3

Instruction-Level Parallelism and Its Exploitation

"Who's first?"
"America."
"Who's second?"
"Sir, there is no second."

Dialog between two observers of the sailing race in 1851, later named "The America's Cup," which was the inspiration for John Cocke's naming of an IBM research processor as "America," the first superscalar processor, and a precursor to the PowerPC.

Thus, the IA-64 gambles that, in the future, power will not be the critical limitation, and massive resources…will not penalize clock speed, path length, or CPI factors. My view is clearly skeptical…

Marty Hopkins (2000), IBM Fellow and Early RISC pioneer commenting in 2000 on the new Intel Itanium, a joint development of Intel and HP. The Itanium used a static ILP approach (see Appendix H) and was a massive investment for Intel. It never accounted for more than 0.5% of Intel's microprocessor sales.

Computer Architecture. https://doi.org/10.1016/B978-0-12-811905-1.00003-1
© 2019 Elsevier Inc. All rights reserved.

This is achieved by pipelined execution of instructions

3.1 Instruction-Level Parallelism: Concepts and Challenges

It attempts to execute several instructions of the same program (process) in parallel

All processors since about 1985 have used pipelining to overlap the execution of instructions and improve performance. This potential overlap among instructions is called *instruction-level parallelism* (ILP), because the instructions can be evaluated in parallel. In this chapter and Appendix H, we look at a wide range of techniques for extending the basic pipelining concepts by increasing the amount of parallelism exploited among instructions.

This chapter is at a considerably more advanced level than the material on basic pipelining in Appendix C. If you are not thoroughly familiar with the ideas in Appendix C, 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 compiler and the processor to exploit parallelism. These sections introduce a large number of concepts, which we build on throughout this chapter and the next. While some of the more basic material in this chapter could be understood without all of the ideas in the first two sections, this basic material is important to later sections of this chapter.

There are two largely separable approaches to exploiting ILP: (1) an approach that relies on hardware to help discover and exploit the parallelism dynamically, and (2) an approach that relies on software technology to find parallelism statically at compile time. Processors using the dynamic, hardware-based approach, including all recent Intel and many ARM processors, dominate in the desktop and server markets. In the personal mobile device market, the same approaches are used in processors found in tablets and high-end cell phones. In the IOT space, where power and cost constraints dominate performance goals, designers exploit lower levels of instruction-level parallelism. Aggressive compiler-based approaches have been attempted numerous times beginning in the 1980s and most recently in the Intel Itanium series, introduced in 1999. Despite enormous efforts, such approaches have been successful only in domain-specific environments or in well-structured scientific applications with significant data-level parallelism.

battery life and cost of day-to-day devices matter a lot.

Expensive products are not affordable by common people

In the past few years, many of the techniques developed for one approach have been exploited within a design relying primarily on the other. This chapter introduces the basic concepts and both approaches. A discussion of the limitations on ILP approaches is included in this chapter, and it was such limitations that directly led to the movement toward multicore. Understanding the limitations remains important in balancing the use of ILP and thread-level parallelism.

interleaved execution of instructions of different programs on a single CPU

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.

The value of the CPI (cycles per instruction) for a pipelined processor is the sum of the base CPI and all contributions from stalls:

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

This has been explained in Appendix C

one, means apparent number of clock cycle for every instruction is 1. Actual number of clock cycles spent in execution of an instruction does not decrease due to pipelined execution.

Technique	Reduces	Section
Forwarding and bypassing	Potential data hazard stalls	C.2
Simple branch scheduling and prediction	Control hazard stalls	C.2
Basic compiler pipeline scheduling	Data hazard stalls	C.2, 3.2
Basic dynamic scheduling (scoreboarding)	Data hazard stalls from true dependences	C.7
Loop unrolling	Control hazard stalls	3.2
Advanced branch prediction	Control stalls	3.3
Dynamic scheduling with renaming	Stalls from data hazards, output dependences, and antidependences	3.4
Hardware speculation	Data hazard and control hazard stalls	3.6
Dynamic memory disambiguation	Data hazard stalls with memory	3.6
Issuing multiple instructions per cycle	Ideal CPI	3.7, 3.8
Compiler dependence analysis, software pipelining, trace scheduling	Ideal CPI, data hazard stalls	H.2, H.3
Hardware support for compiler speculation	Ideal CPI, data hazard stalls, branch hazard stalls	H.4, H.5

Figure 3.1 The major techniques examined in Appendix C, Chapter 3, and Appendix H are shown together with the component of the CPI equation that the technique affects.

As sson as a branch instruction appears, it marks the end of a basic block.

Branch instructions in executable code are due to decision statements (IF, Switch...Case, etc), loops (for, do, while, etc), and function calls in the high level language programs.

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 decrease the overall pipeline CPI or, alternatively, increase the IPC (instructions per clock). The preceding equation allows us to characterize various techniques by what component of the overall CPI a technique reduces. Figure 3.1 shows the techniques we examine in this chapter and in Appendix H, as well as the topics covered in the introductory material in Appendix C. In this chapter, we will see that the techniques we introduce to decrease the ideal pipeline CPI can increase the importance of dealing with hazards.

It is executing multiple instructions of a program in parallel.

What Is Instruction-Level Parallelism? → ILP

A basic block is defined for an executable code. It is a sequence of n adjacent machine instructions that does not have any branch except the last one.

The nth (last) instruction is a branch instruction that sends control either to 1st instruction or some instruction outside the range 1..n.

All the techniques in this chapter exploit parallelism among instructions. 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 RISC programs, the average dynamic branch frequency is often between 15% and 25%, meaning that between three and six instructions execute between a pair of branches. Because these instructions are likely to depend upon one another, the amount of overlap we can exploit within a basic block is likely to be less than the average basic block size. To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks.

Example:

```
Try     Add r1, r3, r9
        sub  r2. r1, r5
        bne  r2, r0, Try
```

This basic block contains only three instructions as the third instruction is a branch instruction.

The simplest and most common way to increase the ILP 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 that adds two 1000-element arrays and is completely parallel:

```
for (i=0; i<=999; i=i+1)
        x[i] = x[i] + y[i];
```

loop iterations are independent and hence can be carried out in parallel

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

We will examine a number of techniques for converting such loop-level parallelism into instruction-level parallelism. Basically, such techniques work by unrolling the loop either statically by the compiler (as in the next section) or dynamically by the hardware (as in Sections 3.5 and 3.6).

An important alternative method for exploiting loop-level parallelism is the use of SIMD in both vector processors and graphics processing units (GPUs), both of which are covered in Chapter 4. A SIMD instruction exploits data-level parallelism by operating on a small to moderate number of data items in parallel (typically two to eight). A vector instruction exploits data-level parallelism by operating on many data items in parallel using both parallel execution units and a deep pipeline. For example, the preceding code sequence, which in simple form requires seven instructions per iteration (two loads, an add, a store, two address updates, and a branch) for a total of 7000 instructions, might execute in one-quarter as many instructions in some SIMD architecture where four data items are processed per instruction. On some vector processors, this sequence might take only four instructions: 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.

Data Dependences 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 of arbitrary depth without causing any stalls, assuming the pipeline has sufficient resources (and thus no structural hazards exist). If two instructions are dependent, they are not parallel and must be executed in order, although 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:

RAW hazard

WAR and WAW hazards

branch hazards

- Instruction *i* produces a result that may be used by instruction *j*.
- 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. Note that a dependence within a single instruction (such as `add x1,x1,x1`) is not considered a dependence.

For example, consider the following RISC-V code sequence that increments a vector of values in memory (starting at `0(x1)` ending with the last element at `0(x2)`) by a scalar in register `f2`.

```
Loop:  fld     f0,0(x1)    //f0=array element
       fadd.d  f4,f0,f2    //add scalar in f2
       fsd     f4,0(x1)    //store result
       addi    x1,x1,−8    //decrement pointer 8 bytes
       bne     x1,x2,Loop  //branch x1≠x2
```

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

```
Loop:  fld     f0,0(x1)    //f0=array element
       fadd.d  f4,f0,f2    //add scalar in f2
       fsd     f4,0(x1)    //store result
```

Three RAW hazards

and integer data:

```
       addi    x1,x1,-8    //decrement pointer
                           //8 bytes (per DW)
       bne     x1,x2,Loop//branch x1ax2
```

In both of the preceding dependent sequences, as shown by the arrows, each instruction depends 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 must execute in order and 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. (See Appendix C for a brief description of data hazards, which we will define precisely in a few pages.) Executing the instructions simultaneously will cause a processor with pipeline interlocks (and a pipeline depth longer than the distance between the instructions in cycles) 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 because 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.

A data dependence conveys three things: (1) the possibility of a hazard, (2) the order in which results must be calculated, and (3) an upper bound on how much parallelism can possibly be exploited. Such limits are explored in a pitfall on page 262 and in Appendix H in more detail.

Because a data dependence can limit the amount of instruction-level parallelism we can exploit, a major focus of this chapter is overcoming these limitations. A dependence can be overcome in two different ways: (1) maintaining the dependence but avoiding a hazard, and (2) eliminating a dependence by transforming the code. Scheduling the code is the primary method used to avoid a hazard without altering a dependence, and such scheduling can be done both by the compiler and by the hardware.

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

Dependences that flow through memory locations are more difficult to detect because two addresses may refer to the same location but look different: For example, `100(x4)` and `20(x6)` may be identical memory addresses. In addition, the effective address of a load or store may change from one execution of the instruction to another (so that `20(x4)` and `20(x4)` may be different), further complicating the detection of a dependence.

In this chapter, we examine hardware for detecting data dependences that involve memory locations, but we will see that these techniques also have limitations. The compiler techniques for detecting such dependences are critical in uncovering loop-level parallelism.

Data dependence, name dependence, antidependence and control dependence are not different concepts than what are discussed in Appendix C as RAW, WAR and WAW type data hazards and control hazards.

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

WAR hazard

ordering must be preserved to ensure that *i* reads the correct value. In the example on page 171, there is an antidependence between `fsd` and `addi` on register `x1`.

WAW hazard

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, because there is no value being transmitted between the instructions. Because 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 exists whenever there is a name or data dependence between instructions, and they are close enough that the overlap during execution 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, which are informally described in Appendix C, 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* preceding *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*.
- 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.

- 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 (or name dependence). WAR hazards cannot occur in most static issue pipelines—even deeper pipelines or floating-point pipelines—because all reads are early (in ID in the pipeline in Appendix C) and all writes are late (in WB in the pipeline in Appendix C). 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 instruction *i* is executed in correct program order and only when it should be. Every instruction, except for those in the first basic block of the program, is control-dependent on some set of branches, and in general, these control dependences must be preserved to preserve program order. One of the simplest examples of a control dependence is the dependence of the statements in the "then" part of an if statement on the branch. For example, in the code segment

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

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

In general, two constraints are 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.

When processors preserve strict program order, they ensure that control dependences are also preserved. We may be willing to execute instructions that should not have been executed, however, thereby violating the control dependences, *if* we

can do so without affecting the correctness of the program. Thus 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 dependences—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:

```
      add  x2,x3,x4
      beq  x2,x0,L1
      ld   x1,0(x2)
L1:
```

In this case, it is easy to see that if we do not maintain the data dependence involving `x2`, 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 `ld`; it is only the control dependence. To allow us to reorder these instructions (and still preserve the data dependence), we want to just ignore the exception when the branch is taken. In Section 3.6, we will look at a hardware technique, *speculation*, which allows us to overcome this exception problem. Appendix H looks at software techniques for supporting speculation.

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 because they allow the source of data for a given instruction to come from many points. Put another way, it is insufficient 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:

```
      add  x1,x2,x3
      beq  x4,x0,L
      sub  x1,x5,x6
L:    ...
      or   x7,x1,x8
```

In this example, the value of `x1` 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 `add` and `sub` instructions, but preserving that 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 `x1` computed by the `sub` should be used by the `or`, and if the branch is taken, the value of `x1` computed by the `add` 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 `sub` 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 3.6.

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

```
        add  x1,x2,x3
        beq  x12,x0,skip
        sub  x4,x5,x6
        add  x5,x4,x9
skip:   or   x7,x8,x9
```

Suppose we knew that the register destination of the `sub` instruction (`x4`) was unused after the instruction labeled `skip`. (The property of whether a value will be used by an upcoming instruction is called *liveness*.) If `x4` were unused, then changing the value of `x4` just before the branch would not affect the data flow because `x4` would be *dead* (rather than live) in the code region after `skip`. Thus, if `x4` were dead and the existing `sub` instruction could not generate an exception (other than those from which the processor resumes the same process), we could move the `sub` instruction before the branch because the data flow could not be affected by this change.

If the branch is taken, the `sub` instruction will execute and will be useless, but it will not affect the program results. This type of code scheduling is also a form of speculation, often called software speculation, because 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 Appendix H. Normally, it will be clear when we say speculation or speculative whether the mechanism is a hardware or software mechanism; when it is not clear, it is best to say "hardware speculation" or "software speculation."

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, which we examine in Section 3.3.

3.2 Basic Compiler Techniques for Exposing ILP

This section examines the use of simple compiler technology to enhance a processor's ability to exploit ILP. These techniques are crucial for processors that use static issue or static scheduling. Armed with this compiler technology, we will shortly examine the design and performance of processors using static issuing. Appendix H will investigate more sophisticated compiler and associated hardware schemes designed to enable a processor to exploit more instruction-level parallelism.

Basic Pipeline Scheduling and Loop Unrolling

To keep a pipeline full, parallelism among instructions must be exploited by finding sequences of unrelated instructions that can be overlapped in the pipeline. To avoid a pipeline stall, the execution of a dependent instruction must be separated from the source instruction by a distance in clock cycles equal to the pipeline latency of that source instruction. A compiler's ability to perform this scheduling depends both on the amount of ILP available in the program and on the latencies of the functional units in the pipeline. Figure 3.2 shows the FP unit latencies we assume in this chapter, unless different latencies are explicitly stated. We assume the standard five-stage integer pipeline so that branches have a delay of one clock cycle. We assume that the functional units are fully pipelined or replicated (as many times as the pipeline depth) so that an operation of any type can be issued on every clock cycle and there are no structural hazards.

In this section, we look at how the compiler can increase the amount of available ILP by transforming loops. This example serves both to illustrate an important technique as well as to motivate the more powerful program transformations described in Appendix H. We will rely on the following code segment, which adds a scalar to a vector:

```
for (i=999; i>=0; i=i–1)
        x[i] = x[i] + s;
```

Each element of the array x[i] can be modified in parallel as each iteration of the for loop makes a computation independent of other iteration.

We can see that this loop is parallel by noticing that the body of each iteration is independent. We formalize this notion in Appendix H and describe how we can test whether loop iterations are independent at compile time. First, let's look at the performance of this loop, which shows how we can use the parallelism to improve its performance for a RISC-V pipeline with the preceding latencies.

The first step is to translate the preceding segment to RISC-V assembly language. In the following code segment, `x1` is initially the address of the element in the array with the highest address, and `f2` contains the scalar value *s*. Register `x2` is precomputed so that `Regs[x2]+8` is the address of the last element to operate on.

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Figure 3.2 Latencies of FP operations used in this chapter. The last column is the number of intervening clock cycles needed to avoid a stall. These numbers are similar to the average latencies we would see on an FP unit. The latency of a floating-point load to a store is 0 because the result of the load can be bypassed without stalling the store. We will continue to assume an integer load latency of 1 and an integer ALU operation latency of 0 (which includes ALU operation to branch).

load floating point data from memory to CPU register f0

it is memory location whose address is 0 (zero) + content of CPU register x1

The straightforward RISC-V code, not scheduled for the pipeline, looks like this:

```
Loop:  fld     f0,0(x1)      //f0=array element
       fadd.d  f4,f0,f2      //add scalar in f2
       fsd     f4,0(x1)      //store result
       addi    x1,x1,-8      //decrement pointer
                             //8 bytes (per DW)
       bne     x1,x2,Loop    //branch x1≠x2
```

store floating point data from CPU register f4 to memory location specified as 0(x1)

Let's start by seeing how well this loop will run when it is scheduled on a simple pipeline for RISC-V with the latencies in Figure 3.2.

Example Show how the loop would look on RISC-V, both scheduled and unscheduled, including any stalls or idle clock cycles. Schedule for delays from floating-point operations.

Answer Without any scheduling, the loop will execute as follows, taking nine cycles:

```
                                  Clock cycle issued
Loop:  fld     f0,0(x1)           1
       stall                      2
       fadd.d  f4,f0,f2           3
       stall                      4
       stall                      5
       fsd     f4,0(x1)           6
       addi    x1,x1,-8           7
       bne     x1,x2,Loop         8
```

These three statements increments the content of memory location (array element)

move addi before fadd.d so as to utilize the stalled cycle. This eliminates the first stall

x1 is decremented in order to point to next element of the array (stored in memory)

We can schedule the loop to obtain only two stalls and reduce the time to seven cycles:

```
Loop:  fld     f0,0(x1)
       addi    x1,x1,-8
       fadd.d  f4,f0,f2
       stall
       stall
       fsd     f4,8(x1)
       bne     x1,x2,Loop
```

array elements are incremented in reverse order, last element to first element, hence negative value to decrement pointer

The stalls after `fadd.d` are for use by the `fsd`, and repositioning the `addi` prevents the stall after the `fld`.

In the previous example, we complete one loop iteration and store back one array element every seven clock cycles, but the actual work of operating on the array element takes just three (the load, add, and store) of those seven clock cycles.

The remaining four clock cycles consist of loop overhead—the `addi` and `bne`—and two stalls. To eliminate these four clock cycles, we need to get more operations relative to the number of overhead instructions.

A simple scheme for increasing the number of instructions relative to the branch and overhead instructions is *loop unrolling*. Unrolling simply replicates the loop body multiple times, adjusting the loop termination code.

Loop unrolling can also be used to improve scheduling. Because it eliminates the branch, it allows instructions from different iterations to be scheduled together. In this case, we can eliminate the data use stalls by creating additional independent instructions within the loop body. If we simply replicated the instructions when we unrolled the loop, the resulting use of the same registers could prevent us from effectively scheduling the loop. Thus we will want to use different registers for each iteration, increasing the required number of registers.

Example Show our loop unrolled so that there are four copies of the loop body, assuming `x1 – x2` (that is, the size of the array) is initially a multiple of 32, which means that the number of loop iterations is a multiple of 4. Eliminate any obviously redundant computations and do not reuse any of the registers.

Answer Here is the result after merging the `addi` instructions and dropping the unnecessary `bne` operations that are duplicated during unrolling. Note that `x2` must now be set so that `Regs[x2]+32` is the starting address of the last four elements.

```
Loop: fld     f0,0(x1)
      fadd.d  f4,f0,f2
      fsd     f4,0(x1)
      fld     f6,-8(x1)
      fadd.d  f8,f6,f2
      fsd     f8,-8(x1)
      fld     f0,-16(x1)
      fadd.d  f12,f0,f2
      fsd     f12,-16(x1)
      fld     f14,-24(x1)
      fadd.d  f16,f14,f2
      fsd     f16,-24(x1)
      addi    x1,x1,-32
      bne     x1,x2,Loop
```

f2 contains increment value

load array (memory) element in CPU register
increment register value and store it to array element

//drop addi & bne

//drop addi & bne

loop body is replicated 4 times in order to get rid of 3 branch instructions and associated 3 stalls

No scheduling (reordering of instructions) has been performed here

loop index has been adjusted so as to access correct array elements

Loop unrolling is done 4 times to eliminate 3 branches, means 3 control (branch) hazards and stalls due to them were eliminated.

This speeds up the execution of the code.

This is done by compiler. This increases size of execuatble code, but speeds up computation. There is a trade off between memory used and speed archived .

We have eliminated three branches and three decrements of `x1`. The addresses on the loads and stores have been compensated to allow the `addi` instructions on `x1` to be merged. This optimization may seem trivial, but it is not; it requires symbolic substitution and simplification. Symbolic substitution and simplification will rearrange expressions so as to allow constants to be collapsed, allowing an expression such as $((i+1)+1)$ to be rewritten as $(i+(1+1))$ and then simplified to $(i+2)$.

We will see more general forms of these optimizations that eliminate dependent computations in Appendix H.

Without scheduling, every FP load or operation in the unrolled loop is followed by a dependent operation and thus will cause a stall. This unrolled loop will run in 26 clock cycles—each `fld` has 1 stall, each `fadd.d` has 2, plus 14 instruction issue cycles—or 6.5 clock cycles for each of the four elements, but it can be scheduled to improve performance significantly. Loop unrolling is normally done early in the compilation process so that redundant computations can be exposed and eliminated by the optimizer.

In real programs, we do not usually know the upper bound on the loop. Suppose it is *n,* and we want to unroll the loop to make *k* copies of the body. Instead of a single unrolled loop, we generate a pair of consecutive loops. The first executes (*n* mod *k*) times and has a body that is the original loop. The second is the unrolled body surrounded by an outer loop that iterates (*n*/*k*) times. (As we will see in Chapter 4, this technique is similar to a technique called *strip mining*, used in compilers for vector processors.) For large values of *n*, most of the execution time will be spent in the unrolled loop body.

In the previous example, unrolling improves the performance of this loop by eliminating overhead instructions, although it increases code size substantially. How will the unrolled loop perform when it is scheduled for the pipeline described earlier?

Example Show the unrolled loop in the previous example after it has been scheduled for the pipeline with the latencies in Figure 3.2.

Answer

```
Loop:  fld     f0,0(x1)
       fld     f6,-8(x1)
       fld     f0,-16(x1)
       fld     f14,-24(x1)
       fadd.d  f4,f0,f2
       fadd.d  f8,f6,f2
       fadd.d  f12,f0,f2
       fadd.d  f16,f14,f2
       fsd     f4,0(x1)
       fsd     f8,-8(x1)
       fsd     f12,16(x1)
       fsd     f16,8(x1)
       addi    x1,x1,-32
       bne     x1,x2,Loop
```

load 4 consecutive array elements in 4 different registers of CPU. These instructions are independent of each other. Hence, no stall

4 consecutive fadd.d instructions independent of each other causing no data hazard. But, if CPU does not have 4 adders, then some fadd.d may have to wait (stall) for adder becoming free. This is structural hazard.

Loop unrolling and scheduling (reordering) of instructions has been done without affecting the program output.

This done to reduce control hazards and stalls due to them. This speeds up program execution.

Compute the number of CC used to execute the code
(a) in orginal code without loop unrolling and without scheduling
(b) with loop unrolling but no scheduling
(c) without loop unrolling, but with scheduling
(d) With both unrolling as well as scheduling

Stall between load data by fld and use that data by fadd.d is eliminated by inserting (moving from below) other independent fld instructions between them.

The execution time of the unrolled loop has dropped to a total of 14 clock cycles, or 3.5 clock cycles per element, compared with 8 cycles per element before any unrolling or scheduling and 6.5 cycles when unrolled but not scheduled.

If CPU has 4 floating point adders, then no stall between consecutive fadd.d instructions.

The gain from scheduling on the unrolled loop is even larger than on the original loop. This increase arises because unrolling the loop exposes more computation that can be scheduled to minimize the stalls; the preceding code has no stalls. Scheduling the loop in this fashion necessitates realizing that the loads and stores are independent and can be interchanged.

Summary of the Loop Unrolling and Scheduling

Throughout this chapter and Appendix H, we will look at a variety of hardware and software techniques that allow us to take advantage of instruction-level parallelism to fully utilize the potential of the functional units in a processor. The key to most of these techniques is to know when and how the ordering among instructions may be changed. In our example, we made many such changes, which to us, as human beings, were obviously allowable. In practice, this process must be performed in a methodical fashion either by a compiler or by hardware. To obtain the final unrolled code, we had to make the following decisions and transformations:

- Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.
- Use different registers to avoid unnecessary constraints that would be forced by using the same registers for different computations (e.g., name dependences).
- Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.
- Determine that the loads and stores in the unrolled loop can be interchanged by observing that the loads and stores from different iterations are independent. This transformation requires analyzing the memory addresses and finding that they do not refer to the same address.
- Schedule the code, preserving any dependences needed to yield the same result as the original code.

The key requirement underlying all of these transformations is an understanding of how one instruction depends on another and how the instructions can be changed or reordered given the dependences.

Three different effects limit the gains from loop unrolling: (1) a decrease in the amount of overhead amortized with each unroll, (2) code size limitations, and (3) compiler limitations. Let's consider the question of loop overhead first. When we unrolled the loop four times, it generated sufficient parallelism among the instructions that the loop could be scheduled with no stall cycles. In fact, in 14 clock cycles, only 2 cycles were loop overhead: the `addi`, which maintains the index value, and the `bne`, which terminates the loop. If the loop is unrolled eight times, the overhead is reduced from 1/2 cycle per element to 1/4.

A second limit to unrolling is the resulting growth in code size. For larger loops, the code size growth may be a concern, particularly if it causes an increase in the instruction cache miss rate.

Another factor often more important than code size is the potential shortfall in registers that is created by aggressive unrolling and scheduling. This secondary effect that results from instruction scheduling in large code segments is called *register pressure*. It arises because scheduling code to increase ILP causes the number of live values to increase. After aggressive instruction scheduling, it may not be possible to allocate all the live values to registers. The transformed code, while theoretically faster, may lose some or all of its advantage because it leads to a shortage of registers. Without unrolling, aggressive scheduling is sufficiently limited by branches so that register pressure is rarely a problem. The combination of unrolling and aggressive scheduling can, however, cause this problem. The problem becomes especially challenging in multiple-issue processors that require the exposure of more independent instruction sequences whose execution can be overlapped. In general, the use of sophisticated high-level transformations, whose potential improvements are difficult to measure before detailed code generation, has led to significant increases in the complexity of modern compilers.

Loop unrolling is a simple but useful method for increasing the size of straight-line code fragments that can be scheduled effectively. This transformation is useful in a variety of processors, from simple pipelines like those we have examined so far to the multiple-issue superscalars and VLIWs explored later in this chapter.

3.3 Reducing Branch Costs With Advanced Branch Prediction

Because of the need to enforce control dependences through branch hazards and stalls, branches will hurt pipeline performance. Loop unrolling is one way to reduce the number of branch hazards; we can also reduce the performance losses of branches by predicting how they will behave. In Appendix C, we examine simple branch predictors that rely either on compile-time information or on the observed dynamic behavior of a single branch in isolation. As the number of instructions in flight has increased with deeper pipelines and more issues per clock, the importance of more accurate branch prediction has grown. In this section, we examine techniques for improving dynamic prediction accuracy. This section makes extensive use of the simple 2-bit predictor covered in Section C.2, and it is critical that the reader understand the operation of that predictor before proceeding.

