming the FP latencies of Figure 4.2. Assume one fully pipelined copy of each functional unit (e.g., FP adder, FP multiplier) and two integer units.

<table>
<thead>
<tr>
<th>Instruction producing result</th>
<th>Instruction using result</th>
<th>Latency in clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP multiply</td>
<td>FP ALU op</td>
<td>6</td>
</tr>
<tr>
<td>FP add</td>
<td>FP ALU op</td>
<td>4</td>
</tr>
<tr>
<td>FP multiply</td>
<td>FP store</td>
<td>5</td>
</tr>
<tr>
<td>FP add</td>
<td>FP store</td>
<td>3</td>
</tr>
<tr>
<td>Integer operation (including load)</td>
<td>Any</td>
<td>0</td>
</tr>
</tbody>
</table>

FIGURE 3.64   Pipeline latencies where latency is number of cycles between producing and consuming instruction.
functional units with latency to use of 0. How many clock cycles will each iteration on the original code take? When unwinding, you should optimize the code as in section 3.1. What is the speedup versus the original code?

f. [25] <3.6> In a superpipelined processor, rather than have multiple functional units, we would fully pipeline all the units. Suppose we designed a superpipelined MIPS that had twice the clock rate of our standard MIPS pipeline and could issue any two unrelated instructions in the same time that the normal MIPS pipeline issued one operation. If the second instruction is dependent on the first, only the first will issue. Unroll the MIPS SAXPY code to make four copies of the loop body and schedule it for this superpipelined processor, assuming the FP latencies of Figure 3.64. Also assume the load to use latency is 1 cycle, but other integer unit latencies are 0 cycles. How many clock cycles does each loop iteration take? Remember that these clock cycles are half as long as those on a standard MIPS pipeline or a superscalar MIPS.

g. [22] <3.2,3.5> Using the MIPS code for SAXPY above, assume a speculative processor with the functional unit organization used in section 3.5 and separate functional units for comparison, for branches, for effective address calculation, and for ALU operations. Assume the latencies shown in Figure 3.64. Show the state of the processor (as in Figure 3.30) when the branch issues for the second time. Assume the branch was correctly predicted taken and took one cycle. How many clock cycles does each loop iteration take?

h. [22] <3.2,3.5> Using the MIPS code for SAXPY above, assume a speculative processor like Figure 3.29 that can issue one load-store, one integer operation, and one FP operation each cycle. Assume the latencies in clock cycles of Figure 3.64. Show the state of the processor (as in Figure 3.30) when the branch issues for the second time. Assume the branch was correctly predicted taken and took one cycle. How many clock cycles does each loop iteration take?

3.12 [15/15] <3.5> Consider our speculative processor from section 3.5. Since the reorder buffer contains a value field, you might think that the value field of the reservation stations could be eliminated.

a. [15] <3.5> Show an example where this is the case and an example where the value field of the reservation stations is still needed. Use the speculative processor shown in Figure 3.29. Show MIPS code for both examples. How many value fields are needed in each reservation station?

b. [15] <3.5> Find a modification to the rules for instruction commit that allows elimination of the value fields in the reservation station. What are the negative side effects of such a change?

3.13 [20] <3.5> Our implementation of speculation uses a reorder buffer and introduces the concept of instruction commit, delaying commit and the irrevocable updating of the registers until we know an instruction will complete. There are two other possible implementation techniques, both originally developed as a method for preserving precise interrupts when issuing out of order. One idea introduces a future file that keeps future values of a register; this idea is similar to the reorder buffer. An alternative is to keep a history buffer that records values of registers that have been speculatively overwritten.
3.14 [30/30] <3.10> This exercise involves a programming assignment to evaluate what types of parallelism might be expected in more modest, and more realistic, processors than those studied in section 3.8. These studies can be done using traces available with this text or obtained from other tracing programs. For simplicity, assume perfect caches. For a more ambitious project, assume a real cache. To simplify the task, make the following assumptions:

- Assume perfect branch and jump prediction: hence you can use the trace as the input to the window, without having to consider branch effects—the trace is perfect.
- Assume there are 64 spare integer and 64 spare floating-point registers; this is easily implemented by stalling the issue of the processor whenever there are more live registers required.
- Assume a window size of 64 instructions (the same for alias detection). Use greedy scheduling of instructions in the window. That is, at any clock cycle, pick for execution the first \( n \) instructions in the window that meet the issue constraints.

a. [30] <3.10> Determine the effect of limited instruction issue by performing the following experiments:
   - Vary the issue count from 4–16 instructions per clock,
   - Assuming eight issues per clock: determine what the effect of restricting the processor to two memory references per clock is.

b. [30] <3.10> Determine the impact of latency in instructions. Assume the following latency models for a processor that issues up to 16 instructions per clock:
   - Model 1: All latencies are one clock.
   - Model 2: Load latency and branch latency are one clock; all FP latencies are two clocks.
   - Model 3: Load and branch latency is two clocks; all FP latencies are five clocks.

Remember that with limited issue and a greedy scheduler, the impact of latency effects will be greater.

3.15 [Discussion] <3.4,3.5> Dynamic instruction scheduling requires a considerable investment in hardware. In return, this capability allows the hardware to run programs that could not be run at full speed with only compile-time, static scheduling. What trade-offs should be taken into account in trying to decide between a dynamically and a statically scheduled implementation? What situations in either hardware technology or program characteristics are likely to favor one approach or the other? Most speculative schemes rely on dynamic scheduling; how does speculation affect the arguments in favor of dynamic scheduling?

3.16 [Discussion] <3.4> There is a subtle problem that must be considered when imple-
menting Tomasulo’s algorithm. It might be called the “two ships passing in the night problem.” What happens if an instruction is being passed to a reservation station during the same clock period as one of its operands is going onto the common data bus? Before an instruction is in a reservation station, the operands are fetched from the register file; but once it is in the station, the operands are always obtained from the CDB. Since the instruction and its operand tag are in transit to the reservation station, the tag cannot be matched against the tag on the CDB. So there is a possibility that the instruction will then sit in the reservation station forever waiting for its operand, which it just missed. How might this problem be solved? You might consider subdividing one of the steps in the algorithm into multiple parts. (This intriguing problem is courtesy of J. E. Smith.)

3.17 [Discussion] <3.6-3.5> Discuss the advantages and disadvantages of a superscalar implementation, a superpipelined implementation, and a VLIW approach in the context of MIPS. What levels of ILP favor each approach? What other concerns would you consider in choosing which type of processor to build? How does speculation affect the results?

Need some more exercises on speculation, newer branch predictors, and probably also multiple issue with Tomasulo and with speculation--maybe an integer loop?

Add something on multiple processors/chip