Correlating Branch Predictors

The 2-bit predictor schemes in Appendix C 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 eqntott benchmark, a member of early SPEC benchmark suites that displayed particularly bad branch prediction behavior:

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

Here is the RISC-V code that we would typically generate for this code fragment assuming that `aa` and `bb` are assigned to registers `x1` and `x2`:

```
     addi  x3,x1,-2
     bnez  x3,L1       //branch b1  (aa!=2)
     add   x1,x0,x0    //aa=0
L1:  addi  x3,x2,-2
     bnez  x3,L2       //branch b2  (bb!=2)
     add   x2,x0,x0    //bb=0
L2:  sub   x3,x1,x2    //x3=aa-bb
     beqz  x3,L3       //branch b3  (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 neither branches b1 nor b2 are taken (i.e., if the conditions both evaluate to true and `aa` and `bb` are both assigned 0), then b3 will be taken, because `aa` and `bb` are clearly equal. A predictor that uses the behavior of only 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*. Existing correlating predictors add information about the behavior of the most recent branches to decide how to predict a given branch. For example, a (1,2) predictor uses the behavior of the last branch to choose from among a pair of 2-bit branch predictors in predicting a particular branch. 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 an *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 2-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, in a (2,2) buffer with 64 total entries, the 4 low-order address bits of the branch (word address) and the 2 global bits representing the behavior of the two most recently executed branches form a 6-bit index that can be used to index the 64 counters. By combining the local and global information by concatenation (or a simple hash function), we can index the predictor table with the result and get a prediction as fast as we could for the standard 2-bit predictor, as we will do very shortly.

How much better do the correlating branch predictors work when compared with the standard 2-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 2-bit predictor with no global history is simply a (0,2) predictor.

Example How many bits are in the (0,2) branch predictor with 4K entries? How many entries are in a (2,2) predictor with the same number of bits?

Answer The predictor with 4K entries has

$$2^0 \times 2 \times 4\text{K} = 8\text{K bits}$$

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

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

Therefore the number of prediction entries selected by the branch = 1K.

Figure 3.3 compares the misprediction rates of the earlier (0,2) predictor with 4K entries and a (2,2) predictor with 1K entries. As you can see, this correlating predictor not only outperforms a simple 2-bit predictor with the same total number of state bits, but it also often outperforms a 2-bit predictor with an unlimited number of entries.

Perhaps the best-known example of a correlating predictor is McFarling's gshare predictor. In gshare the index is formed by combining the address of the branch and the most recent conditional branch outcomes using an exclusive-OR, which essentially acts as a hash of the branch address and the branch history. The hashed result is used to index a prediction array of 2-bit counters, as shown in Figure 3.4. The gshare predictor works remarkably well for a simple predictor, and is often used as the baseline for comparison with more sophisticated predictors. Predictors that combine local branch information and global branch history are also called *alloyed predictors or hybrid predictors*.

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. Adding global history could help remedy this situation. *Tournament predictors* take this insight to the next level, by using multiple predictors, usually a global predictor and a local predictor, and choosing between them

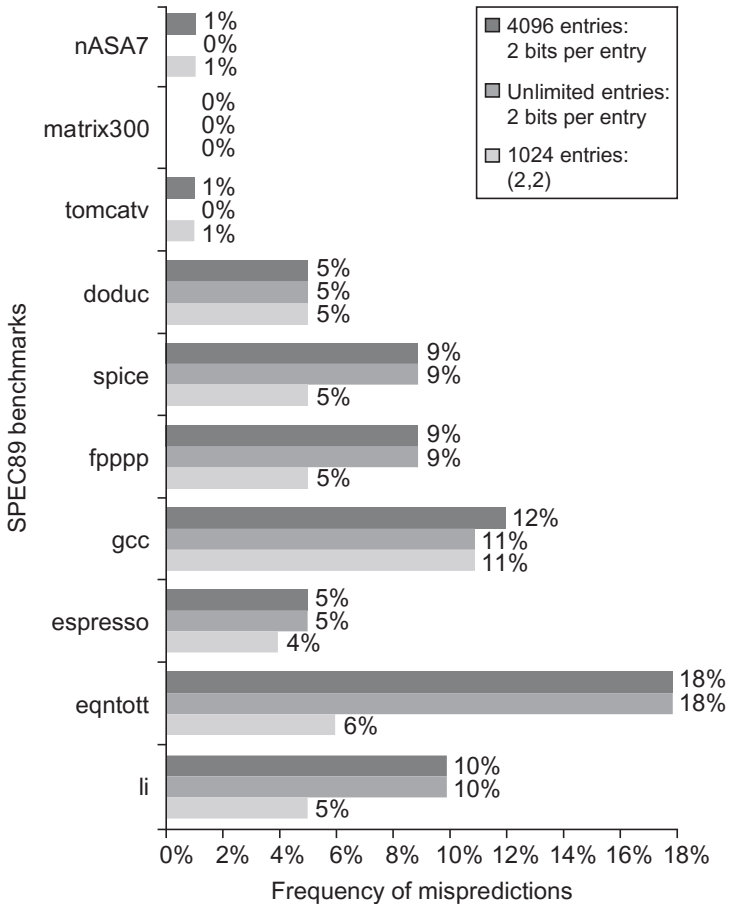

Figure 3.3 Comparison of 2-bit predictors. A noncorrelating predictor for 4096 bits is first, followed by a noncorrelating 2-bit predictor with unlimited entries and a 2-bit predictor with 2 bits of global history and a total of 1024 entries. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks would show similar differences in accuracy.

with a selector, as shown in Figure 3.5. A *global predictor* uses the most recent branch history to index the predictor, while a *local predictor* uses the address of the branch as the index. Tournament predictors are another form of hybrid or alloyed predictors.

Tournament predictors can achieve better accuracy at medium sizes (8K–32K bits) and also effectively use very large numbers of prediction bits. Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor (local, global, or even some time-varying mix) was most effective in recent predictions. As in a simple 2-bit predictor, the saturating counter requires two mispredictions before changing the identity of the preferred predictor.

The advantage of a tournament predictor is its ability to select the right predictor for a particular branch, which is particularly crucial for the integer benchmarks.

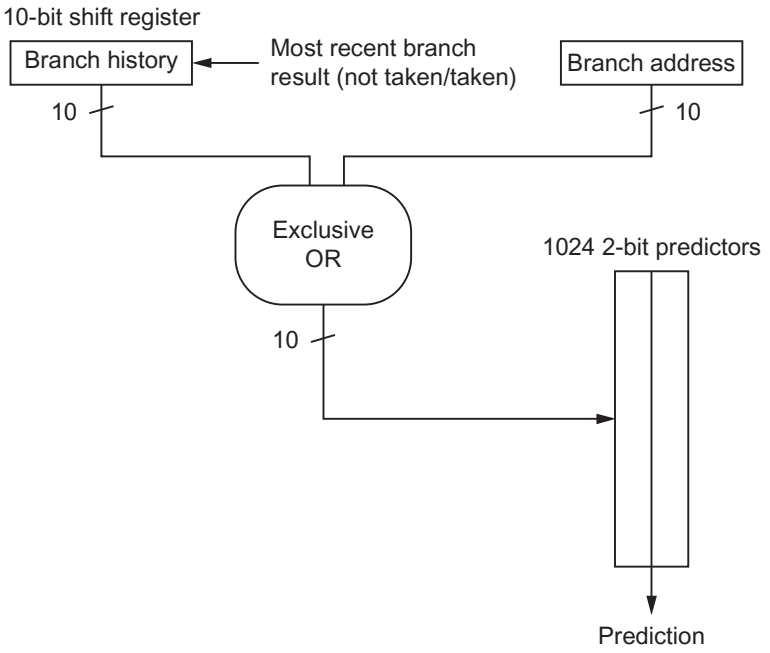

Figure 3.4 A gshare predictor with 1024 entries, each being a standard 2-bit predictor.

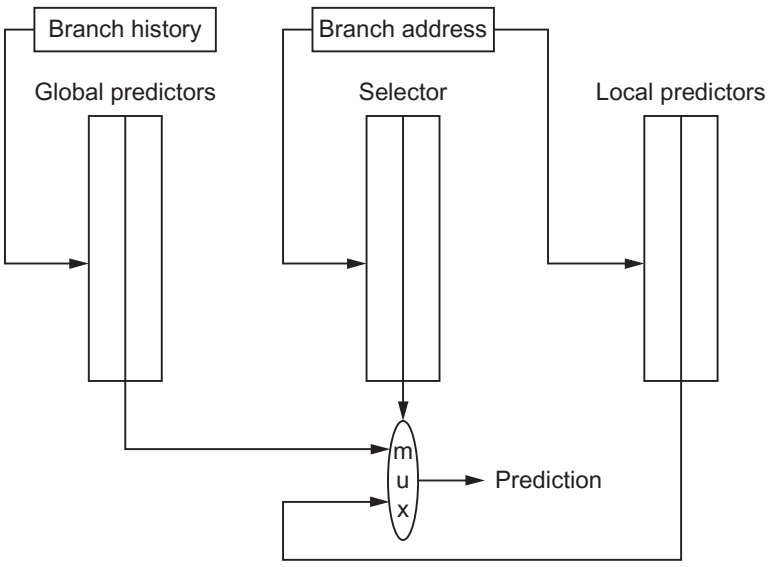

Figure 3.5 A tournament predictor using the branch address to index a set of 2-bit selection counters, which choose between a local and a global predictor. In this case, the index to the selector table is the current branch address. The two tables are also 2-bit predictors that are indexed by the global history and branch address, respectively. The selector acts like a 2-bit predictor, changing the preferred predictor for a branch address when two mispredicts occur in a row. The number of bits of the branch address used to index the selector table and the local predictor table is equal to the length of the global branch history used to index the global prediction table. Note that misprediction is a bit tricky because we need to change both the selector table and either the global or local predictor.

A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks. In addition to the Alpha processors that pioneered tournament predictors, several AMD processors have used tournament-style predictors.

Figure 3.6 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. The local predictor reaches its limit first. The correlating predictor shows a significant improvement, and the tournament predictor generates a slightly better performance. For more recent versions of the SPEC, the results would be similar, but the asymptotic behavior would not be reached until slightly larger predictor sizes.

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 10 branch outcomes for the entry. That is, if the branch is 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 consists of alternating 0s and 1s. This 10-bit history allows patterns of up to 10 branches to be discovered and predicted. The selected entry from the local history table is used to index a table of 1K entries consisting of 3-bit saturating counters, which provide the local prediction. This combination, which uses a total of 29K bits, leads to high accuracy in

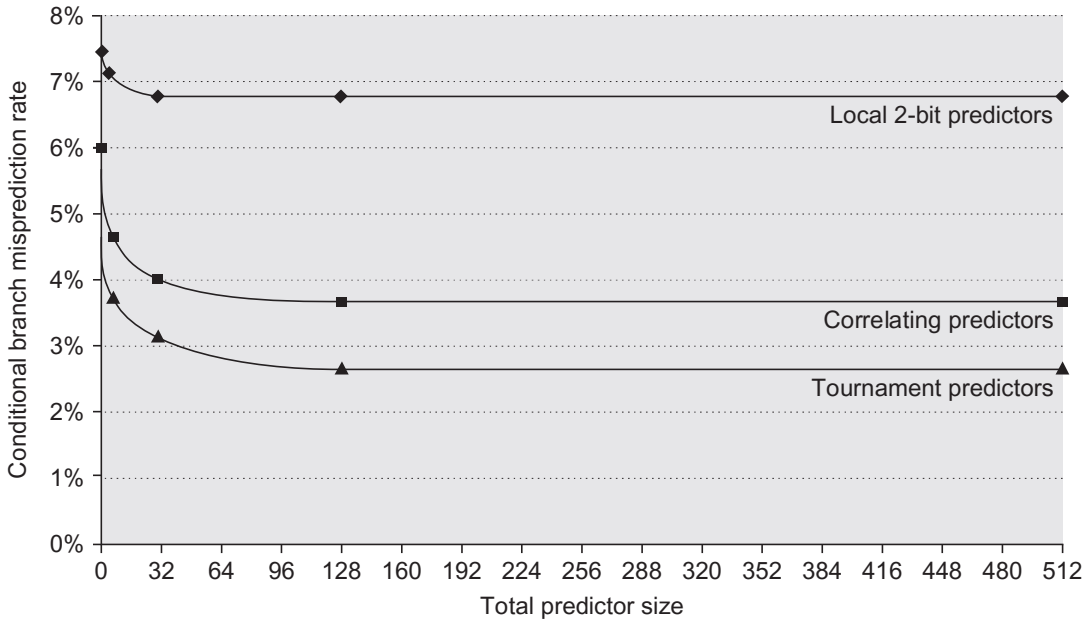

Figure 3.6 The misprediction rate for three different predictors on SPEC89 versus the size of the predictor in kilobits. The predictors are a local 2-bit predictor, a correlating predictor that is optimally structured in its use of global and local information at each point in the graph, and a tournament predictor. Although these data are for an older version of SPEC, data for more recent SPEC benchmarks show similar behavior, perhaps converging to the asymptotic limit at slightly larger predictor sizes.

branch prediction while requiring fewer bits than a single level table with the same prediction accuracy.

Tagged Hybrid Predictors

The best performing branch prediction schemes as of 2017 involve combining multiple predictors that track whether a prediction is likely to be associated with the current branch. One important class of predictors is loosely based on an algorithm for statistical compression called PPM (Prediction by Partial Matching). PPM (see Jiménez and Lin, 2001), like a branch prediction algorithm, attempts to predict future behavior based on history. This class of branch predictors, which we call *tagged hybrid predictors* (see Seznec and Michaud, 2006), employs a series of global predictors indexed with different length histories.

For example, as shown in Figure 3.7, a five-component tagged hybrid predictor has five prediction tables: P(0), P(1), . . . P(4), where P(*i*) is accessed using a hash of

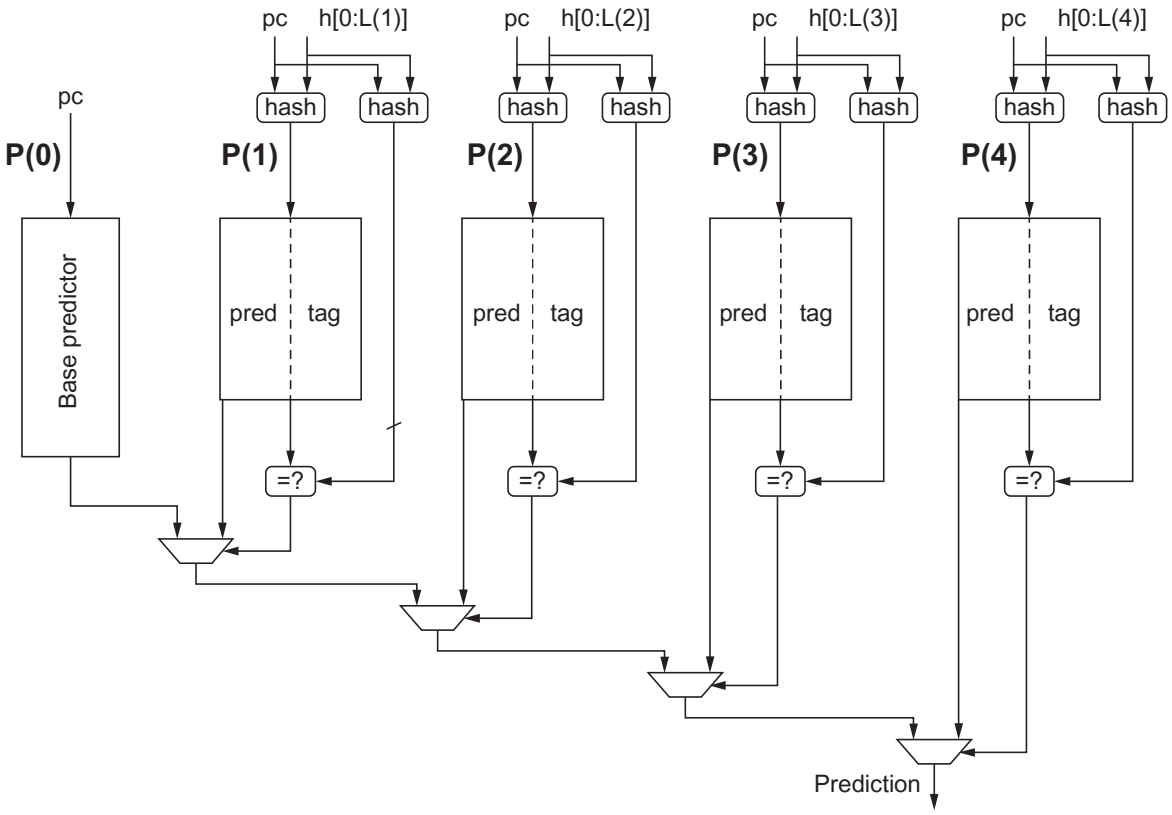

Figure 3.7 A five-component tagged hybrid predictor has five separate prediction tables, indexed by a hash of the branch address and a segment of recent branch history of length 0–4 labeled "h" in this figure. The hash can be as simple as an exclusive-OR, as in gshare. Each predictor is a 2-bit (or possibly 3-bit) predictor. The tags are typically 4–8 bits. The chosen prediction is the one with the longest history where the tags also match.

the PC and the history of the most recent *i* branches (kept in a shift register, h, just as in gshare). The use of multiple history lengths to index separate predictors is the first critical difference. The second critical difference is the use of tags in tables P(1) through P(4). The tags can be short because 100% matches are not required: a small tag of 4–8 bits appears to gain most of the advantage. A prediction from P(1), . . . P(4) is used only if the tags match the hash of the branch address and global branch history. Each of the predictors in P(0…*n*) can be a standard 2-bit predictor. In practice a 3-bit counter, which requires three mispredictions to change a prediction, gives slightly better results than a 2-bit counter.

The prediction for a given branch is the predictor with the longest branch history that also has matching tags. P(0) always matches because it uses no tags and becomes the default prediction if none of P(1) through P(*n*) match. The tagged hybrid version of this predictor also includes a 2-bit use field in each of the history-indexed predictors. The use field indicates whether a prediction was recently used and therefore likely to be more accurate; the use field can be periodically reset in all entries so that old predictions are cleared. Many more details are involved in implementing this style of predictor, especially how to handle mispredictions. The search space for the optimal predictor is also very large because the number of predictors, the exact history used for indexing, and the size of each predictor are all variable.

Tagged hybrid predictors (sometimes called TAGE—TAgged GEometic—predictors) and the earlier PPM-based predictors have been the winners in recent annual international branch-prediction competitions. Such predictors outperform gshare and the tournament predictors with modest amounts of memory (32–64 KiB), and in addition, this class of predictors seems able to effectively use larger prediction caches to deliver improved prediction accuracy.

Another issue for larger predictors is how to initialize the predictor. It could be initialized randomly, in which case, it will take a fair amount of execution time to fill the predictor with useful predictions. Some predictors (including many recent predictors) include a valid bit, indicating whether an entry in the predictor has been set or is in the "unused state." In the latter case, rather than use a random prediction, we could use some method to initialize that prediction entry. For example, some instruction sets contain a bit that indicates whether an associated branch is expected to be taken or not. In the days before dynamic branch prediction, such hint bits *were* the prediction; in recent processors, that hint bit can be used to set the initial prediction. We could also set the initial prediction on the basis of the branch direction: forward going branches are initialized as not taken, while backward going branches, which are likely to be loop branches, are initialized as taken. For programs with shorter running times and processors with larger predictors, this initial setting can have a measurable impact on prediction performance.

Figure 3.8 shows that a hybrid tagged predictor significantly outperforms gshare, especially for the less predictable programs like SPECint and server applications. In this figure, performance is measured as mispredicts per thousand instructions; assuming a branch frequency of 20%–25%, gshare has a mispredict rate (per branch) of 2.7%–3.4% for the multimedia benchmarks, while the tagged

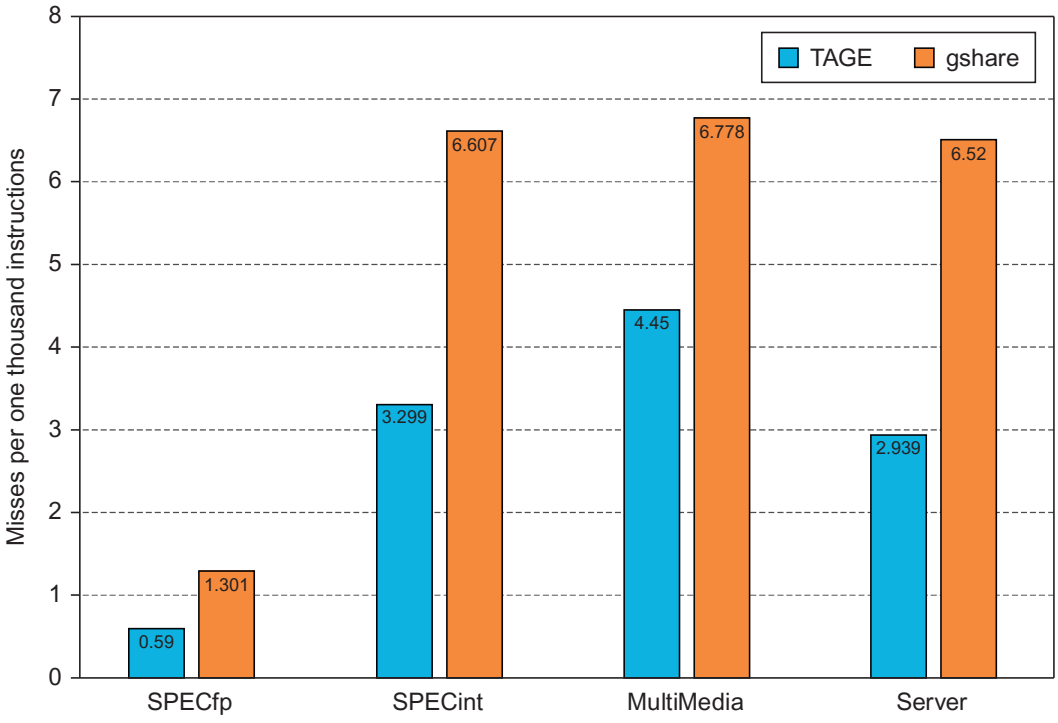

Figure 3.8 A comparison of the misprediction rate (measured as mispredicts per 1000 instructions executed) for tagged hybrid versus gshare. Both predictors use the same total number of bits, although tagged hybrid uses some of that storage for tags, while gshare contains no tags. The benchmarks consist of traces from SPECfp and SPECint, a series of multimedia and server benchmarks. The latter two behave more like SPECint.

hybrid predictor has a misprediction rate of 1.8%–2.2%, or roughly one-third fewer mispredicts. Compared to gshare, tagged hybrid predictors are more complex to implement and are probably slightly slower because of the need to check multiple tags and choose a prediction result. Nonetheless, for deeply pipelined processors with large penalties for branch misprediction, the increased accuracy outweighs those disadvantages. Thus many designers of higher-end processors have opted to include tagged hybrid predictors in their newest implementations.

The Evolution of the Intel Core i7 Branch Predictor

As mentioned in the previous chapter, there were six generations of Intel Core i7 processors between 2008 (Core i7 920 using the Nehalem microarchitecture) and 2016 (Core i7 6700 using the Skylake microarchitecture). Because of the combination of deep pipelining and multiple issues per clock, the i7 has many instructions in-flight at once (up to 256, and typically at least 30). This makes branch prediction critical, and it has been an area where Intel has been making constant improvements. Perhaps because of the performance-critical nature of the branch predictor, Intel has tended to keep the details of its branch predictors highly secret.

Even for older processors such as the Core i7 920 introduced in 2008, they have released only limited amounts of information. In this section, we briefly describe what is known and compare the performance of predictors of the Core i7 920 with those in the latest Core i7 6700.

The Core i7 920 used a two-level predictor that has a smaller first-level predictor, designed to meet the cycle constraints of predicting a branch every clock cycle, and a larger second-level predictor as a backup. Each predictor combines three different predictors: (1) the simple 2-bit predictor, which is introduced in Appendix C (and used in the preceding tournament predictor); (2) a global history predictor, like those we just saw; and (3) a loop exit predictor. The loop exit predictor uses a counter to predict the exact number of taken branches (which is the number of loop iterations) for a branch that is detected as a loop branch. For each branch, the best prediction is chosen from among the three predictors by tracking the accuracy of each prediction, like a tournament predictor. In addition to this multilevel main predictor, a separate unit predicts target addresses for indirect branches, and a stack to predict return addresses is also used.

Although even less is known about the predictors in the newest i7 processors, there is good reason to believe that Intel is employing a tagged hybrid predictor. One advantage of such a predictor is that it combines the functions of all three second-level predictors in the earlier i7. The tagged hybrid predictor with different history lengths subsumes the loop exit predictor as well as the local and global history predictor. A separate return address predictor is still employed.

As in other cases, speculation causes some challenges in evaluating the predictor because a mispredicted branch can easily lead to another branch being fetched and mispredicted. To keep things simple, we look at the number of mispredictions as a percentage of the number of successfully completed branches (those that were not the result of misspeculation). Figure 3.9 shows these data for SPECPUint2006 benchmarks. These benchmarks are considerably larger than SPEC89 or SPEC2000, with the result being that the misprediction rates are higher than those in Figure 3.6 even with a more powerful combination of predictors. Because branch misprediction leads to ineffective speculation, it contributes to the wasted work, as we will see later in this chapter.

3.4 Overcoming Data Hazards With Dynamic Scheduling

A simple statically scheduled pipeline fetches an instruction and issues it, unless there is a data dependence between an instruction already in the pipeline and the fetched instruction that cannot be hidden with bypassing or forwarding. (Forwarding logic reduces the effective pipeline latency so that the certain dependences do not result in hazards.) If there is 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 *dynamic scheduling*, a technique by which the hardware reorders the instruction execution to reduce the stalls while maintaining data

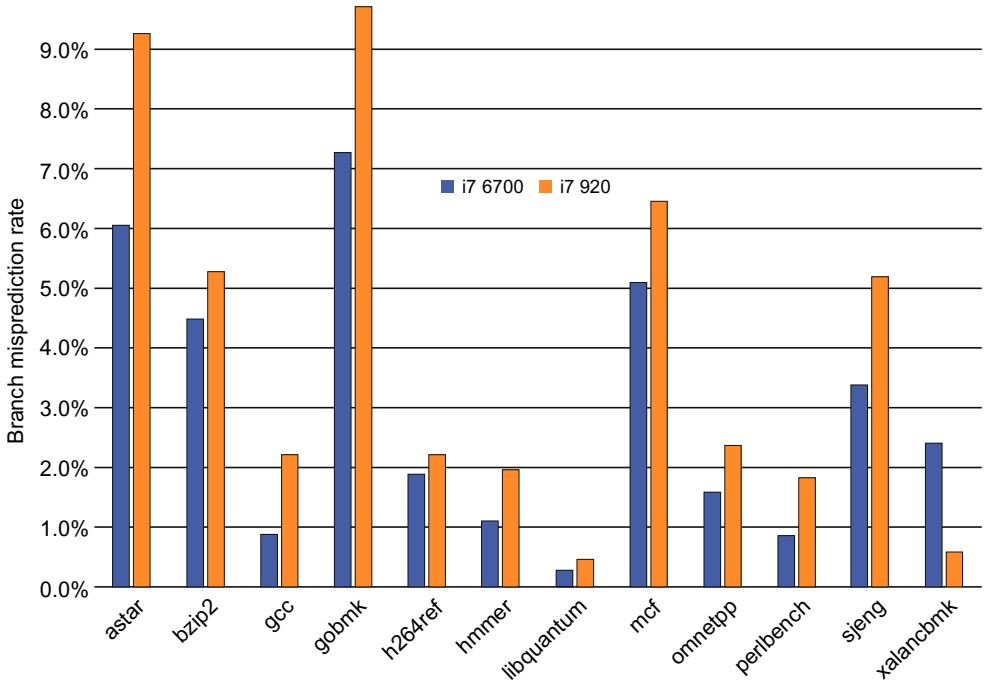

Figure 3.9 The misprediction rate for the integer SPECCPU2006 benchmarks on the Intel Core i7 920 and 6700. The misprediction rate is computed as the ratio of completed branches that are mispredicted versus all completed branches. This could understate the misprediction rate somewhat because if a branch is mispredicted and led to another mispredicted branch (which should not have been executed), it will be counted as only one misprediction. On average, the i7 920 mispredicts branches 1.3 times as often as the i7 6700.

flow and exception behavior. Dynamic scheduling offers several advantages. First, it allows code that was compiled with one pipeline in mind to run efficiently on a different pipeline, eliminating the need to have multiple binaries and recompile for a different microarchitecture. In today's computing environment, where much of the software is from third parties and distributed in binary form, this advantage is significant. Second, it enables handling some cases when dependences are unknown at compile time; for example, they may involve a memory reference or a data-dependent branch, or they may result from a modern programming environment that uses dynamic linking or dispatching. Third, and perhaps most importantly, it allows the processor to tolerate unpredictable delays, such as cache misses, by executing other code while waiting for the miss to resolve. In Section 3.6, we explore hardware speculation, a technique with additional performance advantages, which builds on dynamic scheduling. As we will see, the advantages of dynamic scheduling are gained at the 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 are present. In contrast, static pipeline scheduling by the compiler (covered in Section 3.2) 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 in code destined to run on a processor with a dynamically scheduled pipeline.

Dynamic Scheduling: The Idea

A major limitation of simple pipelining techniques is that they 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, it will lead to a hazard, and a stall will result. If there are multiple functional units, these units could lie idle. If instruction *j* depends on a long-running instruction *i,* currently in execution in the pipeline, then all instructions after *j* must be stalled until *i* is finished and *j* can execute. For example, consider this code:

```
fdiv.d  f0,f2,f4
fadd.d  f10,f0,f8
fsub.d  f12,f8,f14
```

The `fsub.d` instruction cannot execute because the dependence of `fadd.d` on `fdiv.d` causes the pipeline to stall; yet, `fsub.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, both structural and data hazards could be checked during instruction decode (ID): when an instruction could execute without hazards, it was issued from ID, with the recognition that all data hazards had been resolved.

To allow us to begin executing the `fsub.d` in the preceding example, we must separate the issue process into two parts: checking for any structural hazards and waiting for the absence of a data hazard. Thus we still use in-order instruction issue (i.e., instructions issued in program order), but we want an instruction to begin execution as soon as its data operands are available. Such a pipeline does *out-of-order execution*, which implies *out-of-order completion*.

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

```
fdiv.d  f0,f2,f4
fmul.d  f6,f0,f8
fadd.d  f0,f10,f14
```

There is an antidependence between the `fmul.d` and the `fadd.d` (for the register `f0`), and if the pipeline executes the `fadd.d` before the `fmul.d` (which is waiting for the `fdiv.d`), it will violate the antidependence, yielding a WAR hazard. Likewise, to avoid violating output dependences, such as the write of `f0` by `fadd.d` before `fdiv.d` completes, WAW hazards must be handled. As we will see, both these hazards are avoided by the use of register renaming.

Out-of-order completion also creates major complications in handling exceptions. Dynamic scheduling with out-of-order completion must preserve exception behavior in the sense that *exactly* those exceptions that would arise if the program were executed in strict program order *actually* do arise. Dynamically scheduled processors preserve exception behavior by delaying the notification of an associated exception until the processor knows that the instruction should be the next one completed.

Although exception behavior must be preserved, dynamically scheduled processors could generate *imprecise* exceptions. An exception is *imprecise* if the processor state when an exception is raised does not look exactly as if the instructions were executed sequentially in strict program order. Imprecise exceptions can occur because of two possibilities:

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

Imprecise exceptions make it difficult to restart execution after an exception. Rather than address these problems in this section, we will discuss a solution that provides precise exceptions in the context of a processor with speculation in Section 3.6. For floating-point exceptions, other solutions have been used, as discussed in Appendix J.

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

We 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; 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. Because 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 can 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. Here we focus on a more sophisticated technique, called *Tomasulo's algorithm.* The primary difference is that Tomasulo's algorithm handles antidependences and output dependences by effectively renaming the registers dynamically. Additionally, Tomasulo's algorithm can be extended to handle *speculation*, a technique to reduce the effect of control dependences by predicting the outcome of a branch, executing instructions at the predicted destination address, and taking corrective actions when the prediction was wrong. While the use of scoreboarding is probably sufficient to support simpler processors, more sophisticated, higher performance processors make use of speculation.

Dynamic Scheduling Using Tomasulo's Approach

The IBM 360/91 floating-point unit used a sophisticated scheme to allow out-of-order execution. This scheme, invented by Robert Tomasulo, tracks when operands for instructions are available to minimize RAW hazards and introduces register renaming in hardware to minimize WAW and WAR hazards. Although there are many variations of this scheme in recent processors, they all rely on two key principles: dynamically determining when an instruction is ready to execute and renaming registers to avoid unnecessary hazards.

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 limited 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 RISC-V instruction set. The primary difference between RISC-V and the 360 is the presence of register-memory instructions in the latter architecture. 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.

RAW hazards are avoided by executing an instruction only when its operands are available, which is exactly what the simpler scoreboarding approach provides. 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. The compiler could typically implement such renaming, if there were enough registers available in the ISA. The original 360/91 had only four floating-point registers, and Tomasulo's algorithm was created to overcome this shortage. Whereas modern processors have 32–64 floating-point and integer registers, the number of renaming registers available in recent implementations is in the hundreds.

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

```
fdiv.d  f0,f2,f4
fadd.d  f6,f0,f8
fsd     f6,0(x1)
fsub.d  f8,f10,f14
fmul.d  f6,f10,f8
```

There are two antidependences: between the `fadd.d` and the `fsub.d` and between the `fsd` and the `fmul.d`. There is also an output dependence between the `fadd.d` and the `fmul.d`, leading to three possible hazards: WAR hazards on the use of `f8` by `fadd.d` and its use by the `fsub.d`, as well as a WAW hazard because the `fadd.d` may finish later than the `fmul.d`. There are also three true data dependences: between the `fdiv.d` and the `fadd.d`, between the `fsub.d` and the `fmul.d`, and between the `fadd.d` and the `fsd`.

These three name dependences can all 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

```
fdiv.d  f0,f2,f4
fadd.d  S,f0,f8
fsd     S,0(x1)
fsub.d  T,f10,f14
fmul.d  f6,f10,T
```

In addition, any subsequent uses of `f8` must be replaced by the register `T`. In this example, 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 because there may be intervening branches between the preceding 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 *reservation stations*, which buffer the operands of instructions waiting to issue and are associated with the functional units. 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.

Because 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 determines 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 that issue multiple instructions per clock and also have multiple execution units, more than one result bus will be needed.

Figure 3.10 shows the basic structure of a Tomasulo-based processor, including both the floating-point unit and the load/store unit; none of the execution control tables is shown. Each reservation station holds an instruction that has been issued and is awaiting execution at a functional unit. If the operand values for that instruction have been computed, they are also stored in that entry; otherwise, the reservation station entry keeps the names of the reservation stations that will provide the operand values.

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

Before we describe the details of the reservation stations and the algorithm, let's look at the steps an instruction goes through. There are only three steps, although 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 issue stalls until a station or buffer is freed. If the operands are not in the registers, keep track of the functional units that will produce the operands. This step renames registers, eliminating WAR and WAW hazards. (This stage is sometimes called *dispatch* in a dynamically scheduled processor.)

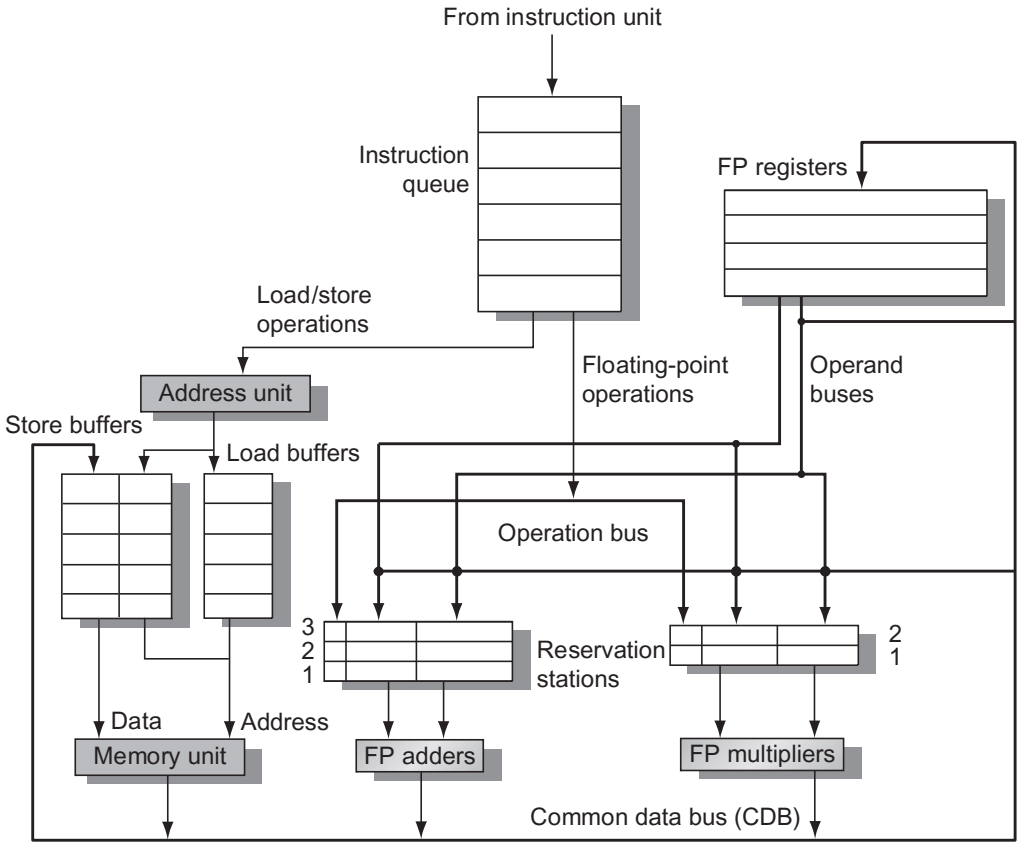

Figure 3.10 The basic structure of a RISC-V floating-point unit using Tomasulo's algorithm. Instructions are sent from the instruction unit into the instruction queue from which they are issued in first-in, first-out (FIFO) order. The reservation stations include the operation and the actual operands, as well as information used for detecting and resolving hazards. Load buffers have three functions: (1) hold the components of the effective address until it is computed, (2) track outstanding loads that are waiting on the memory, and (3) hold the results of completed loads that are waiting for the CDB. Similarly, store buffers have three functions: (1) hold the components of the effective address until it is computed, (2) hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and (3) hold the address and value to store until the memory unit is available. All results from either the FP units or the load unit are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers. The FP adders implement addition and subtraction, and the FP multipliers do multiplication and division.

2. *Execute*—If one or more of the operands is not yet available, monitor the common data bus while waiting for it to be computed. When an operand becomes available, it is placed into any reservation station awaiting it. 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. (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.)

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 for the value to be stored before being sent to the memory unit. Loads and stores are maintained in program order through the effective address calculation, which will help to prevent hazards through memory.

To preserve exception behavior, no instruction is allowed to initiate execution until a branch that precedes the instruction in program order has completed. This restriction guarantees that an instruction that causes an exception during execution really would have been executed. In a processor using branch prediction (as all dynamically scheduled processors do), this means that the processor must know that the branch prediction was correct before allowing an instruction after the branch to begin execution. If the processor records the occurrence of the exception, but does not actually raise it, an instruction can start execution but not stall until it enters Write Result.

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 are buffered in the store buffer until both the value to be stored and the store address are available; then the result is written as soon as the memory unit is free.

The data structures that 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 for renaming. In our example, the tag field is a 4-bit quantity that denotes one of the five reservation stations or one of the five load buffers. This combination produces the equivalent of 10 registers (5 reservation stations + 5 load buffers) 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 want renaming to provide an even larger set of virtual registers, often numbering in the hundreds. 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 3.6.

In Tomasulo's scheme, as well as the subsequent methods we look at for supporting speculation, results are broadcast on a bus (the CDB), which is monitored by the reservation stations. The combination of the common result bus and the retrieval of results from the bus by the reservation stations implements the forwarding and bypassing mechanisms used in a statically scheduled pipeline. In doing so, however, a dynamically scheduled scheme, such as Tomasulo's algorithm, introduces one cycle of latency between source and result because the matching of a result and its use cannot be done until the end of the Write Result stage, as opposed to the end of the Execute stage for a simpler pipeline. Thus, in a dynamically scheduled pipeline, the effective latency between a producing instruction and a consuming instruction is at least one cycle longer than the latency of the functional unit producing the result.

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. (This is a key difference between Tomasulo's scheme and scoreboarding: in scoreboarding, operands stay in the registers and are read only after the producing instruction completes and the consuming instruction is ready to execute.)

Each reservation station has seven fields:

- Op—The operation to perform on source operands S1 and S2.
- Qj, Qk—The reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary.
- Vj, Vk—The value of the source operands. Note that only one of the V fields or the Q field is valid for each operand. For loads, the Vk field is used to hold the offset field.
- 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.

3.5 Dynamic Scheduling: Examples and the Algorithm

Before we examine Tomasulo's algorithm in detail, let's consider a few examples that 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.  fld     f6,32(x2)
2.  fld     f2,44(x3)
3.  fmul.d  f0,f2,f4
4.  fsub.d  f8,f2,f6
5.  fdiv.d  f0,f0,f6
6.  fadd.d  f6,f8,f2
```

Answer Figure 3.11 shows the result in three tables. 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 of the result on the CDB. If a centralized register file were used, the units would have to read their results from the registers when register buses were available.

The second advantage, the elimination of WAW and WAR hazards, is accomplished by renaming registers using the reservation stations and by the process of storing operands into the reservation station as soon as they are available.

For example, the code sequence in Figure 3.11 issues both the `fdiv.d` and the `fadd.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 `fdiv.d` has completed, then Vk will store the result, allowing `fdiv.d` to execute independent of the `fadd.d` (this is the case shown). On the other hand, if the `fld` hasn't completed, then Qk will point to the Load1 reservation station, and the `fdiv.d` instruction will be independent of the `fadd.d`. Thus, in either case, the `fadd.d` can issue and begin executing. Any uses of the result of the `fdiv.d` will point to the reservation station, allowing the `fadd.d` to complete and store its value into the registers without affecting the `fdiv.d`.

	Instruction status		
Instruction	**Issue**	**Execute**	**Write result**
`fld f6,32(x2)`	√	√	√
`fld f2,44(x3)`	√	√	
`fmul.d f0,f2,f4`	√		
`fsub.d f8,f2,f6`	√		
`fdiv.d f0,f0,f6`	√		
`fadd.d f6,f8,f2`	√		

	Reservation stations						
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**A**
Load1	No						
Load2	Yes	Load					`44 + Regs[x3]`
Add1	Yes	SUB		`Mem[32 + Regs[x2]]`	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		`Regs[f4]`	Load2		
Mult2	Yes	DIV		`Mem[32 + Regs[x2]]`	Mult1		

	Register status								
Field	**f0**	**f2**	**f4**	**f6**	**f8**	**f10**	**f12**	…	**f30**
Qi	Mult1	Load2		Add2	Add1	Mult2			

Figure 3.11 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 `fadd.d` instruction, which has a WAR hazard at the WB stage, has issued and could complete before the `fdiv.d` initiates.

We'll see an example of the elimination of a WAW hazard shortly. But let's first look at how our earlier example continues execution. In this example, and the ones that follow in this chapter, assume the following latencies: load is 1 clock cycle, add is 2 clock cycles, multiply is 6 clock cycles, and divide is 12 clock cycles.

Example Using the same code segment as in the previous example (page 201), show what the status tables look like when the `fmul.d` is ready to write its result.

Answer The result is shown in the three tables in Figure 3.12. Notice that `fadd.d` has completed because the operands of `fdiv.d` were copied, thereby overcoming the WAR hazard. Notice that even if the load of `f6` was `fdiv.d`, the add into `f6` could be executed without triggering a WAW hazard.

		Instruction status		
Instruction		**Issue**	**Execute**	**Write result**
`fld`	`f6,32(x2)`	√	√	√
`fld`	`f2,44(x3)`	√	√	√
`fmul.d`	`f0,f2,f4`	√	√	
`fsub.d`	`f8,f2,f6`	√	√	√
`fdiv.d`	`f0,f0,f6`	√		
`fadd.d`	`f6,f8,f2`	√	√	√

	Reservation stations						
Name	**Busy**	**Op**	**Vj**	**Vk**	**Qj**	**Qk**	**A**
Load1	No						
Load2	No						
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL	`Mem[44 + Regs[x3]]`	`Regs[f4]`			
Mult2	Yes	DIV		`Mem[32 + Regs[x2]]`	Mult1		

	Register status								
Field	**f0**	**f2**	**f4**	**f6**	**f8**	**f10**	**f12**	…	**f30**
Qi	Mult1					Mult2			

Figure 3.12 Multiply and divide are the only instructions not finished.

Tomasulo's Algorithm: The Details

Figure 3.13 specifies 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 3.6.

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 following simple sequence for multiplying the elements of an array by a scalar in `f2`:

```
Loop:  fld     f0,0(x1)
       fmul.d  f4,f0,f2
       fsd     f4,0(x1)
       addi    x1,x1,-8
       bne     x1,x2,Loop  // branches if x1≠x2
```

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

Let's assume we have issued all the instructions in two successive iterations of the loop, but none of the floating-point load/stores or operations have completed. Figure 3.14 shows reservation stations, register status tables, and load and store buffers at this point. (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. With a latency of six cycles, additional iterations will need to be processed before the steady state can be reached. This requires more reservation stations to hold instructions that are in execution. As we will see later in this chapter, when extended with multiple issue instructions, Tomasulo's approach can sustain more than one instruction per clock.

A load and a store can be done safely out of order, provided they access different addresses. If a load and a store access the same address, one of two things happens:

Instruction state	Wait until	Action or bookkeeping

Issue
FP operation
	Station r empty	`if (RegisterStat[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].Qi`
		`else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`
		`RS[r].Busy ← yes; RegisterStat[rd].Q ← r;`

Load or store	Buffer r empty	`if (RegisterStat[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 (RegisterStat[rt].Qi≠0)`
		`  {RS[r].Qk ← RegisterStat[rs].Qi}`
		`else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`

Execute FP operation	`(RS[r].Qj = 0)` and `(RS[r].Qk = 0)`	Compute result: operands are in Vj and Vk
Load/storestep 1	RS[r].Qj = 0 & r is head of load-store queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 complete	Read from `Mem[RS[r].A]`

Write result
FP operation
or load
	Execution complete at r & CDB available	`∀x(if (RegisterStat[x].Qi=r) {Regs[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 3.13 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 an 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 values of Qj or Qk are 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 no concept of program order is 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 3.6, we will see how speculation support removes this restriction.

		Instruction status			
Instruction		From iteration	Issue	Execute	Write result
`fld`	`f0,0(x1)`	1	√	√	
`fmul.d`	`f4,f0,f2`	1	√		
`fsd`	`f4,0(x1)`	1	√		
`fld`	`f0,0(x1)`	2	√	√	
`fmul.d`	`f4,f0,f2`	2	√		
`fsd`	`f4,0(x1)`	2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	Yes	Load					`Regs[x1] + 0`
Load2	Yes	Load					`Regs[x1] – 8`
Add1	No						
Add2	No						
Add3	No						
Mult1	Yes	MUL		`Regs[f2]`	Load1		
Mult2	Yes	MUL		`Regs[f2]`	Load2		
Store1	Yes	Store	`Regs[x1]`			Mult1	
Store2	Yes	Store	`Regs[x1] – 8`			Mult2	

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	…	f30
Qi	Load2		Mult2						

Figure 3.14 Two active iterations of the loop with no instruction yet completed. Entries in the multiplier reservation stations indicate that the outstanding loads are the sources. The store reservation stations indicate that the multiply destination is the source of the value to store.

- The load is before the store in program order and interchanging them results in a WAR hazard.
- 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 a WAW hazard.

Therefore, 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. We consider a method to eliminate this restriction in Section 3.9.

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 to 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, that 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 because conflicting stores cannot be reordered with respect to either a load or a store.

A dynamically scheduled pipeline can yield very high performance, provided branches are predicted accurately—an issue we addressed in the previous 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. The performance can also be limited by the single CDB. Although additional CDBs can be added, each CDB must interact with each reservation station, and the associative tag-matching hardware would have to be duplicated at each station for each CDB. In the 1990s, only high-end processors could take advantage of dynamic scheduling (and its extension to speculation); however, recently even processors designed for PMDs are using these techniques, and processors for high-end desktops and small servers have hundreds of buffers to support dynamic scheduling.

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 was unused for many years after the 360/91, but was widely adopted in multiple-issue processors starting in the 1990s for several reasons:

1. Although Tomasulo's algorithm was designed before caches, the presence of caches, with the inherently unpredictable delays, has become one of the major motivations for dynamic scheduling. Out-of-order execution allows the processors to continue executing instructions while awaiting the completion of a cache miss, thus hiding all or part of the cache miss penalty.
2. As processors became more aggressive in their issue capability and designers were concerned with the performance of difficult-to-schedule code (such as most nonnumeric code), techniques such as register renaming, dynamic scheduling, and speculation became more important.
3. It can achieve high performance without requiring the compiler to target code to a specific pipeline structure, a valuable property in the era of shrink-wrapped mass market software.

3.6 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. Thus exploiting more parallelism requires that we overcome the limitation of control dependence.

Overcoming control dependence is done by speculating on the outcome of branches and executing the program as if our guesses are 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 are always correct; dynamic scheduling only fetches and issues such instructions. Of course, we need mechanisms to handle the situation where the speculation is incorrect. Appendix H 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: (1) dynamic branch prediction to choose which instructions to execute, (2) 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 (3) 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 follows 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.

To extend Tomasulo's algorithm to support speculation, 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 because 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 irrevocable action (such as updating state or taking an exception) until an instruction commits. Therefore, when we add speculation, we need to separate the process of completing execution from instruction commit, because instructions may finish execution considerably before they are ready to commit. Adding this commit phase to the instruction execution sequence requires 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) 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. The ROB therefore 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, all 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.

Figure 3.15 shows the hardware structure of the processor including the ROB. Each entry in the ROB contains four fields: the instruction type, the destination field, the value field, and the ready 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 has register destinations). The destination field supplies the register number (for loads

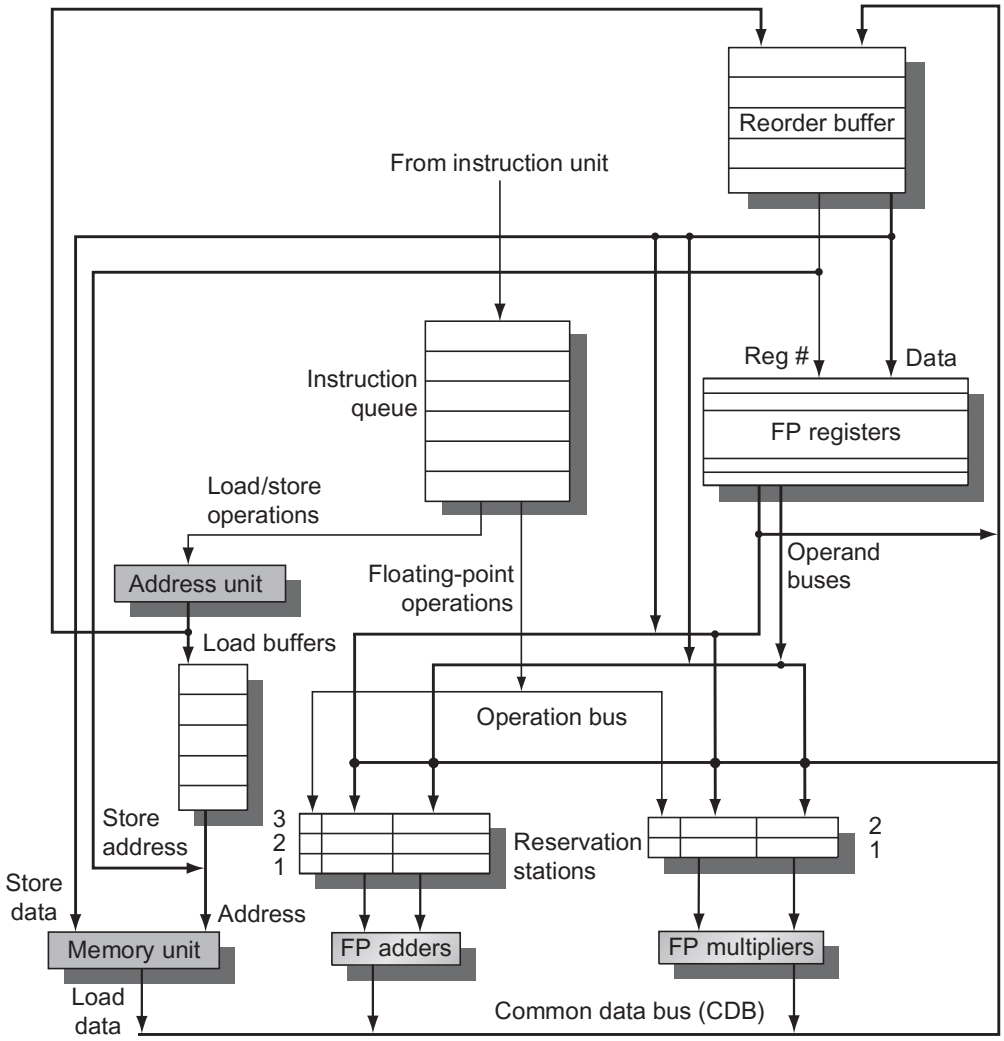

Figure 3.15 The basic structure of a FP unit using Tomasulo's algorithm and extended to handle speculation. Comparing this to Figure 3.10 on page 198, which implemented Tomasulo's algorithm, we can see that the major change is the addition of the ROB and the elimination of the store buffer, whose function is integrated into the ROB. This mechanism can be extended to allow multiple issues per clock by making the CDB wider to allow for multiple completions per clock.

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. Finally, the ready field indicates that the instruction has completed execution, and the value is ready.

The ROB subsumes 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. Because 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 a queue that replaces the ROB to decide 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 are available in either the registers or the ROB. Update the control entries to indicate the buffers are in use. The number of the ROB entry 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 the instruction issue is stalled until both have available entries.
2. *Execute*—If one or more of the operands is not yet available, monitor the CDB 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. Instructions may take multiple clock cycles in this stage, and loads still require two steps in this stage. Stores only need the base register at this step, because execution 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 we assume that this occurs during the Write Result stage of a store; we discuss relaxing this requirement later.
4. *Commit*—This is the final stage of completing an instruction, after which only its result remains. (Some processors call this commit phase "completion" or "graduation.") 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.

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 6 clock cycles, and divide is 12 clock cycles. Using the following code segment, the same one we used to generate Figure 3.12, show what the status tables look like when the `fmul.d` is ready to go to commit.

```
fld     f6,32(x2)
fld     f2,44(x3)
fmul.d  f0,f2,f4
fsub.d  f8,f2,f6
fdiv.d  f0,f0,f6
fadd.d  f6,f8,f2
```

Answer Figure 3.16 shows the result in the three tables. Notice that although the `fsub.d` instruction has completed execution, it does not commit until the `fmul.d` commits. The reservation stations and register status field contain the same basic information that they did for Tomasulo's algorithm (see page 200 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 added the Dest field to the reservation stations. The Dest field designates the ROB entry that is the destination for the result produced by this reservation station entry.

The preceding example illustrates the key important difference between a processor with speculation and a processor with dynamic scheduling. Compare the content of Figure 3.16 with that of Figure 3.12 on page 184, which shows the same code sequence in operation on a processor with Tomasulo's algorithm. The key difference is that, in the preceding example, no instruction after the earliest uncompleted instruction (`fmul.d` in preceding example) is allowed to complete. In contrast, in Figure 3.12 the `fsub.d` and `fadd.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 `fmul.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 from the ROB. Because instruction commit happens in order, this yields a precise exception.

By contrast, in the example using Tomasulo's algorithm, the `fsub.d` and `fadd.d` instructions could both complete before the `fmul.d` raised the exception. The result is that the registers `f8` and `f6` (destinations of the `fsub.d` and

	Reorder buffer				
Entry	Busy	Instruction	State	Destination	Value
1	No	`fld f6,32(x2)`	Commit	`f6`	`Mem[32 + Regs[x2]]`
2	No	`fld f2,44(x3)`	Commit	`f2`	`Mem[44 + Regs[x3]]`
3	Yes	`fmul.d f0,f2,f4`	Write result	`f0`	#2 × `Regs[f4]`
4	Yes	`fsub.d f8,f2,f6`	Write result	`f8`	#2 − #1
5	Yes	`fdiv.d f0,f0,f6`	Execute	`f0`	
6	Yes	`fadd.d f6,f8,f2`	Write result	`f6`	#4 + #2

	Reservation stations							
Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	`fmul.d`	`Mem[44 + Regs[x3]]`	`Regs[f4]`			#3	
Mult2	Yes	`fdiv.d`		`Mem[32 + Regs[x2]]`	#3		#5	

	FP register status									
Field	f0	f1	f2	f3	f4	f5	f6	f7	f8	f10
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	…	Yes	Yes

Figure 3.16 At the time the `fmul.d` is ready to commit, only the two `fld` instructions have committed, although several others have completed execution. The `fmul.d` is at the head of the ROB, and the two `fld` instructions are there only to ease understanding. The `fsub.d` and `fadd.d` instructions will not commit until the `fmul.d` instruction commits, although the results of the instructions are available and can be used as sources for other instructions. The `fdiv.d` is in execution, but has not completed solely because of its longer latency than that of `fmul.d`. The Value column indicates the value being held; the format #X is used to refer to a value field of ROB entry X. Reorder buffers 1 and 2 are actually completed but are shown for informational purposes. We do not show the entries for the load/store queue, but these entries are kept in order.

`fadd.d` instructions) could be overwritten, in which case the interrupt would be imprecise.

Some users and architects have decided that imprecise floating-point exceptions are acceptable in high-performance processors because the program will likely terminate; see Appendix J for further discussion of this topic. Other types

of exceptions, such as page faults, are much more difficult to accommodate if they are imprecise because 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 3.14 in execution:

```
Loop:  fld     f0,0(x1)
       fmul.d  f4,f0,f2
       fsd     f4,0(x1)
       addi    x1,x1,−8
       bne     x1,x2,Loop    //branches if x1≠x2
```

Assume that we have issued all the instructions in the loop twice. Let's also assume that the `fld` and `fmul.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 (`x1` in this example) and the value (`f4` in this example). Because 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 Figure 3.17 shows the result in two tables.

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 the branch `bne` is not taken the first time in Figure 3.17. 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, processors 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, performance is more sensitive to the branch prediction because the impact of a misprediction will be higher. Thus all the aspects of handling branches—prediction accuracy, latency of misprediction detection, and misprediction recovery time—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

	Reorder buffer				
Entry	Busy	Instruction	State	Destination	Value
1	No	`fld f0,0(x1)`	Commit	`f0`	`Mem[0 + Regs[x1]]`
2	No	`fmul.d f4,f0,f2`	Commit	`f4`	#1 × `Regs[f2]`
3	Yes	`fsd f4,0(x1)`	Write result	`0 + Regs[x1]`	#2
4	Yes	`addi x1,x1,−8`	Write result	`x1`	`Regs[x1] − 8`
5	Yes	`bne x1,x2,Loop`	Write result		
6	Yes	`fld f0,0(x1)`	Write result	`f0`	`Mem[#4]`
7	Yes	`fmul.d f4,f0,f2`	Write result	`f4`	#6 × `Regs[f2]`
8	Yes	`fsd f4,0(x1)`	Write result	0 + #4	#7
9	Yes	`addi x1,x1,−8`	Write result	`x1`	#4 − 8
10	Yes	`bne x1,x2,Loop`	Write result		

	FP register status								
Field	f0	f1	f2	f3	f4	F5	f6	F7	f8
Reorder #	6								
Busy	Yes	No	No	No	Yes	No	No	…	No

Figure 3.17 Only the `fld` and `fmul.d` instructions have committed, although all the others have completed execution. Thus no reservation stations are busy and none are shown. The remaining instructions will be committed as quickly as possible. The first two reorder buffers are empty, but are shown for completeness.

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 3.18 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 3.18 with the comparable figure for Tomasulo's algorithm in Figure 3.13 shows that speculation adds significant complications to the control. In addition, remember that branch mispredictions are somewhat more complex.

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 Result (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

Status	Wait until	Action or bookkeeping
Issue all instructions	Reservation station (r) and ROB (b) both available	`if (RegisterStat[rs].Busy)/*in-flight instr. writes rs*/ {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[b].Instruction ← opcode; ROB[b].Dest ← rd; ROB[b].Ready ← no;`
FP operations and stores		`if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/ {h ← RegisterStat[rt].Reorder; if (ROB[h].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].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd;`
Loads		`RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt;`
Stores		`RS[r].A ← imm;`
Execute FP op	`(RS[r].Qj == 0)` and `(RS[r].Qk == 0)`	Compute results—operands are in Vj and Vk
Load step 1	`(RS[r].Qj == 0)` and there are no stores earlier in the queue	`RS[r].A ← RS[r].Vj + RS[r].A;`
Load step 2	Load step 1 done and all stores earlier in ROB have different address	Read from `Mem[RS[r].A]`
Store	`(RS[r].Qj == 0)` and store at queue head	`ROB[h].Address ← RS[r].Vj + RS[r].A;`
Write result all but store	Execution done at *r* and CDB available	`b ← RS[r].Dest; 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* and `(RS[r].Qk == 0)`	`ROB[h].Value ← RS[r].Vk;`
Commit	Instruction is at the head of the ROB (entry h) and `ROB[h].ready == yes`	`d ← ROB[h].Dest; /* register dest, if exists */ if (ROB[h].Instruction==Branch) {if (branch is mispredicted) {clear ROB[h], RegisterStat; fetch branch dest;};} else if (ROB[h].Instruction==Store) {Mem[ROB[h].Destination] ← ROB[h].Value;} else /* put the result in the register destination */ {Regs[d] ← ROB[h].Value;}; ROB[h].Busy ← no; /* free up ROB entry */ /* free up dest register if no one else writing it */ if (RegisterStat[d].Reorder==h) {RegisterStat[d].Busy ← no;};`

Figure 3.18 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, `r` is the reservation station allocated, `b` is the assigned ROB entry, and `h` is the head entry of the ROB. `RS` is the reservation station data structure. The value returned by a reservation station is called the `result`. `Register-Stat` is the register data structure, `Regs` represents the actual registers, and `ROB` is the reorder buffer data structure.

the ROB. This difference ensures that memory is not updated until an instruction is no longer speculative.

Figure 3.18 has one significant simplification for stores, which is unneeded in practice. Figure 3.18 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 3.18, which is already in a small font, would be even longer! Although Figure 3.18 makes this simplification, in our examples, we will allow the store to pass through the Write Result 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, so 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 a Destination field that matches the value of the A field of the load
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 store has written the data. Some speculative processors will actually bypass the value from the store to the load directly when such a RAW hazard occurs. Another approach is to predict potential collisions using a form of value prediction; we consider this in Section 3.9.

Although this explanation of speculative execution has focused on floating point, the techniques easily extend to the integer registers and functional units. Indeed, because such programs tend to have code where the branch behavior is less predictable, speculation may be more useful in integer programs. 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 because less ambitious techniques can probably exploit sufficient ILP within basic blocks when assisted by a compiler.

3.7 Exploiting ILP Using Multiple Issue and Static Scheduling

The techniques of the preceding sections can be used to eliminate data, control stalls, and achieve an ideal CPI of one. To improve performance further, we want 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 the next few sections, is to allow multiple instructions to issue in a clock cycle. Multiple-issue processors come in three major flavors:

1. Statically scheduled superscalar processors
2. VLIW (very long instruction word) processors
3. Dynamically scheduled superscalar processors

The two types of superscalar processors issue varying numbers of instructions per clock and use in-order execution if they are statically scheduled or out-of-order execution if they are dynamically scheduled.

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. VLIW processors are inherently statically scheduled by the compiler. When Intel and HP created the IA-64 architecture, described in Appendix H, they also introduced the name EPIC (explicitly parallel instruction computer) for this architectural style.

Although statically scheduled superscalars issue a varying rather than a fixed number of instructions per clock, they are actually closer in concept to VLIWs because both approaches rely on the compiler to schedule code for the processor. Because of the diminishing advantages of a statically scheduled superscalar as the issue width grows, statically scheduled superscalars are used primarily for narrow issue widths, normally just two instructions. Beyond that width, most designers choose to implement either a VLIW or a dynamically scheduled superscalar. Because of the similarities in hardware and required compiler technology, we focus on VLIWs in this section, and we will see them again in Chapter 7. The insights of this section are easily extrapolated to a statically scheduled superscalar.

Figure 3.19 summarizes the basic approaches to multiple issue and their distinguishing characteristics and shows processors that use each approach.

The Basic VLIW Approach

VLIWs use multiple, independent functional units. Rather than attempting to issue multiple, independent instructions to the units, a VLIW packages the multiple operations into one very long instruction or requires that the instructions in the

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the Cortex-A53
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

Figure 3.19 The five primary approaches in use for multiple-issue processors and the primary characteristics that distinguish them. This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. Appendix H focuses on compiler-based approaches. The EPIC approach, as embodied in the IA-64 architecture, extends many of the concepts of the early VLIW approaches, providing a blend of static and dynamic approaches.

issue packet satisfy the same constraints. Because there is no fundamental difference in the two approaches, we will just assume that multiple operations are placed in one instruction, as in the original VLIW approach.

Because the advantage of a VLIW increases as the maximum issue rate grows, we focus on a wider issue processor. Indeed, for simple two-issue processors, the overhead of a superscalar is probably minimal. Many designers would probably argue that a four-issue processor has manageable overhead, but as we will see later in this chapter, the growth in overhead is a major factor limiting wider issue processors.

Let's consider a VLIW processor with instructions that contain five operations, including one integer operation (which could also be a branch), two floating-point operations, and two memory references. The instruction would have a set of fields for each functional unit—perhaps 16–24 bits per unit, yielding an instruction length of between 80 and 120 bits. By comparison, the Intel Itanium 1 and 2 contain six operations per instruction packet (i.e., they allow concurrent issue of two three-instruction bundles, as Appendix H describes).

To keep the functional units busy, there must be enough parallelism in a code sequence to fill the available operation slots. This parallelism is uncovered by unrolling loops and scheduling the code within the single larger loop body. If the unrolling generates straight-line code, then *local scheduling* techniques, which operate on a single basic block, can be used. If finding and exploiting the parallelism require scheduling code across branches, a substantially more complex *global*

scheduling algorithm must be used. Global scheduling algorithms are not only more complex in structure, but they also must deal with significantly more complicated trade-offs in optimization, because moving code across branches is expensive.

In Appendix H, we discuss *trace scheduling*, one of these global scheduling techniques developed specifically for VLIWs; we will also explore special hardware support that allows some conditional branches to be eliminated, extending the usefulness of local scheduling and enhancing the performance of global scheduling.

For now, we will rely on loop unrolling to generate long, straight-line code sequences so that we can use local scheduling to build up VLIW instructions and focus on how well these processors operate.

Example Suppose we have a VLIW that could issue two memory references, two FP operations, and one integer operation or branch in every clock cycle. Show an unrolled version of the loop `x[i] = x[i] + s` (see page 158 for the RISC-V code) for such a processor. Unroll as many times as necessary to eliminate any stalls.

Answer Figure 3.20 shows the code. The loop has been unrolled to make seven copies of the body, which eliminates all stalls (i.e., completely empty issue cycles), and runs in 9 cycles for the unrolled and scheduled loop. This code yields a running rate of seven results in 9 cycles, or 1.29 cycles per result, nearly twice as fast as the two-issue superscalar of Section 3.2 that used unrolled and scheduled code.

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
`fld f0,0(x1)`	`fld f6,-8(x1)`			
`fld f10,-16(x1)`	`fld f14,-24(x1)`			
`fld f18,-32(x1)`	`fld f22,-40(x1)`	`fadd.d f4,f0,f2`	`fadd.d f8,f6,f2`	
`fld f26,-48(x1)`		`fadd.d f12,f0,f2`	`fadd.d f16,f14,f2`	
		`fadd.d f20,f18,f2`	`fadd.d f24,f22,f2`	
`fsd f4,0(x1)`	`fsd f8,-8(x1)`	`fadd.d f28,f26,f24`		
`fsd f12,-16(x1)`	`fsd f16,-24(x1)`			`addi x1,x1,-56`
`fsd f20,24(x1)`	`fsd f24,16(x1)`			
`fsd f28,8(x1)`				`bne x1,x2,Loop`

Figure 3.20 VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes 9 cycles assuming correct branch prediction. The issue rate is 23 operations in 9 clock cycles, or 2.5 operations per cycle. The efficiency, the percentage of available slots that contained an operation, is about 60%. To achieve this issue rate requires a larger number of registers than RISC-V would normally use in this loop. The preceding VLIW code sequence requires at least eight FP registers, whereas the same code sequence for the base RISC-V processor can use as few as two FP registers or as many as five when unrolled and scheduled.

For the original VLIW model, there were both technical and logistical problems that made the approach less efficient. The technical problems were the increase in code size and the limitations of the lockstep operation. Two different elements combine to increase code size substantially for a VLIW. First, generating enough operations in a straight-line code fragment requires ambitiously unrolling loops (as in earlier examples), thereby increasing code size. Second, whenever instructions are not full, the unused functional units translate to wasted bits in the instruction encoding. In Appendix H, we examine software scheduling approaches, such as software pipelining, that can achieve the benefits of unrolling without as much code expansion.

To combat this code size increase, clever encodings are sometimes used. For example, there may be only one large immediate field for use by any functional unit. Another technique is to compress the instructions in main memory and expand them when they are read into the cache or are decoded. In Appendix H, we show other techniques, as well as document the significant code expansion seen in IA-64.

Early VLIWs operated in lockstep; there was no hazard-detection hardware at all. This structure dictated that a stall in any functional unit pipeline must cause the entire processor to stall because all the functional units had to be kept synchronized. Although a compiler might have been able to schedule the deterministic functional units to prevent stalls, predicting which data accesses would encounter a cache stall and scheduling them were very difficult to do. Thus caches needed to be blocking and causing *all* the functional units to stall. As the issue rate and number of memory references became large, this synchronization restriction became unacceptable. In more recent processors, the functional units operate more independently, and the compiler is used to avoid hazards at issue time, while hardware checks allow for unsynchronized execution once instructions are issued.

Binary code compatibility has also been a major logistical problem for general-purpose VLIWs or those that run third-party software. In a strict VLIW approach, the code sequence makes use of both the instruction set definition and the detailed pipeline structure, including both functional units and their latencies. Thus different numbers of functional units and unit latencies require different versions of the code. This requirement makes migrating between successive implementations, or between implementations with different issue widths, more difficult than it is for a superscalar design. Of course, obtaining improved performance from a new superscalar design may require recompilation. Nonetheless, the ability to run old binary files is a practical advantage for the superscalar approach. In the domain-specific architectures, which we examine in Chapter 7, this problem is not serious because applications are written specifically for an architectural configuration.

The EPIC approach, of which the IA-64 architecture is the primary example, provides solutions to many of the problems encountered in early general-purpose VLIW designs, including extensions for more aggressive software speculation and methods to overcome the limitation of hardware dependence while preserving binary compatibility.

The major challenge for all multiple-issue processors is to try to exploit large amounts of ILP. When the parallelism comes from unrolling simple loops in FP

programs, the original loop probably could have been run efficiently on a vector processor (described in the next chapter). It is not clear that a multiple-issue processor is preferred over a vector processor for such applications; the costs are similar, and the vector processor is typically the same speed or faster. The potential advantages of a multiple-issue processor versus a vector processor are the former's ability to extract some parallelism from less structured code and to easily cache all forms of data. For these reasons, multiple-issue approaches have become the primary method for taking advantage of instruction-level parallelism, and vectors have become primarily an extension to these processors.

3.8 Exploiting ILP Using Dynamic Scheduling, Multiple Issue, and Speculation

So far we have seen how the individual mechanisms of dynamic scheduling, multiple issue, and speculation work. In this section, we put all three together, which yields a microarchitecture quite similar to those in modern microprocessors. For simplicity we consider only an issue rate of two instructions per clock, but the concepts are no different from modern processors that issue three or more instructions per clock.

Let's assume we want to extend Tomasulo's algorithm to support multiple-issue superscalar pipeline with separate integer, load/store, and floating-point units (both FP multiply and FP add), each of which can initiate an operation on every clock. We do not want to issue instructions to the reservation stations out of order because this could lead to a violation of the program semantics. To gain the full advantage of dynamic scheduling, we will allow the pipeline to issue any combination of two instructions in a clock, using the scheduling hardware to actually assign operations to the integer and floating-point unit. 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, as well as incorporating speculative execution. As Figure 3.21 shows, the basic organization is similar to that of a processor with speculation with one issue per clock, except that the issue and completion logic must be enhanced to allow multiple instructions to be processed per clock.

Issuing multiple instructions per clock in a dynamically scheduled processor (with or without speculation) is very complex for the simple reason that the multiple instructions may depend on one another. Because of this, the tables must be updated for the instructions in parallel; otherwise, the tables will be incorrect or the dependence may be lost.

Two different approaches have been used to issue multiple instructions per clock in a dynamically scheduled processor, and both rely on the observation that the 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; this approach cannot be easily extended to handle four instructions per clock, unfortunately.

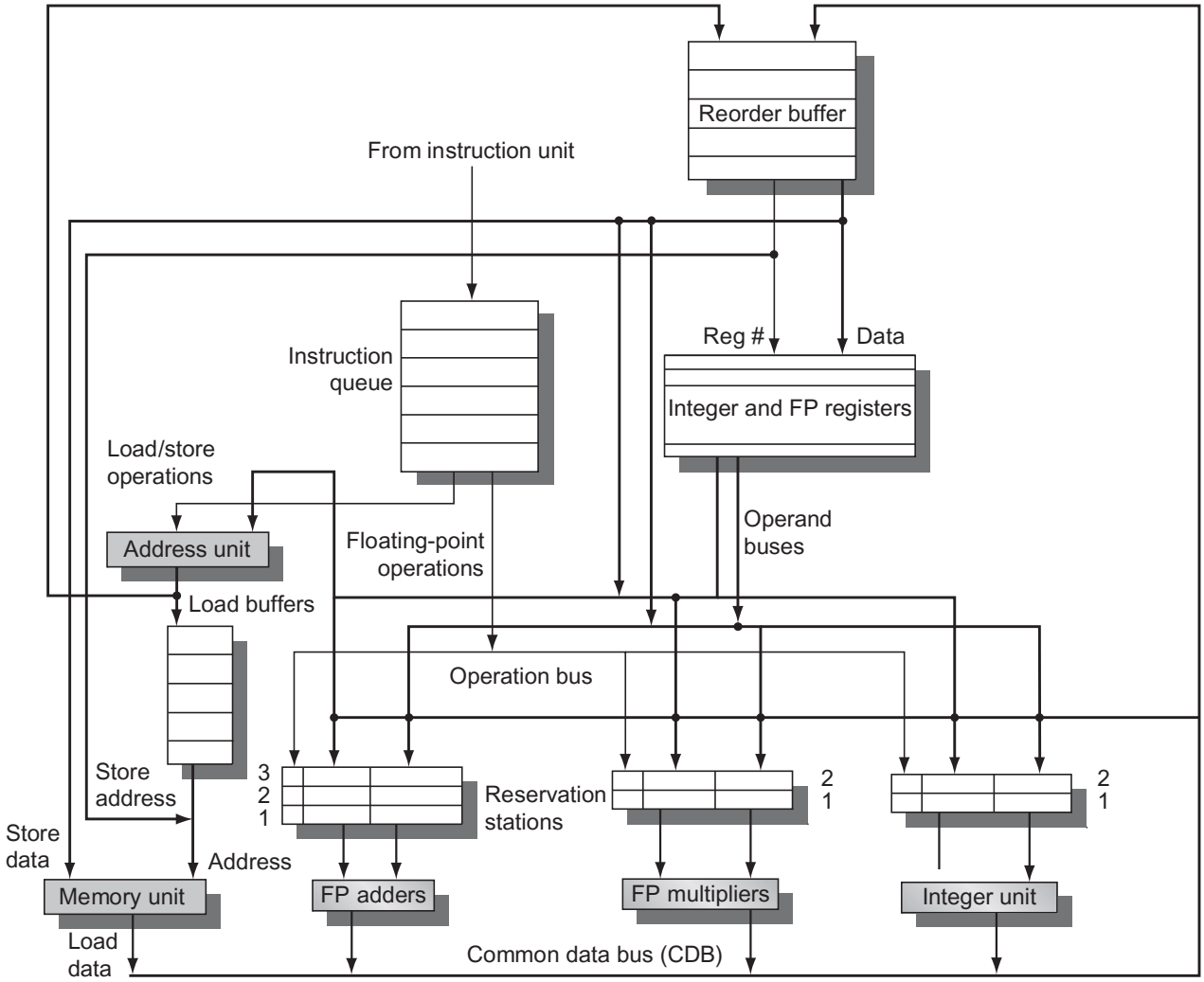

Figure 3.21 The basic organization of a multiple issue processor with speculation. In this case, the organization could allow a FP multiply, FP add, integer, and load/store to all issues simultaneously (assuming one issue per clock per functional unit). Note that several datapaths must be widened to support multiple issues: the CDB, the operand buses, and, critically, the instruction issue logic, which is not shown in this figure. The last is a difficult problem, as we discuss in the text.

A second alternative is to build the logic necessary to handle two or more instructions at once, including any possible dependences between the instructions. Modern superscalar processors that issue four or more instructions per clock may include both approaches: They both pipeline and widen the issue logic. A key observation is that we cannot simply pipeline away the problem. By making instruction issues take multiple clocks because new instructions are issuing every clock cycle, we must be able to assign the reservation station and to update the pipeline tables so that a dependent instruction issuing on the next clock can use the updated information.

This issue step is one of the most fundamental bottlenecks in dynamically scheduled superscalars. To illustrate the complexity of this process, Figure 3.22 shows the issue logic for one case: issuing a load followed by a dependent FP operation. The logic is based on that in Figure 3.18 on page 197, but represents only one case. In a modern superscalar, every possible combination of dependent instructions that is allowed to issue in the same clock cycle must be considered. Because the number of possibilities climbs as the square of the number of instructions that can be issued in a clock, the issue step is a likely bottleneck for attempts to go beyond four instructions per clock.

We can generalize the detail of Figure 3.22 to describe the basic strategy for updating the issue logic and the reservation tables in a dynamically scheduled superscalar with up to n issues per clock as follows:

1. Assign a reservation station and a reorder buffer for *every* instruction that *might* be issued in the next issue bundle. This assignment can be done before the instruction types are known simply by preallocating the reorder buffer entries sequentially to the instructions in the packet using n available reorder buffer entries and by ensuring that enough reservation stations are available to issue the whole bundle, independent of what it contains. By limiting the number of instructions of a given class (say, one FP, one integer, one load, one store), the necessary reservation stations can be preallocated. Should sufficient reservation stations not be available (such as when the next few instructions in the program are all of one instruction type), the bundle is broken, and only a subset of the instructions, in the original program order, is issued. The remainder of the instructions in the bundle can be placed in the next bundle for potential issue.
2. Analyze all the dependences among the instructions in the issue bundle.
3. If an instruction in the bundle depends on an earlier instruction in the bundle, use the assigned reorder buffer number to update the reservation table for the dependent instruction. Otherwise, use the existing reservation table and reorder buffer information to update the reservation table entries for the issuing instruction.

Of course, what makes the preceding very complicated is that it is all done in parallel in a single clock cycle!

At the back-end of the pipeline, we must be able to complete and commit multiple instructions per clock. These steps are somewhat easier than the issue problems because multiple instructions that can actually commit in the same clock cycle must have already dealt with and resolved any dependences. As we will see, designers have figured out how to handle this complexity: The Intel i7, which we examine in Section 3.12, uses essentially the scheme we have described for speculative multiple issue, including a large number of reservation stations, a reorder buffer, and a load and store buffer that is also used to handle nonblocking cache misses.

From a performance viewpoint, we can show how the concepts fit together with an example.

Action or bookkeeping	Comments
`if (RegisterStat[rs1].Busy)/*in-flight instr. writes rs*/ {h ← RegisterStat[rs1].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[x1].Vj ← ROB[h].Value; RS[x1].Qj ← 0;} else {RS[x1].Qj ← h;} /* wait for instruction */ } else {RS[x1].Vj ← Regs[rs]; RS[x1].Qj ← 0;}; RS[x1].Busy ← yes; RS[x1].Dest ← b1; ROB[b1].Instruction ← Load; ROB[b1].Dest ← rd1; ROB[b1].Ready ← no; RS[r].A ← imm1; RegisterStat[rt1].Reorder ← b1; RegisterStat[rt1].Busy ← yes; ROB[b1].Dest ← rt1;`	Updating the reservation tables for the load instruction, which has a single source operand. Because this is the first instruction in this issue bundle, it looks no different than what would normally happen for a load.
`RS[x2].Qj ← b1;} /* wait for load instruction */`	Because we know that the first operand of the FP operation is from the load, this step simply updates the reservation station to point to the load. Notice that the dependence must be analyzed on the fly and the ROB entries must be allocated during this issue step so that the reservation tables can be correctly updated.
`if (RegisterStat[rt2].Busy) /*in-flight instr writes rt*/ {h ← RegisterStat[rt2].Reorder; if (ROB[h].Ready)/* Instr completed already */ {RS[x2].Vk ← ROB[h].Value; RS[x2].Qk ← 0;} else {RS[x2].Qk ← h;} /* wait for instruction */ } else {RS[x2].Vk ← Regs[rt2]; RS[x2].Qk ← 0;}; RegisterStat[rd2].Reorder ← b2; RegisterStat[rd2].Busy ← yes; ROB[b2].Dest ← rd2;`	Because we assumed that the second operand of the FP instruction was from a prior issue bundle, this step looks like it would in the single-issue case. Of course, if this instruction were dependent on something in the same issue bundle, the tables would need to be updated using the assigned reservation buffer.
`RS[x2].Busy ← yes; RS[x2].Dest ← b2; ROB[b2].Instruction ← FP operation; ROB[b2].Dest ← rd2; ROB[b2].Ready ← no;`	This section simply updates the tables for the FP operation and is independent of the load. Of course, if further instructions in this issue bundle depended on the FP operation (as could happen with a four-issue superscalar), the updates to the reservation tables for those instructions would be effected by this instruction.

Figure 3.22 The issue steps for a pair of dependent instructions (called 1 and 2), where instruction 1 is FP load and instruction 2 is an FP operation whose first operand is the result of the load instruction; `x1` and `x2` are the assigned reservation stations for the instructions; and `b1` and `b2` are the assigned reorder buffer entries. For the issuing instructions, `rd1` and `rd2` are the destinations; `rs1`, `rs2`, and `rt2` are the sources (the load has only one source); `x1` and `x2` are the reservation stations allocated; and `b1` and `b2` are the assigned ROB entries. `RS` is the reservation station data structure. `RegisterStat` is the register data structure, `Regs` represents the actual registers, and `ROB` is the reorder buffer data structure. Notice that we need to have assigned reorder buffer entries for this logic to operate properly, and recall that all these updates happen in a single clock cycle in parallel, not sequentially.

Example Consider the execution of the following loop, which increments each element of an integer array, on a two-issue processor, once without speculation and once with speculation:

```
Loop:  ld    x2,0(x1)      //x2=array element
       addi  x2,x2,1       //increment x2
       sd    x2,0(x1)      //store result
       addi  x1,x1,8       //increment pointer
       bne   x2,x3,Loop    //branch if not last
```

Assume that there are separate integer functional units for effective address calculation, for ALU operations, and for branch condition evaluation. Create a table for the first three iterations of this loop for both processors. Assume that up to two instructions of any type can commit per clock.

Answer Figures 3.23 and 3.24 show the performance for a two-issue, dynamically scheduled processor, without and with speculation. In this case, where a branch

Iteration number	Instructions	Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB at clock cycle number	Comment
1	`ld x2,0(x1)`	1	2	3	4	First issue
1	`addi x2,x2,1`	1	5		6	Wait for `ld`
1	`sd x2,0(x1)`	2	3	7		Wait for `addi`
1	`addi x1,x1,8`	2	3		4	Execute directly
1	`bne x2,x3,Loop`	3	7			Wait for `addi`
2	`ld x2,0(x1)`	4	8	9	10	Wait for `bne`
2	`addi x2,x2,1`	4	11		12	Wait for `ld`
2	`sd x2,0(x1)`	5	9	13		Wait for `addi`
2	`addi x1,x1,8`	5	8		9	Wait for `bne`
2	`bne x2,x3,Loop`	6	13			Wait for `addi`
3	`ld x2,0(x1)`	7	14	15	16	Wait for `bne`
3	`addi x2,x2,1`	7	17		18	Wait for `ld`
3	`sd x2,0(x1)`	8	15	19		Wait for `addi`
3	`addi x1,x1,8`	8	14		15	Wait for `bne`
3	`bne x2,x3,Loop`	9	19			Wait for `addi`

Figure 3.23 The time of issue, execution, and writing result for a dual-issue version of our pipeline *without* speculation. Note that the `ld` 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. Figure 3.24 shows this example with speculation.

Iteration number	Instructions	Issues at clock number	Executes at clock number	Read access at clock number	Write CDB at clock number	Commits at clock number	Comment
1	`ld x2,0(x1)`	1	2	3	4	5	First issue
1	`addi x2,x2,1`	1	5		6	7	Wait for `ld`
1	`sd x2,0(x1)`	2	3			7	Wait for `addi`
1	`addi x1,x1,8`	2	3		4	8	Commit in order
1	`bne x2,x3,Loop`	3	7			8	Wait for `addi`
2	`ld x2,0(x1)`	4	5	6	7	9	No execute delay
2	`addi x2,x2,1`	4	8		9	10	Wait for `ld`
2	`sd x2,0(x1)`	5	6			10	Wait for `addi`
2	`addi x1,x1,8`	5	6		7	11	Commit in order
2	`bne x2,x3,Loop`	6	10			11	Wait for `addi`
3	`ld x2,0(x1)`	7	8	9	10	12	Earliest possible
3	`addi x2,x2,1`	7	11		12	13	Wait for `ld`
3	`sd x2,0(x1)`	8	9			13	Wait for `addi`
3	`addi x1,x1,8`	8	9		10	14	Executes earlier
3	`bne x2,x3,Loop`	9	13			14	Wait for `addi`

Figure 3.24 The time of issue, execution, and writing result for a dual-issue version of our pipeline *with* speculation. Note that the `ld` following the `bne` can start execution early because it is speculative.

can be a critical performance limiter, speculation helps significantly. The third branch in the speculative processor executes in clock cycle 13, whereas it executes in clock cycle 19 on the 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 1 clock per iteration.

This 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 does not improve performance; in fact, it typically harms performance and, as we shall see, dramatically lowers energy efficiency.

3.9 Advanced Techniques for Instruction Delivery and Speculation

In a high-performance pipeline, especially one with multiple issues, 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. We look at methods for increasing instruction delivery bandwidth first. We then turn to a set of key issues in implementing advanced speculation techniques, including the use of register renaming versus reorder buffers, the aggressiveness of speculation, and a technique called *value prediction*, which attempts to predict the result of a computation and which could further enhance ILP.

Increasing Instruction Fetch Bandwidth

A multiple-issue processor will require that the average number of instructions fetched every clock cycle be at least as large as the average throughput. Of course, fetching these instructions requires wide enough paths to the instruction cache, but the most difficult aspect is handling branches. In this section, we look at two methods for dealing with branches and then discuss how modern processors integrate the instruction prediction and prefetch functions.

Branch-Target Buffers

To reduce the branch penalty for our simple five-stage pipeline, as well as for deeper pipelines, we must know whether the as-yet-undecoded instruction is a branch and, if so, what the next program counter (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.* Figure 3.25 shows a branch-target buffer.

Because a branch-target buffer predicts 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. If the PC of the fetched instruction matches an address in the prediction buffer, then the corresponding predicted PC is used as the next PC. The hardware for this branch-target buffer is essentially identical to the hardware for a cache.

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 predictive entry must be matched to this instruction because the predicted PC will be sent

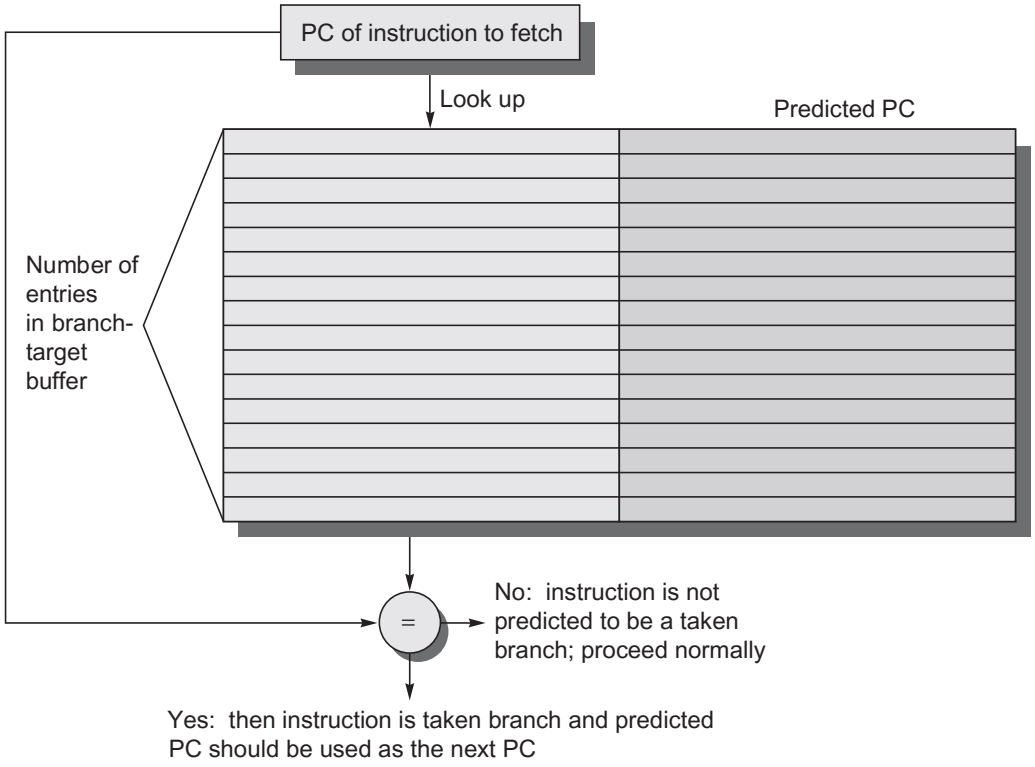

Figure 3.25 A branch-target buffer. The PC of the instruction being fetched is matched against a set of instruction addresses stored in the first column; these represent the addresses of known branches. If the PC matches one of these entries, then the instruction being fetched is a taken branch, and 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.

out before it is known whether this instruction is even a branch. If the processor 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 worse performance. We need to store only the predicted-taken branches in the branch-target buffer because an untaken branch should simply fetch the next sequential instruction, as if it were not a branch.

Figure 3.26 shows the steps when using a branch-target buffer for a simple five-stage pipeline. As we can see in this figure, there will be no branch delay if a branch-prediction entry is found in the buffer and the prediction is correct. Otherwise, there will be a penalty of at least two clock cycles. Dealing with the mispredictions and misses is a significant challenge because we typically will have to halt instruction fetch while we rewrite the buffer entry. Thus we want to make this process fast to minimize the penalty.

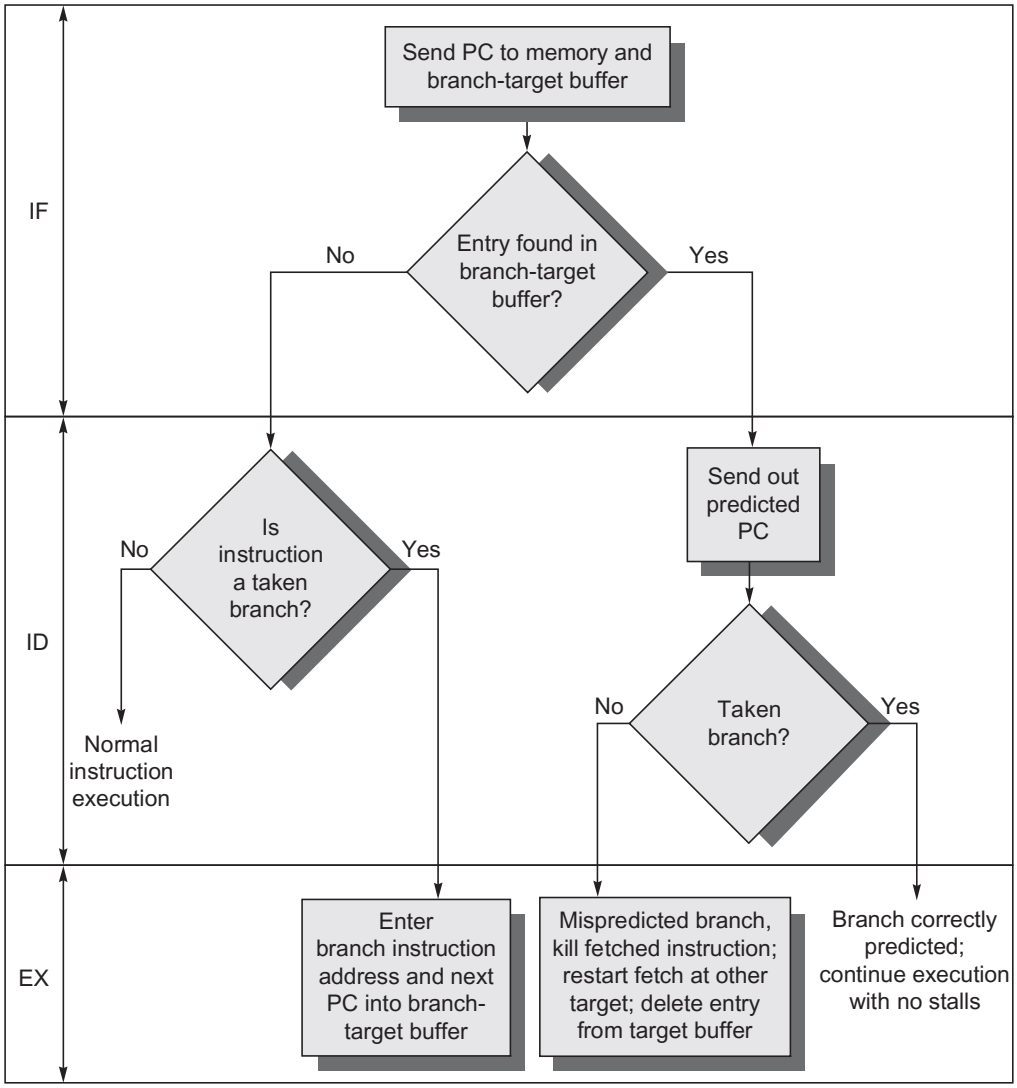

Figure 3.26 **The steps involved in handling an instruction with a branch-target buffer.**

To evaluate how well a branch-target buffer works, we first must determine the penalties in all possible cases. Figure 3.27 contains this information for a simple five-stage pipeline.

Example Determine the total branch penalty for a branch-target buffer assuming the penalty cycles for individual mispredictions in Figure 3.27. 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).

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.

$$\begin{aligned}\text{Probability (branch in buffer, but actually not taken)} &= \text{Percent buffer hit rate} \\ &\quad \times \text{Percent incorrect predictions} \\ &= 90\% \times 10\% = 0.09 \\ \text{Probability (branch not in buffer, but actually taken)} &= 10\% \\ \text{Branch penalty} &= (0.09 + 0.10) \times 2 \\ \text{Branch penalty} &= 0.38\end{aligned}$$

The improvement from dynamic branch prediction will grow as the pipeline length, and thus the branch delay grows; in addition, better predictors will yield a greater performance advantage. Modern high-performance processors have branch misprediction delays on the order of 15 clock cycles; clearly, accurate prediction is critical!

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 0-cycle unconditional branches and sometimes 0-cycle conditional branches. As we will see, the Cortex A-53 uses a single-entry branch target cache that stores the predicted target instructions.

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

Instruction in buffer	Prediction	Actual branch	Penalty cycles
Yes	Taken	Taken	0
Yes	Taken	Not taken	2
No		Taken	2
No		Not taken	0

Figure 3.27 Penalties for all possible combinations of whether the branch is in the buffer and what it actually does, assuming we store only taken branches in the buffer. There is no branch penalty if everything is correctly predicted and the branch is found in the target buffer. If the branch is not correctly predicted, the penalty is equal to 1 clock cycle to update the buffer with the correct information (during which an instruction cannot be fetched) and 1 clock cycle, if needed, to restart fetching the next correct instruction for the branch. If the branch is not found and taken, a 2-cycle penalty is encountered, during which time the buffer is updated.

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 maximum benefit. In some cases, it may be possible to eliminate the cost of a conditional branch.

Specialized Branch Predictors: Predicting Procedure Returns, Indirect Jumps, and Loop Branches

As we try to increase the opportunity and accuracy of speculation, we face the challenge of predicting indirect jumps, that is, jumps whose destination address varies at runtime. High-level language programs will generate such jumps for indirect procedure calls, select or case statements, and FORTRAN-computed gotos, although many indirect jumps simply come from procedure returns. For example, for the SPEC95 benchmarks, procedure returns account for more than 15% of the branches and the vast majority of the indirect jumps on average. For object-oriented languages such as 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. For example, in SPEC CPU95, an aggressive branch predictor achieves an accuracy of less than 60% for such return branches. To overcome this problem, some designs use a small buffer of return addresses operating as a stack. 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 3.28 shows the performance of such a return buffer with 0–16 elements for a number of the SPEC CPU95 benchmarks. We will use a similar return predictor when we examine the studies of ILP in Section 3.10. Both the Intel Core processors and the AMD Phenom processors have return address predictors.

In large server applications, indirect jumps also occur for various function calls and control transfers. Predicting the targets of such branches is not as simple as in a procedure return. Some processors have opted to add specialized predictors for all indirect jumps, whereas others rely on a branch target buffer.

Although a simple predictor like gshare does a good job of predicting many conditional branches, it is not tailored to predicting loop branches, especially for long running loops. As we observed earlier, the Intel Core i7 920 used a specialized loop branch predictor. With the emergence of tagged hybrid predictors, which are as good at predicting loop branches, some recent designers have opted to put the resources into larger tagged hybrid predictors rather than a separate loop branch predictor.

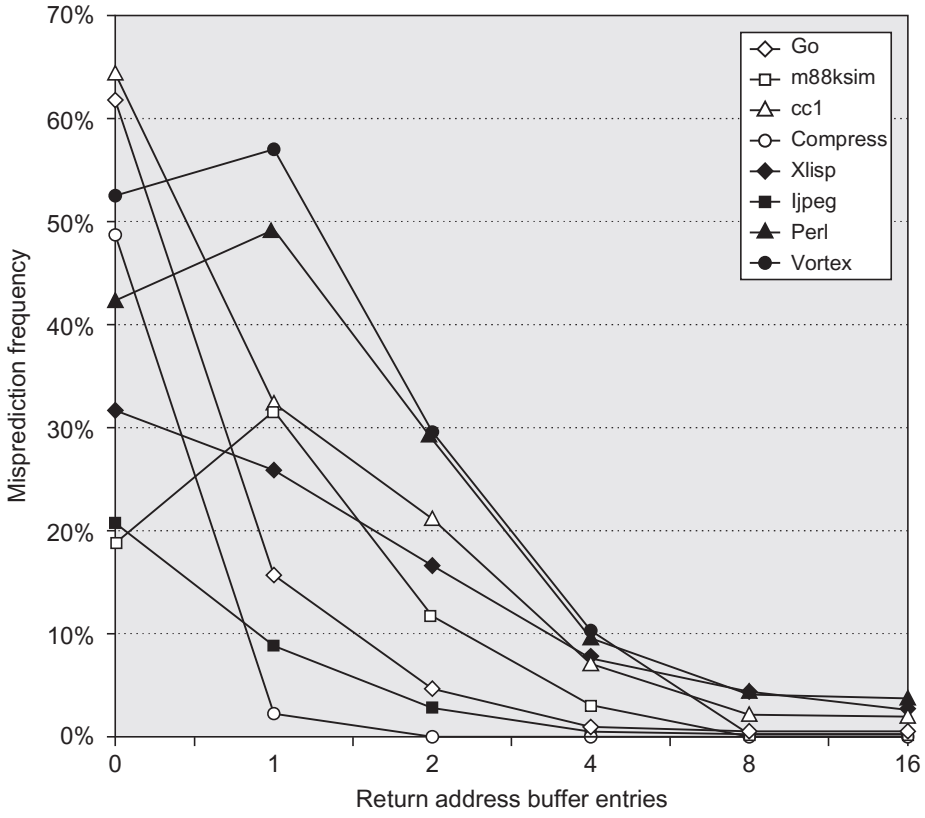

Figure 3.28 Prediction accuracy for a return address buffer operated as a stack on a number of SPEC CPU95 benchmarks. The accuracy is the fraction of return addresses predicted correctly. A buffer of 0 entries implies that the standard branch prediction is used. Because call depths are typically not large, with some exceptions, a modest buffer works well. These data come from Skadron et al. (1999) and use a fix-up mechanism to prevent corruption of the cached return addresses.

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 pipe 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 as 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 2 for a 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.

Virtually all high-end processors now use a separate instruction fetch unit connected to the rest of the pipeline by a buffer containing pending instructions.

Speculation: Implementation Issues and Extensions

In this section, we explore five issues that involve the design trade-offs and challenges in multiple-issue and speculation, starting with the use of register renaming, the approach that is sometimes used instead of a reorder buffer. We then discuss one important possible extension to speculation on control flow: an idea called value prediction.

Speculation Support: Register Renaming Versus Reorder Buffers

One alternative to the use of a reorder buffer (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 and extends it. In Tomasulo's algorithm, the values of the *architecturally visible registers* (x0, . . . r31 and 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 most of the function of the ROB and the reservation stations; only a queue to ensure that instructions complete in order is needed. 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, a function performed by the register status table in Tomasulo's algorithm. When an instruction commits, the renaming table is permanently updated to indicate that a physical register corresponds to the actual architectural register, thus effectively finalizing the update to the processor state. Although an ROB is not necessary with register renaming, the hardware must still track instructions in a queue-like structure and update the renaming table in strict order.

An advantage of the renaming approach versus the ROB approach is that instruction commit is slightly simplified because it requires only two simple actions: (1) record that the mapping between an architectural register number and physical register number is no longer speculative, and (2) 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 entry is freed up when the corresponding instruction commits.

With register renaming, deallocating registers is more complex because 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 outstanding uses of the physical register. 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.

Alternatively, the processor can simply wait until another instruction that writes the same architectural register commits. At that point, there can be no further uses of the older value outstanding. Although this method may tie up a physical register slightly longer than necessary, it is easy to implement and is used in most recent superscalars.

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 reside. To understand how this capability is provided, assume the processor does not issue instructions for some period of time. 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.

Both register renaming and reorder buffers continue to be used in high-end processors, which now feature the ability to have as many as 100 or more instructions (including loads and stores waiting on the cache) in flight. Whether renaming or a reorder buffer is used, the key complexity bottleneck for a dynamically scheduled superscalar remains issuing bundles of instructions with dependences within the bundle. In particular, dependent instructions in an issue bundle must be issued with the assigned virtual registers of the instructions on which they depend. A strategy for instruction issue with register renaming similar to that used for multiple issue with reorder buffers (see page 205) can be deployed, as follows:

1. The issue logic reserves enough physical registers for the entire issue bundle (say, four registers for a four-instruction bundle with at most one register result per instruction).
2. The issue logic determines what dependences exist within the bundle. If a dependence does not exist within the bundle, the register renaming structure is used to determine the physical register that holds, or will hold, the result on which instruction depends. When no dependence exists within the bundle, the result is from an earlier issue bundle, and the register renaming table will have the correct register number.
3. If an instruction depends on an instruction that is earlier in the bundle, then the pre-reserved physical register in which the result will be placed is used to update the information for the issuing instruction.

Note that just as in the reorder buffer case, the issue logic must both determine dependences within the bundle and update the renaming tables in a single clock, and as before, the complexity of doing this for a larger number of instructions per clock becomes a chief limitation in the issue width.

The Challenge of More Issues per Clock

Without speculation, there is little motivation to try to increase the issue rate beyond two, three, or possibly four issues per clock because resolving branches would limit the average issue rate to a smaller number. Once a processor includes accurate branch prediction and speculation, we might conclude that increasing the issue rate would be attractive. Duplicating the functional units is straightforward assuming silicon capacity and power; the real complications arise in the issue step and correspondingly in the commit step. The Commit step is the dual of the issue step, and the requirements are similar, so let's take a look at what has to happen for a six-issue processor using register renaming.

Figure 3.29 shows a six-instruction code sequence and what the issue step must do. Remember that this must all occur in a single clock cycle, if the processor is to maintain a peak rate of six issues per clock! All the dependences must be detected,

Instr. #	Instruction	Physical register assigned or destination	Instruction with physical register numbers	Rename map changes
1	`add x1,x2,x3`	p32	`add p32,p2,p3`	**x1-> p32**
2	`sub x1,x1,x2`	p33	`sub p33,p32,p2`	**x1->p33**
3	`add x2,x1,x2`	p34	`add p34,p33,x2`	**x2->p34**
4	`sub x1,x3,x2`	p35	`sub p35,p3,p34`	**x1->p35**
5	`add x1,x1,x2`	p36	`add p36,p35,p34`	**x1->p36**
6	`sub x1,x3,x1`	p37	`sub p37,p3,p36`	**x1->p37**

Figure 3.29 An example of six instructions to be issued in the same clock cycle and what has to happen. The instructions are shown in program order: 1–6; they are, however, issued in 1 clock cycle! The notation `pi` is used to refer to a physical register; the contents of that register at any point is determined by the renaming map. For simplicity, we assume that the physical registers holding the architectural registers `x1`, `x2`, and `x3` are initially `p1`, `p2`, and `p3` (they could be any physical register). The instructions are issued with physical register numbers, as shown in column four. The rename map, which appears in the last column, shows how the map would change if the instructions were issued sequentially. The difficulty is that all this renaming and replacement of architectural registers by physical renaming registers happens effectively in 1 cycle, not sequentially. The issue logic must find all the dependences and "rewrite" the instruction in parallel.

the physical registers must be assigned, and the instructions must be rewritten using the physical register numbers: in one clock. This example makes it clear why issue rates have grown from 3–4 to only 4–8 in the past 20 years. The complexity of the analysis required during the issue cycle grows as the square of the issue width, and a new processor is typically targeted to have a higher clock rate than in the last generation! Because register renaming and the reorder buffer approaches are duals, the same complexities arise independent of the implementation scheme.

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. Speculation is not free. It takes time and energy, and the recovery of incorrect speculation further reduces performance. In addition, to support the higher instruction execution rate needed to benefit from speculation, the processor must have additional resources, which take silicon area and power. Finally, if speculation causes an exceptional event to occur, such as a cache or translation lookaside buffer (TLB) miss, the potential for significant performance loss increases, if that event would not have occurred without speculation.

To maintain most 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.

In the 1990s the potential downsides of speculation were less obvious. As processors have evolved, the real costs of speculation have become more apparent, and the limitations of wider issue and speculation have been obvious. We return to this issue shortly.

Speculating Through Multiple Branches

In the examples we have considered in this chapter, 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: (1) a very high branch frequency, (2) significant clustering of branches, and (3) 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. As of 2017, no processor has yet combined full speculation with resolving multiple branches per cycle, and it is unlikely that the costs of doing so would be justified in terms of performance versus complexity and power.

Speculation and the Challenge of Energy Efficiency

What is the impact of speculation on energy efficiency? At first glance, one might argue that using speculation always decreases energy efficiency because whenever speculation is wrong, it consumes excess energy in two ways:

1. Instructions that are speculated and whose results are not needed generate excess work for the processor, wasting energy.
2. Undoing the speculation and restoring the state of the processor to continue execution at the appropriate address consumes additional energy that would not be needed without speculation.

Certainly, speculation will raise the power consumption, and if we could control speculation, it would be possible to measure the cost (or at least the dynamic power cost). But, if speculation lowers the execution time by more than it increases the average power consumption, then the total energy consumed may be less.

Thus, to understand the impact of speculation on energy efficiency, we need to look at how often speculation is leading to unnecessary work. If a significant number of unneeded instructions is executed, it is unlikely that speculation will

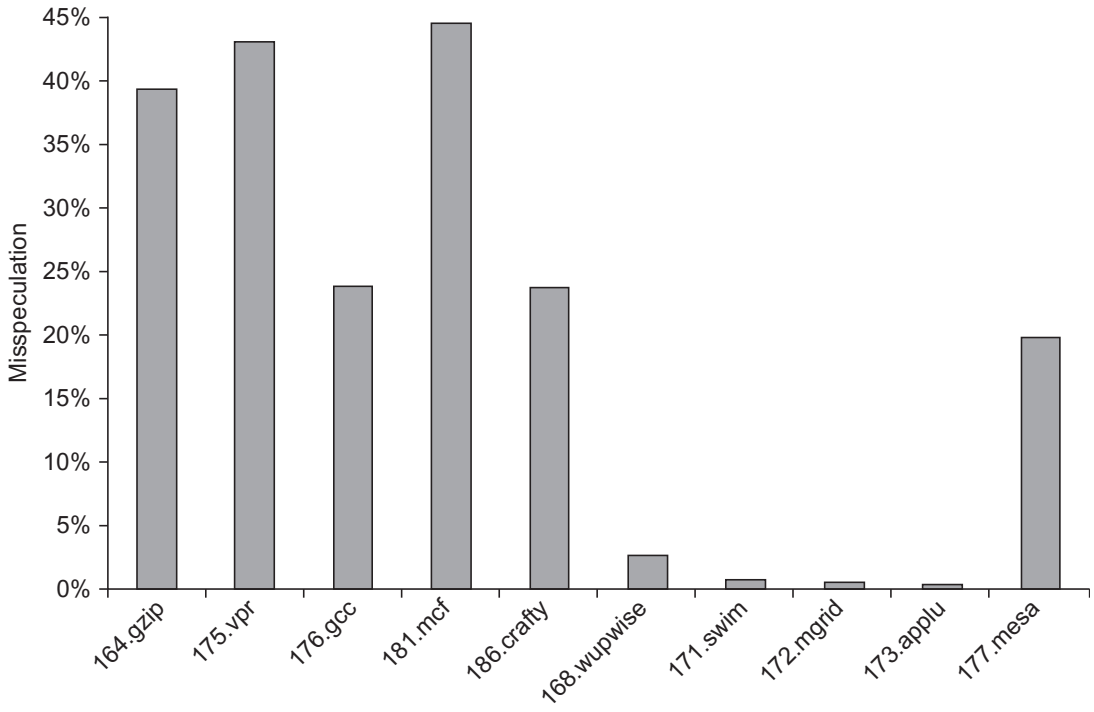

Figure 3.30 The fraction of instructions that are executed as a result of misspeculation is typically much higher for integer programs (the first five) versus FP programs (the last five).

improve running time by a comparable amount. Figure 3.30 shows the fraction of instructions that are executed from misspeculation for a subset of the SPEC2000 benchmarks using a sophisticated branch predictor. As we can see, this fraction of executed misspeculated instructions is small in scientific code and significant (about 30% on average) in integer code. Thus it is unlikely that speculation is energy-efficient for integer applications, and the end of Dennard scaling makes imperfect speculation more problematic. Designers could avoid speculation, try to reduce the misspeculation, or think about new approaches, such as only speculating on branches that are known to be highly predictable.

Address Aliasing Prediction

Address aliasing prediction is a technique that predicts whether two stores or a load and a store refer to the same memory address. If two such references do not refer to the same address, then they may be safely interchanged. Otherwise, we must wait until the memory addresses accessed by the instructions are known. Because we need not actually predict the address values, only whether such values conflict, the prediction can be reasonably accurate with small predictors. Address prediction relies on the ability of a speculative processor to recover after a misprediction; that is, if the actual addresses that were predicted to be different (and thus not alias) turn out to be the same (and thus are aliases), the processor simply restarts the sequence,

just as though it had mispredicted a branch. Address value speculation has been used in several processors already and may become universal in the future.

Address prediction is a simple and restricted form of *value prediction,* which attempts to predict the value that will be produced by an instruction. Value prediction could, if it were highly accurate, eliminate data flow restrictions and achieve higher rates of ILP. Despite many researchers focusing on value prediction in the past 15 years in dozens of papers, the results have never been sufficiently attractive to justify general value prediction in real processors.

3.10 Cross-Cutting Issues

Hardware Versus Software Speculation

The hardware-intensive approaches to speculation in this chapter and the software approaches of Appendix H provide alternative approaches to exploiting ILP. Some of the trade-offs, and the limitations, for these approaches are listed here:

- To speculate extensively, we must be able to disambiguate memory references. This capability is difficult to do at compile time for integer programs that contain pointers. In a hardware-based scheme, dynamic runtime disambiguation of memory addresses is done using the techniques we saw earlier for Tomasulo's algorithm. This disambiguation allows us to move loads past stores at runtime. Support for speculative memory references can help overcome the conservatism of the compiler, but unless such approaches are used carefully, the overhead of the recovery mechanisms may swamp the advantages.
- Hardware-based speculation works better when control flow is unpredictable and when hardware-based branch prediction is superior to software-based branch prediction done at compile time. These properties hold for many integer programs, where the misprediction rates for dynamic predictors are usually less than one-half of those for static predictors. Because speculated instructions may slow down the computation when the prediction is incorrect, this difference is significant. One result of this difference is that even statically scheduled processors normally include dynamic branch predictors.
- Hardware-based speculation maintains a completely precise exception model even for speculated instructions. Recent software-based approaches have added special support to allow this as well.
- Hardware-based speculation does not require compensation or bookkeeping code, which is needed by ambitious software speculation mechanisms.
- Compiler-based approaches may benefit from the ability to see further into the code sequence, resulting in better code scheduling than a purely hardware-driven approach.
- Hardware-based speculation with dynamic scheduling does not require different code sequences to achieve good performance for different implementations

of an architecture. Although this advantage is the hardest to quantify, it may be the most important one in the long run. Interestingly, this was one of the motivations for the IBM 360/91. On the other hand, more recent explicitly parallel architectures, such as IA-64, have added flexibility that reduces the hardware dependence inherent in a code sequence.

The major disadvantage of supporting speculation in hardware is the complexity and additional hardware resources required. This hardware cost must be evaluated against both the complexity of a compiler for a software-based approach and the amount and usefulness of the simplifications in a processor that relies on such a compiler.

Some designers have tried to combine the dynamic and compiler-based approaches to achieve the best of each. Such a combination can generate interesting and obscure interactions. For example, if conditional moves are combined with register renaming, a subtle side effect appears. A conditional move that is annulled must still copy a value to the destination register because it was renamed earlier in the instruction pipeline. These subtle interactions complicate the design and verification process and can also reduce performance.

The Intel Itanium processor was the most ambitious computer ever designed based on the software support for ILP and speculation. It did not deliver on the hopes of the designers, especially for general-purpose, nonscientific code. As designers' ambitions for exploiting ILP were reduced in light of the difficulties described on page 244, most architectures settled on hardware-based mechanisms with issue rates of three to four instructions per clock.

Speculative Execution and the Memory System

Inherent in processors that support speculative execution or conditional instructions is the possibility of generating invalid addresses that would not occur without speculative execution. Not only would this be incorrect behavior if protection exceptions were taken, but also the benefits of speculative execution would be swamped by false exception overhead. Therefore the memory system must identify speculatively executed instructions and conditionally executed instructions and suppress the corresponding exception.

By similar reasoning, we cannot allow such instructions to cause the cache to stall on a miss because, again, unnecessary stalls could overwhelm the benefits of speculation. Thus these processors must be matched with nonblocking caches.

In reality, the penalty of a miss that goes to DRAM is so large that speculated misses are handled only when the next level is on-chip cache (L2 or L3). Figure 2.5 on page 84 shows that for some well-behaved scientific programs, the compiler can sustain multiple outstanding L2 misses to cut the L2 miss penalty effectively. Once again, for this to work, the memory system behind the cache must match the goals of the compiler in number of simultaneous memory accesses.

3.11 Multithreading: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput

The topic we cover in this section, multithreading, is truly a cross-cutting topic, because it has relevance to pipelining and superscalars, to graphics processing units (Chapter 4), and to multiprocessors (Chapter 5). A *thread* is like a process in that it has state and a current program counter, but threads typically share the address space of a single process, allowing a thread to easily access data of other threads within the same process. Multithreading is a technique whereby multiple threads share a processor without requiring an intervening process switch. The ability to switch between threads rapidly is what enables multithreading to be used to hide pipeline and memory latencies.

In the next chapter, we will see how multithreading provides the same advantages in GPUs. Finally, Chapter 5 will explore the combination of multithreading and multiprocessing. These topics are closely interwoven because multithreading is a primary technique for exposing more parallelism to the hardware. In a strict sense, multithreading uses thread-level parallelism, and thus is properly the subject of Chapter 5, but its role in both improving pipeline utilization and in GPUs motivates us to introduce the concept here.

Although increasing performance by using ILP has the great advantage that it is reasonably transparent to the programmer, as we have seen, ILP can be quite limited or difficult to exploit in some applications. In particular, with reasonable instruction issue rates, cache misses that go to memory or off-chip caches are unlikely to be hidden by available ILP. Of course, when the processor is stalled waiting on a cache miss, the utilization of the functional units drops dramatically.

Because attempts to cover long memory stalls with more ILP have limited effectiveness, it is natural to ask whether other forms of parallelism in an application could be used to hide memory delays. For example, an online transaction processing system has natural parallelism among the multiple queries and updates that are presented by requests. Of course, many scientific applications contain natural parallelism because they often model the three-dimensional, parallel structure of nature, and that structure can be exploited by using separate threads. Even desktop applications that use modern Windows-based operating systems often have multiple active applications running, providing a source of parallelism.

Multithreading allows multiple threads to share the functional units of a single processor in an overlapping fashion. In contrast, a more general method to exploit *thread-level parallelism* (TLP) is with a multiprocessor that has multiple independent threads operating at once and in parallel. Multithreading, however, does not duplicate the entire processor as a multiprocessor does. Instead, multithreading shares most of the processor core among a set of threads, duplicating only private state, such as the registers and program counter. As we will see in Chapter 5, many recent processors incorporate both multiple processor cores on a single chip and provide multithreading within each core.

Duplicating the per-thread state of a processor core means creating a separate register file and a separate PC for each thread. The memory itself can be shared through the virtual memory mechanisms, which already support multiprogramming. In addition, the hardware must support the ability to change to a different thread relatively quickly; in particular, a thread switch should be much more efficient than a process switch, which typically requires hundreds to thousands of processor cycles. Of course, for multithreading hardware to achieve performance improvements, a program must contain multiple threads (we sometimes say that the application is multithreaded) that could execute in concurrent fashion. These threads are identified either by a compiler (typically from a language with parallelism constructs) or by the programmer.

There are three main hardware approaches to multithreading: fine-grained, coarse-grained, and simultaneous. *Fine-grained multithreading* switches between threads on each clock cycle, causing the execution of instructions from multiple threads to be interleaved. This interleaving is often done in a round-robin fashion, skipping any threads that are stalled at that time. One key advantage of fine-grained multithreading is that it can hide the throughput losses that arise from both short and long stalls because instructions from other threads can be executed when one thread stalls, even if the stall is only for a few cycles. The primary disadvantage of fine-grained multithreading is that it slows down the execution of an individual thread because a thread that is ready to execute without stalls will be delayed by instructions from other threads. It trades an increase in multithreaded throughput for a loss in the performance (as measured by latency) of a single thread.

The SPARC T1 through T5 processors (originally made by Sun, now made by Oracle and Fujitsu) use fine-grained multithreading. These processors were targeted at multithreaded workloads such as transaction processing and web services. The T1 supported 8 cores per processor and 4 threads per core, while the T5 supports 16 cores and 128 threads per core. Later versions (T2–T5) also supported 4–8 processors. The NVIDIA GPUs, which we look at in the next chapter, also make use of fine-grained multithreading.

Coarse-grained multithreading was invented as an alternative to fine-grained multithreading. Coarse-grained multithreading switches threads only on costly stalls, such as level two or three cache misses. Because instructions from other threads will be issued only when a thread encounters a costly stall, coarse-grained multithreading relieves the need to have thread-switching be essentially free and is much less likely to slow down the execution of any one thread.

Coarse-grained multithreading suffers, however, from a major drawback: it is limited in its ability to overcome throughput losses, especially from shorter stalls. This limitation arises from the pipeline start-up costs of coarse-grained multithreading. Because a processor with coarse-grained multithreading issues instructions from a single thread, when a stall occurs, the pipeline will see a bubble before the new thread begins executing. Because of this start-up overhead, coarse-grained multithreading is much more useful for reducing the penalty of very high-cost stalls, where pipeline refill is negligible compared to the stall time. Several research

projects have explored coarse-grained multithreading, but no major current processors use this technique.

The most common implementation of multithreading is called *simultaneous multithreading* (SMT). Simultaneous multithreading is a variation on fine-grained multithreading that arises naturally when fine-grained multithreading is implemented on top of a multiple-issue, dynamically scheduled processor. As with other forms of multithreading, SMT uses thread-level parallelism to hide long-latency events in a processor, thereby increasing the usage of the functional units. The key insight in SMT is that register renaming and dynamic scheduling allow multiple instructions from independent threads to be executed without regard to the dependences among them; the resolution of the dependences can be handled by the dynamic scheduling capability.

Figure 3.31 conceptually illustrates the differences in a processor's ability to exploit the resources of a superscalar for the following processor configurations:

- A superscalar with no multithreading support
- A superscalar with coarse-grained multithreading

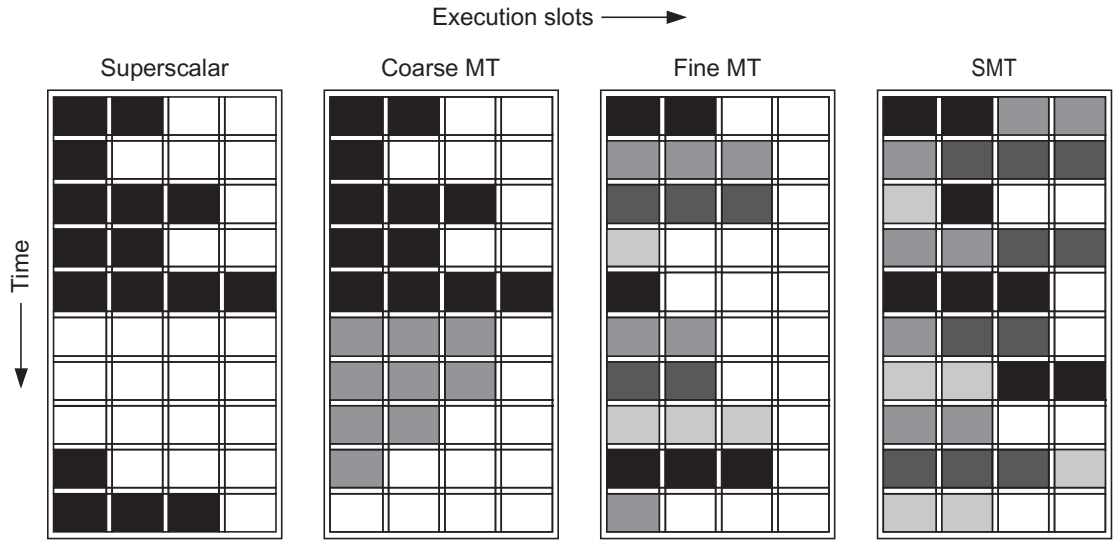

Figure 3.31 How four different approaches use the functional unit execution slots of a superscalar processor. The horizontal dimension represents the instruction execution capability in each clock cycle. The vertical dimension represents a sequence of clock cycles. An empty (white) box indicates that the corresponding execution slot is unused in that clock cycle. The shades of gray and black correspond to four different threads in the multithreading processors. Black is also used to indicate the occupied issue slots in the case of the superscalar without multithreading support. The Sun T1 and T2 (aka Niagara) processors are fine-grained, multithreaded processors, while the Intel Core i7 and IBM Power7 processors use SMT. The T2 has 8 threads, the Power7 has 4, and the Intel i7 has 2. In all existing SMTs, instructions issue from only one thread at a time. The difference in SMT is that the subsequent decision to execute an instruction is decoupled and could execute the operations coming from several different instructions in the same clock cycle.

- A superscalar with fine-grained multithreading
- A superscalar with simultaneous multithreading

In the superscalar without multithreading support, the use of issue slots is limited by a lack of ILP, including ILP to hide memory latency. Because of the length of L2 and L3 cache misses, much of the processor can be left idle.

In the coarse-grained multithreaded superscalar, the long stalls are partially hidden by switching to another thread that uses the resources of the processor. This switching reduces the number of completely idle clock cycles. In a coarse-grained multithreaded processor, however, thread switching occurs only when there is a stall. Because the new thread has a start-up period, there are likely to be some fully idle cycles remaining.

In the fine-grained case, the interleaving of threads can eliminate fully empty slots. In addition, because the issuing thread is changed on every clock cycle, longer latency operations can be hidden. Because instruction issue and execution are connected, a thread can issue only as many instructions as are ready. With a narrow issue width, this is not a problem (a cycle is either occupied or not), which is why fine-grained multithreading works perfectly for a single issue processor, and SMT would make no sense. Indeed, in the Sun T2, there are two issues per clock, but they are from different threads. This eliminates the need to implement the complex dynamic scheduling approach and relies instead on hiding latency with more threads.

If one implements fine-grained threading on top of a multiple-issue, dynamically schedule processor, the result is SMT. In all existing SMT implementations, all issues come from one thread, although instructions from different threads can initiate execution in the same cycle, using the dynamic scheduling hardware to determine what instructions are ready. Although Figure 3.31 greatly simplifies the real operation of these processors, it does illustrate the potential performance advantages of multithreading in general and SMT in wider issue, dynamically scheduled processors.

Simultaneous multithreading uses the insight that a dynamically scheduled processor already has many of the hardware mechanisms needed to support the mechanism, including a large virtual register set. Multithreading can be built on top of an out-of-order processor by adding a per-thread renaming table, keeping separate PCs, and providing the capability for instructions from multiple threads to commit.

Effectiveness of Simultaneous Multithreading on Superscalar Processors

A key question is, how much performance can be gained by implementing SMT? When this question was explored in 2000–2001, researchers assumed that dynamic superscalars would get much wider in the next five years, supporting six to eight issues per clock with speculative dynamic scheduling, many simultaneous loads and stores, large primary caches, and four to eight contexts with simultaneous issue

and retirement from multiple contexts. No processor has gotten close to this combination.

As a result, simulation research results that showed gains for multiprogrammed workloads of two or more times are unrealistic. In practice, the existing implementations of SMT offer only two to four contexts with fetching and issue from only one, and up to four issues per clock. The result is that the gain from SMT is also more modest.

Esmaeilzadeh et al. (2011) did an extensive and insightful set of measurements that examined both the performance and energy benefits of using SMT in a single i7 920 core running a set of multithreaded applications. The Intel i7 920 supported SMT with two threads per core, as does the recent i7 6700. The changes between the i7 920 and the 6700 are relatively small and are unlikely to significantly change the results as shown in this section.

The benchmarks used consist of a collection of parallel scientific applications and a set of multithreaded Java programs from the DaCapo and SPEC Java suite, as summarized in Figure 3.32. Figure 3.31 shows the ratios of performance and energy efficiency for these benchmarks when run on one core of a i7 920 with SMT turned off and on. (We plot the energy efficiency ratio, which is the inverse of energy consumption, so that, like speedup, a higher ratio is better.)

The harmonic mean of the speedup for the Java benchmarks is 1.28, despite the two benchmarks that see small gains. These two benchmarks, pjbb2005 and tradebeans, while multithreaded, have limited parallelism. They are included because they are typical of a multithreaded benchmark that might be run on an SMT processor with the hope of extracting some performance, which they find in limited amounts. The PARSEC benchmarks obtain somewhat better speedups than the full set of Java benchmarks (harmonic mean of 1.31). If tradebeans and pjbb2005 were omitted, the Java workload would actually have significantly better speedup (1.39) than the PARSEC benchmarks. (See the discussion of the implication of using harmonic mean to summarize the results in the caption of Figure 3.33.)

Energy consumption is determined by the combination of speedup and increase in power consumption. For the Java benchmarks, on average, SMT delivers the same energy efficiency as non-SMT (average of 1.0), but it is brought down by the two poor performing benchmarks; without pjbb2005 and tradebeans, the average energy efficiency for the Java benchmarks is 1.06, which is almost as good as the PARSEC benchmarks. In the PARSEC benchmarks, SMT reduces energy by $1-(1/1.08)=7\%$. Such energy-reducing performance enhancements are *very difficult* to find. Of course, the static power associated with SMT is paid in both cases, thus the results probably slightly overstate the energy gains.

These results clearly show that SMT with extensive support in an aggressive speculative processor can improve performance in an energy-efficient fashion. In 2011, the balance between offering multiple simpler cores and fewer more sophisticated cores has shifted in favor of more cores, with each core typically being a three- to four-issue superscalar with SMT supporting two to four threads. Indeed, Esmaeilzadeh et al. (2011) show that the energy improvements from SMT are even larger on the Intel i5 (a processor similar to the i7, but with smaller caches and a

blackscholes	Prices a portfolio of options with the Black-Scholes PDE
bodytrack	Tracks a markerless human body
canneal	Minimizes routing cost of a chip with cache-aware simulated annealing
facesim	Simulates motions of a human face for visualization purposes
ferret	Search engine that finds a set of images similar to a query image
fluidanimate	Simulates physics of fluid motion for animation with SPH algorithm
raytrace	Uses physical simulation for visualization
streamcluster	Computes an approximation for the optimal clustering of data points
swaptions	Prices a portfolio of swap options with the Heath–Jarrow–Morton framework
vips	Applies a series of transformations to an image
x264	MPG-4 AVC/H.264 video encoder
eclipse	Integrated development environment
lusearch	Text search tool
sunflow	Photo-realistic rendering system
tomcat	Tomcat servlet container
tradebeans	Tradebeans Daytrader benchmark
xalan	An XSLT processor for transforming XML documents
pjbb2005	Version of SPEC JBB2005 (but fixed in problem size rather than time)

Figure 3.32 The parallel benchmarks used here to examine multithreading, as well as in Chapter 5 to examine multiprocessing with an i7. The top half of the chart consists of PARSEC benchmarks collected by Bienia et al. (2008). The PARSEC benchmarks are meant to be indicative of compute-intensive, parallel applications that would be appropriate for multicore processors. The lower half consists of multithreaded Java benchmarks from the DaCapo collection (see Blackburn et al., 2006) and pjbb2005 from SPEC. All of these benchmarks contain some parallelism; other Java benchmarks in the DaCapo and SPEC Java workloads use multiple threads but have little or no true parallelism and, hence, are not used here. See Esmaeilzadeh et al. (2011) for additional information on the characteristics of these benchmarks, relative to the measurements here and in Chapter 5.

lower clock rate) and the Intel Atom (an 80 x 86 processor originally designed for the netbook and PMD market, now focused on low-end PCs, and described in Section 3.13).

3.12 Putting It All Together: The Intel Core i7 6700 and ARM Cortex-A53

In this section, we explore the design of two multiple issue processors: the ARM Cortex-A53 core, which is used as the basis for several tablets and cell phones, and the Intel Core i7 6700, a high-end, dynamically scheduled, speculative processor intended for high-end desktops and server applications. We begin with the simpler processor.

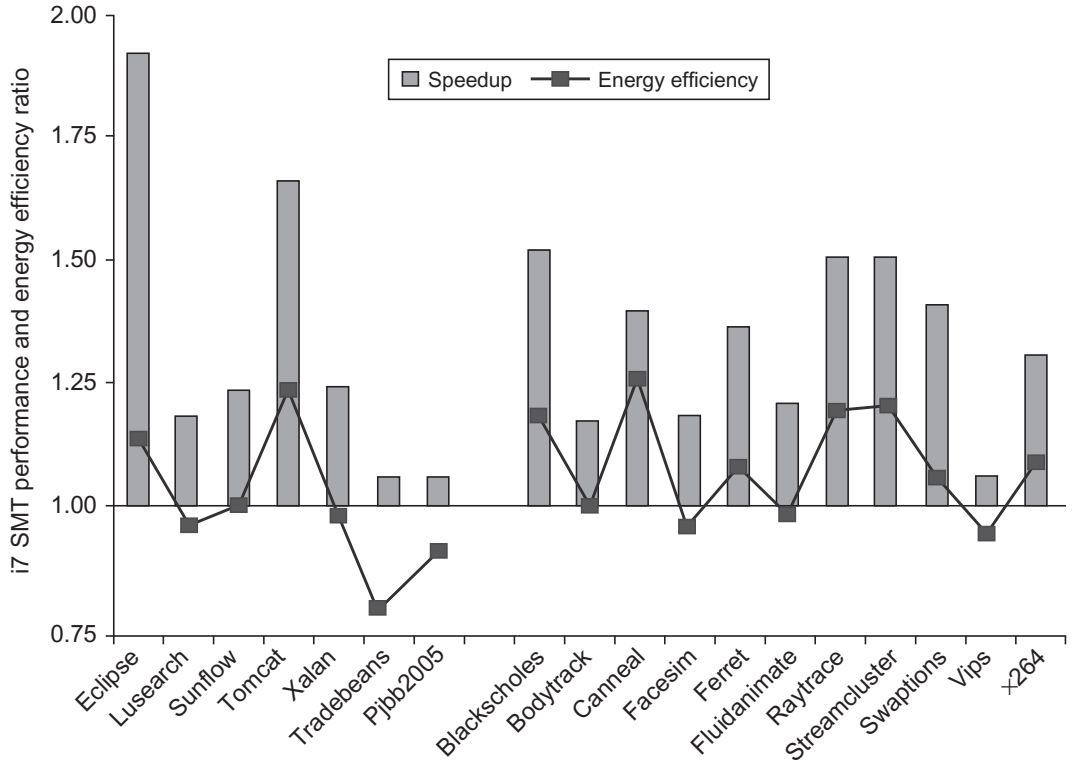

Figure 3.33 The speedup from using multithreading on one core on an i7 processor averages 1.28 for the Java benchmarks and 1.31 for the PARSEC benchmarks (using an unweighted harmonic mean, which implies a workload where the total time spent executing each benchmark in the single-threaded base set was the same). The energy efficiency averages 0.99 and 1.07, respectively (using the harmonic mean). Recall that anything above 1.0 for energy efficiency indicates that the feature reduces execution time by more than it increases average power. Two of the Java benchmarks experience little speedup and have significant negative energy efficiency because of this issue. Turbo Boost is off in all cases. These data were collected and analyzed by Esmaeilzadeh et al. (2011) using the Oracle (Sun) HotSpot build 16.3-b01 Java 1.6.0 Virtual Machine and the gcc v4.4.1 native compiler.

The ARM Cortex-A53

The A53 is a dual-issue, statically scheduled superscalar with dynamic issue detection, which allows the processor to issue two instructions per clock. Figure 3.34 shows the basic pipeline structure of the pipeline. For nonbranch, integer instructions, there are eight stages: F1, F2, D1, D2, D3/ISS, EX1, EX2, and WB, as described in the caption. The pipeline is in order, so an instruction can initiate execution only when its results are available and when proceeding instructions have initiated. Thus, if the next two instructions are dependent, both can proceed to the appropriate execution pipeline, but they will be serialized when they get to the beginning of that pipeline. When the scoreboard-based issue logic indicates that the result from the first instruction is available, the second instruction can issue.

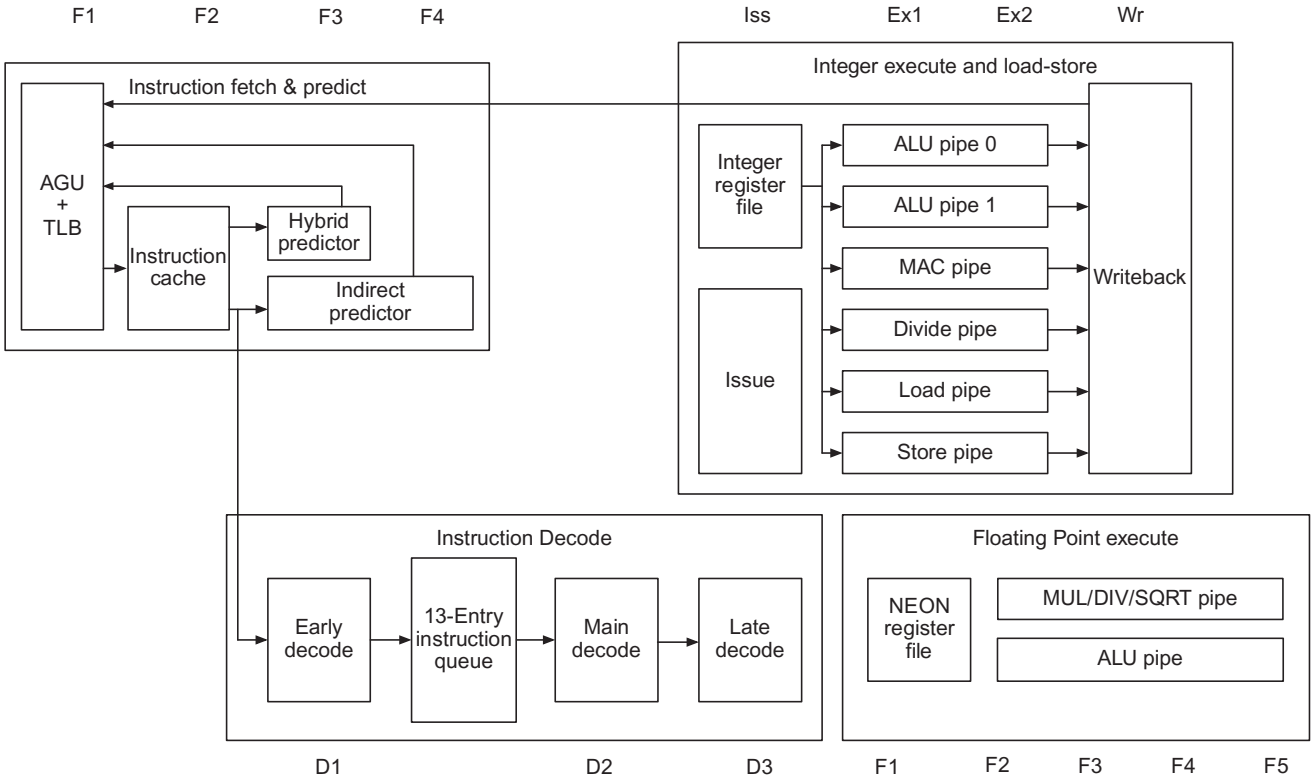

Figure 3.34 The basic structure of the A53 integer pipeline is 8 stages: F1 and F2 fetch the instruction, D1 and D2 do the basic decoding, and D3 decodes some more complex instructions and is overlapped with the first stage of the execution pipeline (ISS). After ISS, the Ex1, EX2, and WB stages complete the integer pipeline. Branches use four different predictors, depending on the type. The floating-point execution pipeline is 5 cycles deep, in addition to the 5 cycles needed for fetch and decode, yielding 10 stages in total.

The four cycles of instruction fetch include an address generation unit that produces the next PC either by incrementing the last PC or from one of four predictors:

1. A single-entry branch target cache containing two instruction cache fetches (the next two instructions following the branch, assuming the prediction is correct). This target cache is checked during the first fetch cycle, if it hits; then the next two instructions are supplied from the target cache. In case of a hit and a correct prediction, the branch is executed with no delay cycles.
2. A 3072-entry hybrid predictor, used for all instructions that do not hit in the branch target cache, and operating during F3. Branches handled by this predictor incur a 2-cycle delay.
3. A 256-entry indirect branch predictor that operates during F4; branches predicted by this predictor incur a three-cycle delay when predicted correctly.
4. An 8-deep return stack, operating during F4 and incurring a three-cycle delay.

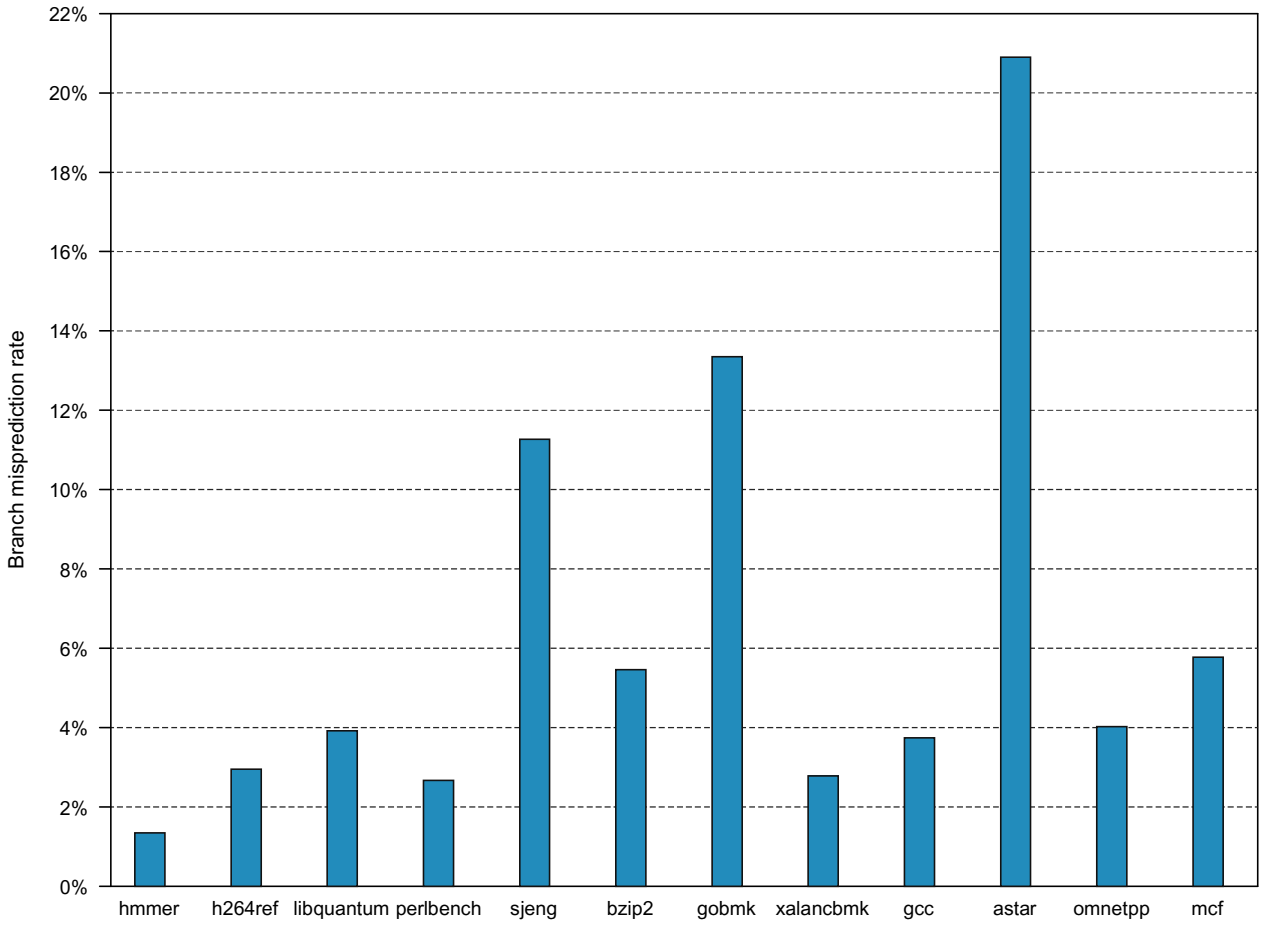

Figure 3.35 Misprediction rate of the A53 branch predictor for SPECint2006.

Branch decisions are made in ALU pipe 0, resulting in a branch misprediction penalty of 8 cycles. Figure 3.35 shows the misprediction rate for SPECint2006. The amount of work that is wasted depends on both the misprediction rate and the issue rate sustained during the time that the mispredicted branch was followed. As Figure 3.36 shows, wasted work generally follows the misprediction rate, though it may be larger or occasionally shorter.

Performance of the A53 Pipeline

The A53 has an ideal CPI of 0.5 because of its dual-issue structure. Pipeline stalls can arise from three sources:

1. Functional hazards, which occur because two adjacent instructions selected for issue simultaneously use the same functional pipeline. Because the A53 is statically scheduled, the compiler should try to avoid such conflicts. When such

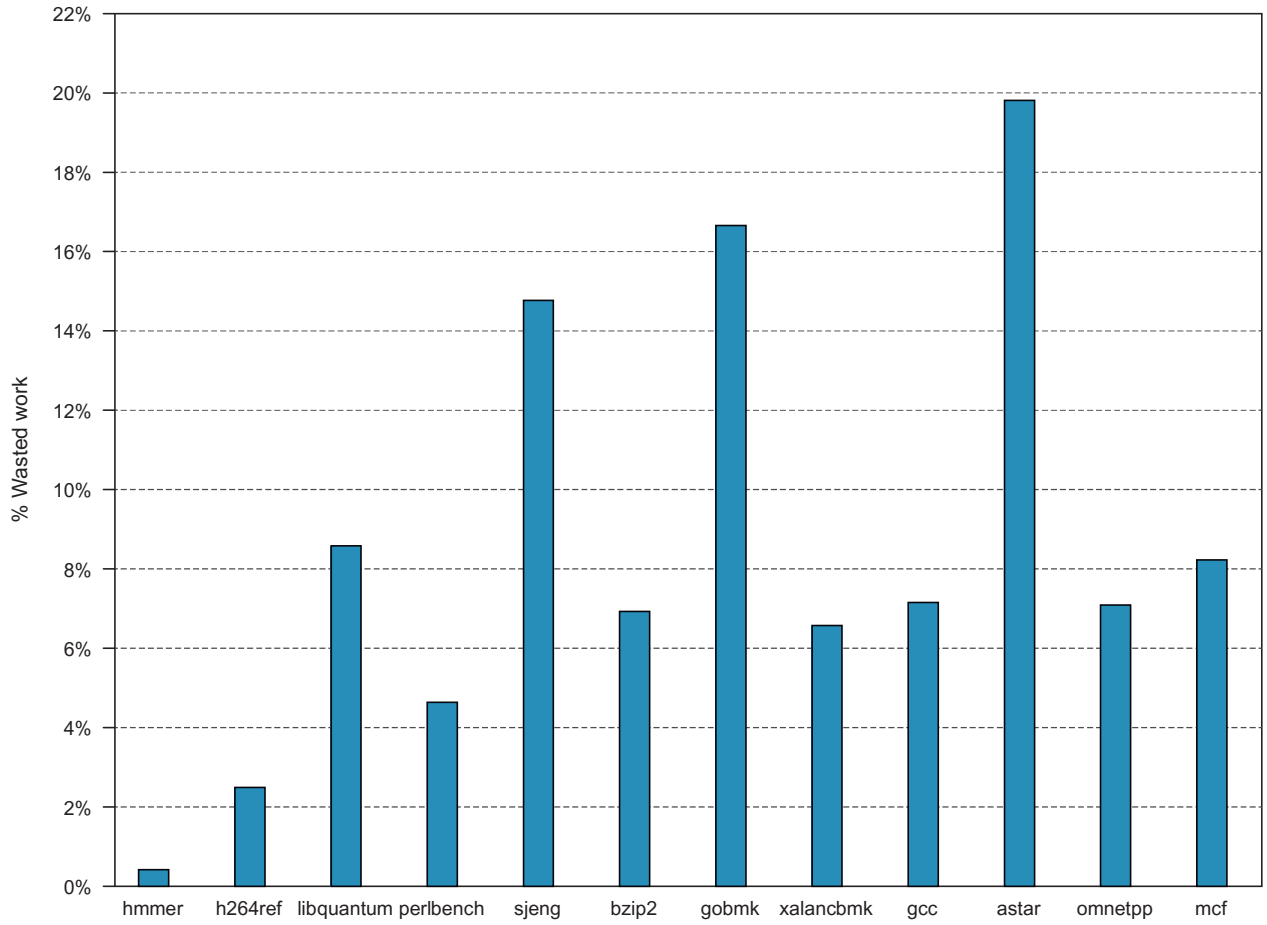

Figure 3.36 Wasted work due to branch misprediction on the A53. Because the A53 is an in-order machine, the amount of wasted work depends on a variety of factors, including data dependences and cache misses, both of which will cause a stall.

instructions appear sequentially, they will be serialized at the beginning of the execution pipeline, when only the first instruction will begin execution.

2. Data hazards, which are detected early in the pipeline and may stall either both instructions (if the first cannot issue, the second is always stalled) or the second of a pair. Again, the compiler should try to prevent such stalls when possible.
3. Control hazards, which arise only when branches are mispredicted.

Both TLB misses and cache misses also cause stalls. On the instruction side, a TLB or cache miss causes a delay in filling the instruction queue, likely leading to a downstream stall of the pipeline. Of course, this depends on whether it is an L1 miss, which might be largely hidden if the instruction queue was full at the time of the miss, or an L2 miss, which takes considerably longer. On the data side, a cache or TLB miss will cause the pipeline to stall because the load or store that

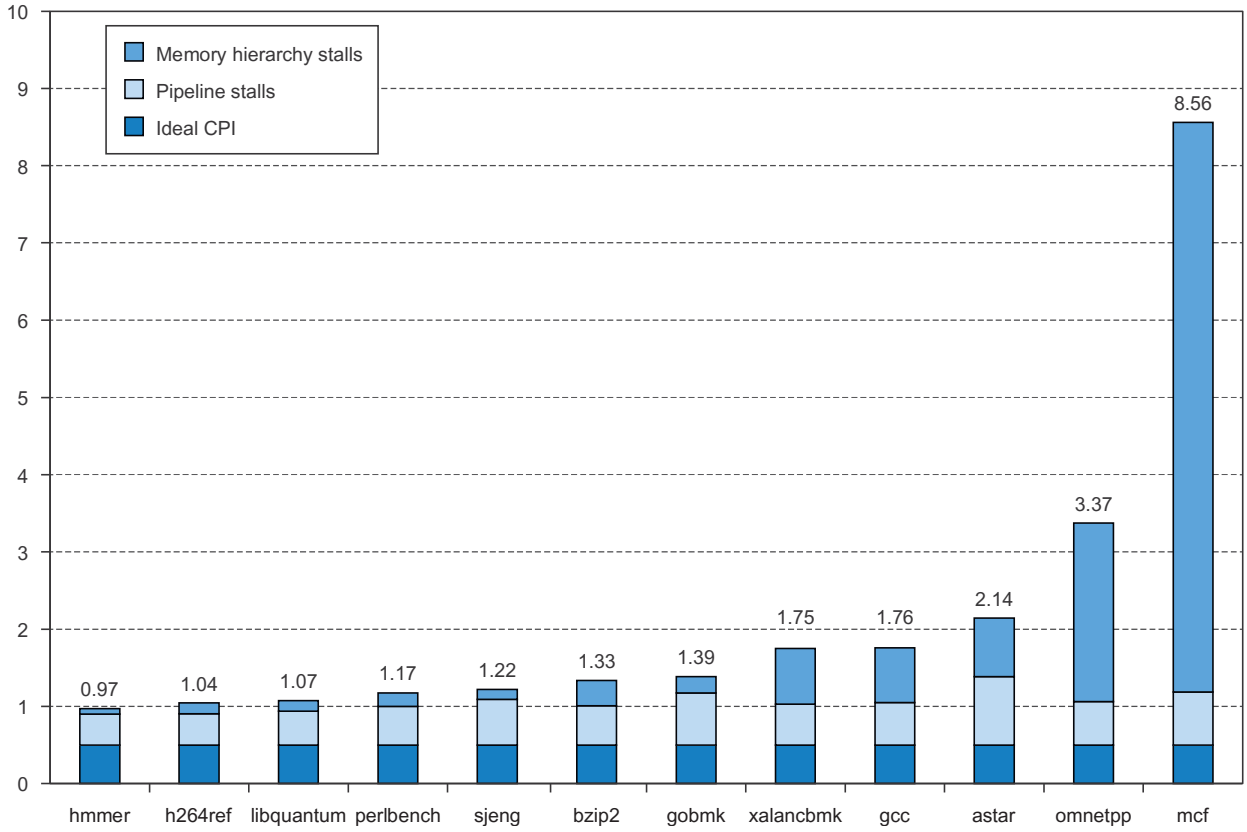

Figure 3.37 The estimated composition of the CPI on the ARM A53 shows that pipeline stalls are significant but are outweighed by cache misses in the poorest performing programs. This estimate is obtained by using the L1 and L2 miss rates and penalties to compute the L1 and L2 generated stalls per instruction. These are subtracted from the CPI measured by a detailed simulator to obtain the pipeline stalls. Pipeline stalls include all three hazards.

caused the miss cannot proceed down the pipeline. All other subsequent instructions will thus be stalled. Figure 3.37 shows the CPI and the estimated contributions from various sources.

The A53 uses a shallow pipeline and a reasonably aggressive branch predictor, leading to modest pipeline losses, while allowing the processor to achieve high clock rates at modest power consumption. In comparison with the i7, the A53 consumes approximately 1/200 the power for a quad core processor!

The Intel Core i7

The i7 uses an aggressive out-of-order speculative microarchitecture with deep pipelines with the goal of achieving high instruction throughput by combining multiple issue and high clock rates. The first i7 processor was introduced in 2008; the i7 6700 is the sixth generation. The basic structure of the i7 is similar, but successive

generations have enhanced performance by changing cache strategies (e.g., the aggressiveness of prefetching), increasing memory bandwidth, expanding the number of instructions in flight, enhancing branch prediction, and improving graphics support. The early i7 microarchitectures used reservations stations and reorder buffers for their out-of-order, speculative pipeline. Later microarchitectures, including the i7 6700, use register renaming, with the reservations stations acting as functional unit queues and the reorder buffer simply tracking control information.

Figure 3.38 shows the overall structure of the i7 pipeline. We will examine the pipeline by starting with instruction fetch and continuing on to instruction commit, following steps labeled in the figure.

1. Instruction fetch—The processor uses a sophisticated multilevel branch predictor to achieve a balance between speed and prediction accuracy. There is also a return address stack to speed up function return. Mispredictions cause a penalty of about 17 cycles. Using the predicted address, the instruction fetch unit fetches 16 bytes from the instruction cache.
2. The 16 bytes are placed in the predecode instruction buffer—In this step, a process called macro-op fusion is executed. *Macro-op fusion* takes instruction combinations such as compare followed by a branch and fuses them into a single operation, which can issue and dispatch as one instruction. Only certain special cases can be fused, since we must know that the only use of the first result is by the second instruction (i.e., compare and branch). In a study of the Intel Core architecture (which has many fewer buffers), Bird et al. (2007) discovered that macrofusion had a significant impact on the performance of integer programs resulting in an 8%–10% average increase in performance with a few programs showing negative results. There was little impact on FP programs; in fact, about half of the SPECFP benchmarks showed negative results from macro-op fusion. The predecode stage also breaks the 16 bytes into individual x86 instructions. This predecode is nontrivial because the length of an x86 instruction can be from 1 to 17 bytes and the predecoder must look through a number of bytes before it knows the instruction length. Individual x86 instructions (including some fused instructions) are placed into the instruction queue.
3. Micro-op decode—Individual x86 instructions are translated into micro-ops. Micro-ops are simple RISC-V-like instructions that can be executed directly by the pipeline; this approach of translating the x86 instruction set into simple operations that are more easily pipelined was introduced in the Pentium Pro in 1997 and has been used since. Three of the decoders handle x86 instructions that translate directly into one micro-op. For x86 instructions that have more complex semantics, there is a microcode engine that is used to produce the micro-op sequence; it can produce up to four micro-ops every cycle and continues until the necessary micro-op sequence has been generated. The micro-ops are placed according to the order of the x86 instructions in the 64-entry micro-op buffer.

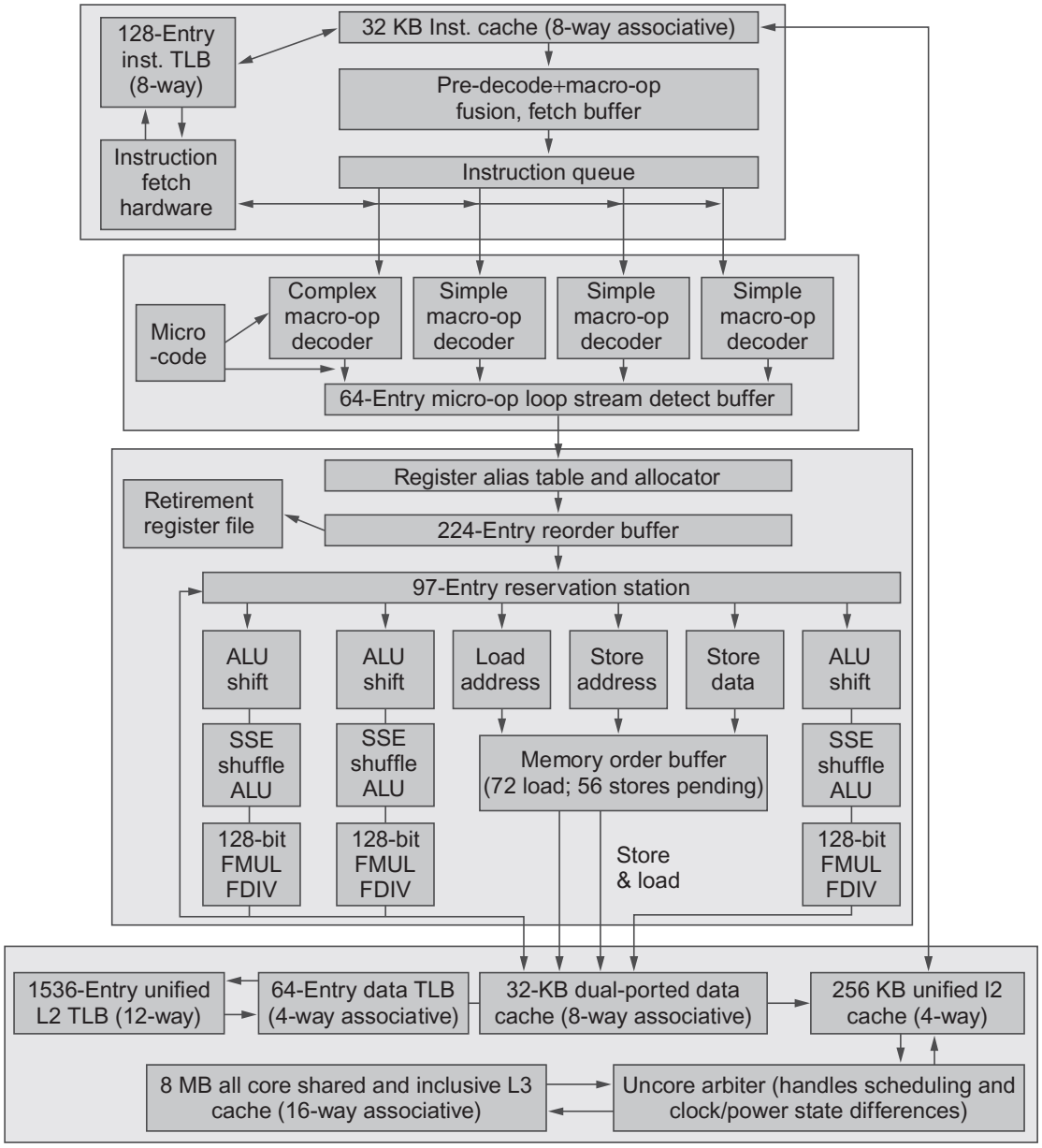

Figure 3.38 The Intel Core i7 pipeline structure shown with the memory system components. The total pipeline depth is 14 stages, with branch mispredictions typically costing 17 cycles, with the extra few cycles likely due to the time to reset the branch predictor. The six independent functional units can each begin execution of a ready micro-op in the same cycle. Up to four micro-ops can be processed in the register renaming table.

4. The micro-op buffer preforms *loop stream detection* and *microfusion*—If there is a small sequence of instructions (less than 64 instructions) that comprises a loop, the loop stream detector will find the loop and directly issue the micro-ops from the buffer, eliminating the need for the instruction fetch and instruction decode stages to be activated. Microfusion combines

instruction pairs such as ALU operation and a dependent store and issues them to a single reservation station (where they can still issue independently), thus increasing the usage of the buffer. Micro-op fusion produces smaller gains for integer programs and larger ones for FP, but the results vary widely. The different results for integer and FP programs with macro and micro fusion, probably arise from the patterns recognized and fused and the frequency of occurrence in integer versus FP programs. In the i7, which has a much larger number of reorder buffer entries, the benefits from both techniques are likely to be smaller.

5. Perform the basic instruction issue—Looking up the register location in the register tables, renaming the registers, allocating a reorder buffer entry, and fetching any results from the registers or reorder buffer before sending the micro-ops to the reservation stations. Up to four micro-ops can be processed every clock cycle; they are assigned the next available reorder buffer entries.
6. The i7 uses a centralized reservation station shared by six functional units. Up to six micro-ops may be dispatched to the functional units every clock cycle.
7. Micro-ops are executed by the individual function units, and then results are sent back to any waiting reservation station as well as to the register retirement unit, where they will update the register state once it is known that the instruction is no longer speculative. The entry corresponding to the instruction in the reorder buffer is marked as complete.
8. When one or more instructions at the head of the reorder buffer have been marked as complete, the pending writes in the register retirement unit are executed, and the instructions are removed from the reorder buffer.

In addition to the changes in the branch predictor, the major changes between the first generation i7 (the 920, Nehalem microarchitecture) and the sixth generation (i7 6700, Skylake microarchitecture) are in the sizes of the various buffers, renaming registers, and resources so as to allow many more outstanding instructions. Figure 3.39 summarizes these differences.

Performance of the i7

In earlier sections, we examined the performance of the i7's branch predictor and also the performance of SMT. In this section, we look at single-thread pipeline performance. Because of the presence of aggressive speculation as well as nonblocking caches, it is difficult to accurately attribute the gap between idealized performance and actual performance. The extensive queues and buffers on the 6700 reduce the probability of stalls because of a lack of reservation stations, renaming registers, or reorder buffers significantly. Indeed, even on the earlier i7 920 with notably fewer buffers, only about 3% of the loads were delayed because no reservation station was available.

Resource	i7 920 (Nehalem)	i7 6700 (Skylake)
Micro-op queue (per thread)	28	64
Reservation stations	36	97
Integer registers	NA	180
FP registers	NA	168
Outstanding load buffer	48	72
Outstanding store buffer	32	56
Reorder buffer	128	256

Figure 3.39 The buffers and queues in the first generation i7 and the latest generation i7. Nehalem used a reservation station plus reorder buffer organization. In later microarchitectures, the reservation stations serve as scheduling resources, and register renaming is used rather than the reorder buffer; the reorder buffer in the Skylake microarchitecture serves only to buffer control information. The choices of the size of various buffers and renaming registers, while appearing sometimes arbitrary, are likely based on extensive simulation.

Thus most losses come either from branch mispredicts or cache misses. The cost of a branch mispredict is 17 cycles, whereas the cost of an L1 miss is about 10 cycles. An L2 miss is slightly more than three times as costly as an L1 miss, and an L3 miss costs about 13 times what an L1 miss costs (130–135 cycles). Although the processor will attempt to find alternative instructions to execute during L2 and L3 misses, it is likely that some of the buffers will fill before a miss completes, causing the processor to stop issuing instructions.

Figure 3.40 shows the overall CPI for the 19 SPECCPUint2006 benchmarks compared to the CPI for the earlier i7 920. The average CPI on the i7 6700 is 0.71, whereas it is almost 1.5 times better on the i7 920, at 1.06. This difference derives from improved branch prediction and a reduction in the demand miss rates (see Figure 2.26 on page 135).

To understand how the 6700 achieves the significant improvement in CPI, let's look at the benchmarks that achieve the largest improvement. Figure 3.41 shows the five benchmarks that have a CPI ratio on the 920 that is at least 1.5 times higher than that of the 6700. Interestingly, three other benchmarks show a significant improvement in branch prediction accuracy (1.5 or more); however, those three benchmarks (HMMER, LIBQUANTUM, and SJENG) show equal or slightly higher L1 demand miss rates on the i7 6700. These misses likely arise because the aggressive prefetching is replacing cache blocks that are actually used. This type of behavior reminds designers of the challenges of maximizing performance in complex speculative multiple issue processors: rarely can significant performance be achieved by tuning only one part of the microarchitecture!

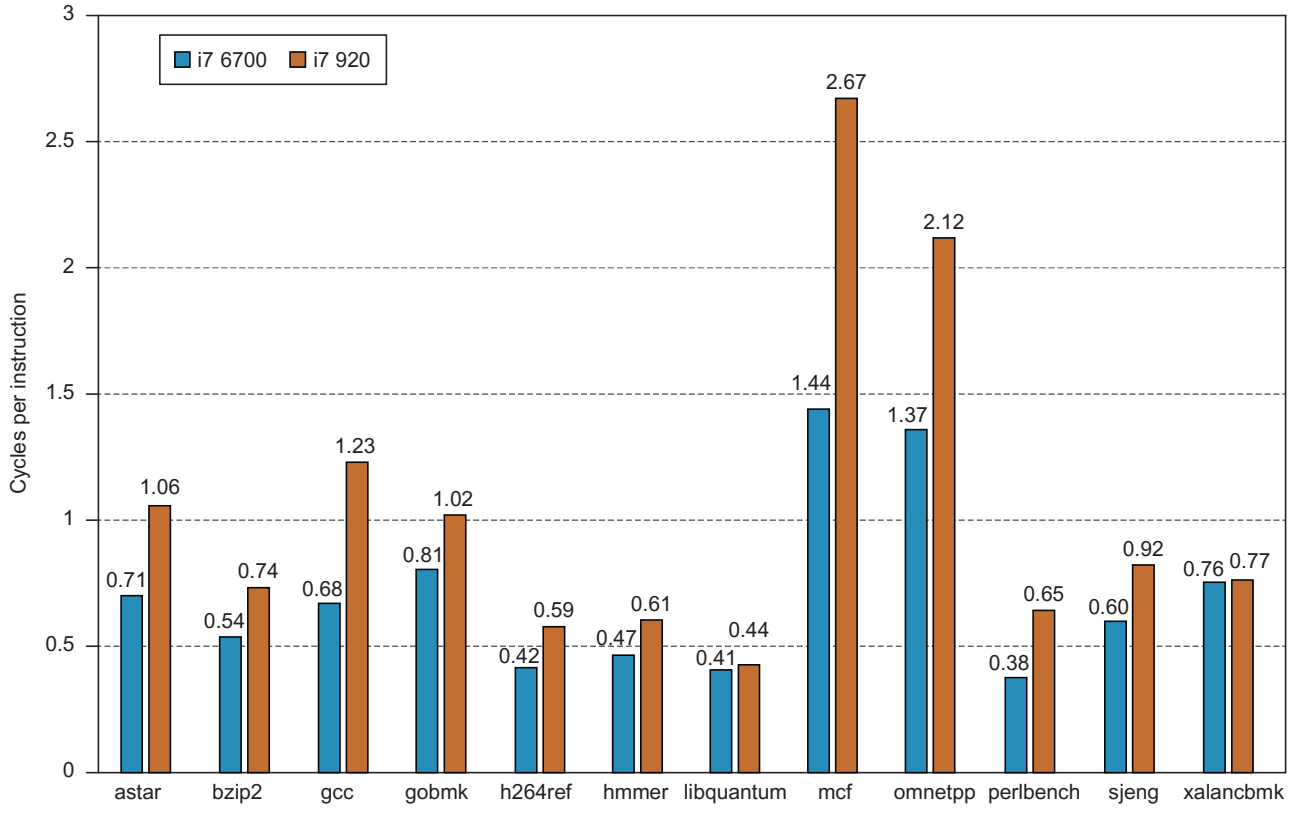

Figure 3.40 The CPI for the SPECCPUint2006 benchmarks on the i7 6700 and the i7 920. The data in this section were collected by Professor Lu Peng and PhD student Qun Liu, both of Louisiana State University.

Benchmark	CPI ratio (920/6700)	Branch mispredict ratio (920/6700)	L1 demand miss ratio (920/6700)
ASTAR	1.51	1.53	2.14
GCC	1.82	2.54	1.82
MCF	1.85	1.27	1.71
OMNETPP	1.55	1.48	1.96
PERLBENCH	1.70	2.11	1.78

Figure 3.41 An analysis of the five integer benchmarks with the largest performance gap between the i7 6700 and 920. These five benchmarks show an improvement in the branch prediction rate and a reduction in the L1 demand miss rate.

3.13 Fallacies and Pitfalls

Our few fallacies focus on the difficulty of predicting performance and energy efficiency and extrapolating from single measures such as clock rate or CPI. We also show that different architectural approaches can have radically different behaviors for different benchmarks.

Fallacy *It is easy to predict the performance and energy efficiency of two different versions of the same instruction set architecture, if we hold the technology constant.*

Intel offers a processor for the low-end Netbook and PMD space called the Atom 230, which implements both the 64-bit and 32-bit versions of the x86 architecture. The Atom is a statically scheduled, 2-issue superscalar, quite similar in its microarchitecture to the ARM A8, a single-core predecessor of the A53. Interestingly, both the Atom 230 and the Core i7 920 have been fabricated in the same 45 nm Intel technology. Figure 3.42 summarizes the Intel Core i7 920, the ARM Cortex-A8, and the Intel Atom 230. These similarities provide a rare opportunity to directly compare two radically different microarchitectures for the same instruction set while holding constant the underlying fabrication technology. Before we do the comparison, we need to say a little more about the Atom 230.

The Atom processors implement the x86 architecture using the standard technique of translating x86 instructions into RISC-like instructions (as every x86 implementation since the mid-1990s has done). Atom uses a slightly more powerful microoperation, which allows an arithmetic operation to be paired with a load or a store; this capability was added to later i7s by the use of macrofusion. This means that on average for a typical instruction mix, only 4% of the instructions require more than one microoperation. The microoperations are then executed in a 16-deep pipeline capable of issuing two instructions per clock, in order, as in the ARM A8. There are dual-integer ALUs, separate pipelines for FP add and other FP operations, and two memory operation pipelines, supporting more general dual execution than the ARM A8 but still limited by the in-order issue capability. The Atom 230 has a 32 KiB instruction cache and a 24 KiB data cache, both backed by a shared 512 KiB L2 on the same die. (The Atom 230 also supports multithreading with two threads, but we will consider only single-threaded comparisons.)

We might expect that these two processors, implemented in the same technology and with the same instruction set, would exhibit predictable behavior, in terms of relative performance and energy consumption, meaning that power and performance would scale close to linearly. We examine this hypothesis using three sets of benchmarks. The first set is a group of Java single-threaded benchmarks that come from the DaCapo benchmarks and the SPEC JVM98 benchmarks (see Esmaeilzadeh et al. (2011) for a discussion of the benchmarks and measurements). The second and third sets of benchmarks are from SPEC CPU2006 and consist of the integer and FP benchmarks, respectively.

Area	Specific characteristic	Intel i7 920 Four cores, each with FP	ARM A8 One core, no FP	Intel Atom 230 One core, with FP
Physical chip properties	Clock rate	2.66 GHz	1 GHz	1.66 GHz
	Thermal design power	130 W	2 W	4 W
	Package	1366-pin BGA	522-pin BGA	437-pin BGA
	TLB	Two-level All four-way set associative 128 I/64 D 512 L2	One-level fully associative 32 I/32 D	Two-level All four-way set associative 16 I/16 D 64 L2
	Caches	Three-level 32 KiB/32 KiB 256 KiB 2–8 MiB	Two-level 16/16 or 32/32 KiB 128 KiB–1 MiB	Two-level 32/24 KiB 512 KiB
Memory system	Peak memory BW	17 GB/s	12 GB/sec	8 GB/s
Pipeline structure	Peak issue rate	4 ops/clock with fusion	2 ops/clock	2 ops/clock
	Pipe line scheduling	Speculating out of order	In-order dynamic issue	In-order dynamic issue
	Branch prediction	Two-level	Two-level 512-entry BTB 4 K global history 8-entry return stack	Two-level

Figure 3.42 An overview of the four-core Intel i7 920, an example of a typical ARM A8 processor chip (with a 256 MiB L2, 32 KiB L1s, and no floating point), and the Intel ARM 230, clearly showing the difference in design philosophy between a processor intended for the PMD (in the case of ARM) or netbook space (in the case of Atom) and a processor for use in servers and high-end desktops. Remember, the i7 includes four cores, each of which is higher in performance than the one-core A8 or Atom. All these processors are implemented in a comparable 45 nm technology.

As we can see in Figure 3.43, the i7 significantly outperforms the Atom. All benchmarks are at least four times faster on the i7, two SPECFP benchmarks are over 10 times faster, and one SPECINT benchmark runs over eight times faster! Because the ratio of clock rates of these two processors is 1.6, most of the advantage comes from a much lower CPI for the i7 920: a factor of 2.8 for the Java benchmarks, a factor of 3.1 for the SPECINT benchmarks, and a factor of 4.3 for the SPECFP benchmarks.

But the average power consumption for the i7 920 is just under 43 W, while the average power consumption of the Atom is 4.2 W, or about one-tenth of the power! Combining the performance and power leads to an energy efficiency advantage for the Atom that is typically more than 1.5 times better and often 2 times better! This comparison of two processors using the same underlying technology makes it clear that the performance advantages of an aggressive

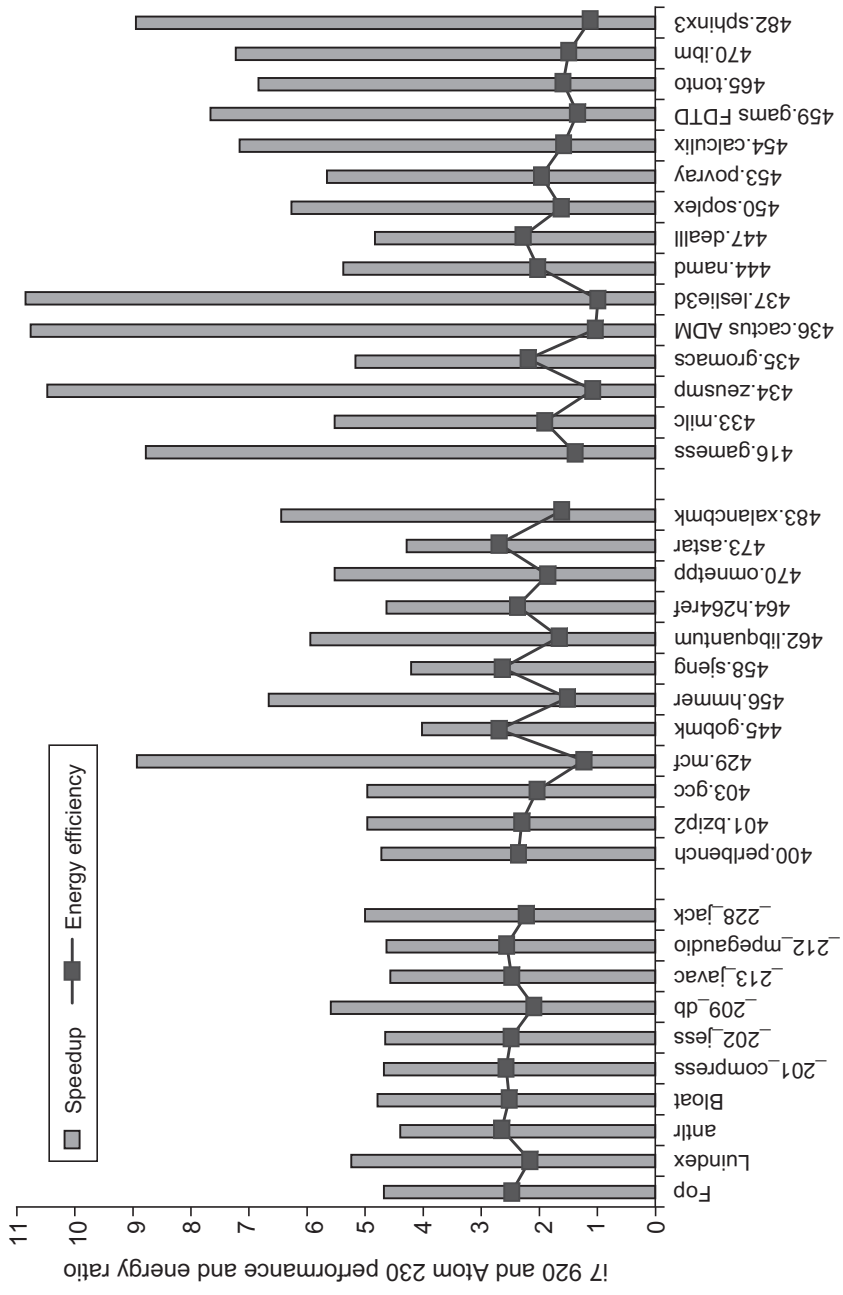

Figure 3.43 The relative performance and energy efficiency for a set of single-threaded benchmarks shows the i7 920 is 4 to over 10 times faster than the Atom 230 but that it is about 2 times *less* power-efficient on average! Performance is shown in the columns as i7 relative to Atom, which is execution time (i7)/execution time (Atom). Energy is shown with the line as Energy (Atom)/Energy (i7). The i7 never beats the Atom in energy efficiency, although it is essentially as good on four benchmarks, three of which are floating point. The data shown here were collected by Esmaeilzadeh et al. (2011). The SPEC benchmarks were compiled with optimization using the standard Intel compiler, while the Java benchmarks use the Sun (Oracle) Hotspot Java VM. Only one core is active on the i7, and the rest are in deep power saving mode. Turbo Boost is used on the i7, which increases its performance advantage but slightly decreases its relative energy efficiency.

superscalar with dynamic scheduling and speculation come with a *significant disadvantage* in energy efficiency.

Fallacy *Processors with lower CPIs will always be faster.*

Fallacy *Processors with faster clock rates will always be faster.*

The key is that it is the product of CPI and clock rate that determines performance. A high clock rate obtained by deeply pipelining the processor must maintain a low CPI to get the full benefit of the faster clock. Similarly, a simple processor with a high clock rate but a low CPI may be slower.

As we saw in the previous fallacy, performance and energy efficiency can diverge significantly among processors designed for different environments even when they have the same ISA. In fact, large differences in performance can show up even within a family of processors from the same company all designed for high-end applications. Figure 3.44 shows the integer and FP performance of two different implementations of the x86 architecture from Intel, as well as a version of the Itanium architecture, also by Intel.

The Pentium 4 was the most aggressively pipelined processor ever built by Intel. It used a pipeline with over 20 stages, had seven functional units, and cached micro-ops rather than x86 instructions. Its relatively inferior performance, given the aggressive implementation, was a clear indication that the attempt to exploit more ILP (there could easily be 50 instructions in flight) had failed. The Pentium's power consumption was similar to the i7, although its transistor count was lower, as its primary caches were half as large as the i7, and it included only a 2 MiB secondary cache with no tertiary cache.

The Intel Itanium is a VLIW-style architecture, which despite the potential decrease in complexity compared to dynamically scheduled superscalars, never attained competitive clock rates with the mainline x86 processors (although it appears to achieve an overall CPI similar to that of the i7). In examining these results, the reader should be aware that they use different implementation technologies, giving the i7 an advantage in terms of transistor speed and hence clock rate for an equivalently pipelined processor. Nonetheless, the wide variation in

Processor	Implementation technology	Clock rate	Power	SPECCInt2006 base	SPECCFP2006 baseline
Intel Pentium 4 670	90 nm	3.8 GHz	115 W	11.5	12.2
Intel Itanium 2	90 nm	1.66 GHz	104 W approx. 70 W one core	14.5	17.3
Intel i7 920	45 nm	3.3 GHz	130 W total approx. 80 W one core	35.5	38.4

Figure 3.44 Three different Intel processors vary widely. Although the Itanium processor has two cores and the i7 four, only one core is used in the benchmarks; the Power column is the thermal design power with estimates for only one core active in the multicore cases.

performance—more than three times between the Pentium and i7—is astonishing. The next pitfall explains where a significant amount of this advantage comes from.

Pitfall *Sometimes bigger and dumber is better.*

Much of the attention in the early 2000s went to building aggressive processors to exploit ILP, including the Pentium 4 architecture, which used the deepest pipeline ever seen in a microprocessor, and the Intel Itanium, which had the highest peak issue rate per clock ever seen. What quickly became clear was that the main limitation in exploiting ILP often turned out to be the memory system. Although speculative out-of-order pipelines were fairly good at hiding a significant fraction of the 10- to 15-cycle miss penalties for a first-level miss, they could do very little to hide the penalties for a second-level miss that, when going to main memory, were likely to be 50–100 clock cycles.

The result was that these designs never came close to achieving the peak instruction throughput despite the large transistor counts and extremely sophisticated and clever techniques. Section 3.15 discusses this dilemma and the turning away from more aggressive ILP schemes to multicore, but there was another change that exemplified this pitfall. Instead of trying to hide even more memory latency with ILP, designers simply used the transistors to build much larger caches. Both the Itanium 2 and the i7 use three-level caches compared to the two-level cache of the Pentium 4, and the third-level caches are 9 and 8 MiB compared to the 2 MiB second-level cache of the Pentium 4. Needless to say, building larger caches is a lot easier than designing the 20+-stage Pentium 4 pipeline, and based on the data in Figure 3.44, doing so seems to be more effective.

Pitfall *And sometimes smarter is better than bigger and dumber.*

One of the more surprising results of the past decade has been in branch prediction. The emergence of hybrid tagged predictors has shown that a more sophisticated predictor can outperform the simple gshare predictor with the same number of bits (see Figure 3.8 on page 171). One reason this result is so surprising is that the tagged predictor actually stores fewer predictions, because it also consumes bits to store tags, whereas gshare has only a large array of predictions. Nonetheless, it appears that the advantage gained by not misusing a prediction for one branch on another branch more than justifies the allocation of bits to tags versus predictions.

Pitfall *Believing that there are large amounts of ILP available, if only we had the right techniques.*

The attempts to exploit large amounts of ILP failed for several reasons, but one of the most important ones, which some designers did not initially accept, is that it is hard to find large amounts of ILP in conventionally structured programs, even with speculation. A famous study by David Wall in 1993 (see Wall, 1993) analyzed the amount of ILP available under a variety of idealistic conditions. We summarize his results for a processor configuration with roughly five to ten times the capability of the most advanced processors in 2017. Wall's study extensively documented a variety of different approaches,

and the reader interested in the challenge of exploiting ILP should read the complete study.

The aggressive processor we consider has the following characteristics:

1. Up to 64 instruction issues and dispatches per clock with *no* issue restrictions, or 8 times the total issue width of the widest processor in 2016 (the IBM Power8) and with up to 32 times as many loads and stores allowed per clock! As we have discussed, there are serious complexity and power problems with large issue rates.
2. A tournament predictor with 1K entries and a 16-entry function return predictor. This predictor is comparable to the best predictors in 2016; the predictor is not a primary bottleneck. Mispredictions are handled in one cycle, but they limit the ability to speculate.
3. Perfect disambiguation of memory references done dynamically—this is ambitious but perhaps attainable for small window sizes.
4. Register renaming with 64 additional integer and 64 additional FP registers, which is somewhat less than the most aggressive processor in 2011. Because the study assumes a latency of only one cycle for all instructions (versus 15 or more on processors like the i7 or Power8), the effective number of rename registers is about five times larger than either of those processors.

Figure 3.45 shows the result for this configuration as we vary the window size. This configuration is more complex and expensive than existing implementations, especially in terms of the number of instruction issues. Nonetheless, it gives a useful upper limit on what future implementations might yield. The data in these figures are likely to be very optimistic for another reason. There are no issue restrictions among the 64 instructions: for example, they may all be memory references. No one would even contemplate this capability in a processor for the near future. In addition, remember that in interpreting these results, cache misses and non-unit latencies were not taken into account, and both these effects have significant impacts.

The most startling observation in Figure 3.45 is that with the preceding realistic processor constraints, the effect of the window size for the integer programs is not as 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 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 most of the market growth in the past decade—transaction processing, web servers, and the like—depended on integer performance, rather than floating point.

Wall's study was not believed by some, but 10 years later, the reality had sunk in, and the combination of modest performance increases with significant hardware resources and major energy issues coming from incorrect speculation forced a change in direction. We will return to this discussion in our concluding remarks.

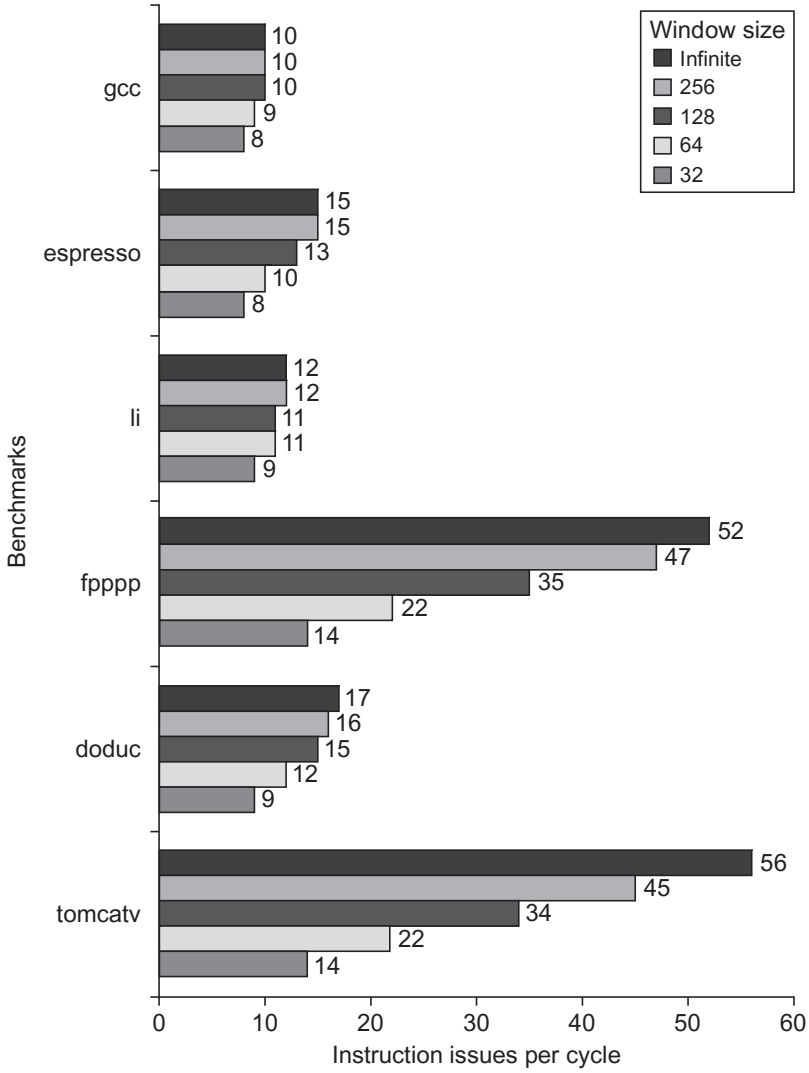

Figure 3.45 The amount of parallelism available versus the window size for a variety of integer and floating-point programs with up to 64 arbitrary instruction issues per clock. Although there are fewer renaming registers than the window size, the fact that all operations have 1-cycle latency and that the number of renaming registers equals the issue width allows the processor to exploit parallelism within the entire window.

3.14 Concluding Remarks: What's Ahead?

As 2000 began the focus on exploiting instruction-level parallelism was at its peak. In the first five years of the new century, it became clear that the ILP approach had likely peaked and that new approaches would be needed. By 2005 Intel and all the other major processor manufacturers had revamped their approach to focus on multicore. Higher performance would be achieved through thread-level parallelism rather than instruction-level parallelism, and the responsibility for using the

processor efficiently would largely shift from the hardware to the software and the programmer. This change was the most significant change in processor architecture since the early days of pipelining and instruction-level parallelism some 25+ years earlier.

During the same period, designers began to explore the use of more data-level parallelism as another approach to obtaining performance. SIMD extensions enabled desktop and server microprocessors to achieve moderate performance increases for graphics and similar functions. More importantly, graphics processing units (GPUs) pursued aggressive use of SIMD, achieving significant performance advantages for applications with extensive data-level parallelism. For scientific applications, such approaches represent a viable alternative to the more general, but less efficient, thread-level parallelism exploited in multicores. The next chapter explores these developments in the use of data-level parallelism.

Many researchers predicted a major retrenchment in the use of ILP, predicting that two issue superscalar processors and larger numbers of cores would be the future. The advantages, however, of slightly higher issue rates and the ability of speculative dynamic scheduling to deal with unpredictable events, such as level-one cache misses, led to moderate ILP (typically about 4 issues/clock) being the primary building block in multicore designs. The addition of SMT and its effectiveness (both for performance and energy efficiency) further cemented the position of the moderate issue, out-of-order, speculative approaches. Indeed, even in the embedded market, the newest processors (e.g., the ARM Cortex-A9 and Cortex-A73) have introduced dynamic scheduling, speculation, and wider issues rates.

It is highly unlikely that future processors will try to increase the width of issue significantly. It is simply too inefficient from the viewpoint of silicon utilization and power efficiency. Consider the data in Figure 3.46 that show the five processors in the IBM Power series. Over more than a decade, there has been a modest improvement in the ILP support in the Power processors, but the dominant portion

	Power4	Power5	Power6	Power7	Power8
Introduced	2001	2004	2007	2010	2014
Initial clock rate (GHz)	1.3	1.9	4.7	3.6	3.3 GHz
Transistor count (M)	174	276	790	1200	4200
Issues per clock	5	5	7	6	8
Functional units per core	8	8	9	12	16
SMT threads per core	0	2	2	4	8
Cores/chip	2	2	2	8	12
SMT threads per core	0	2	2	4	8
Total on-chip cache (MiB)	1.5	2	4.1	32.3	103.0

Figure 3.46 Characteristics of five generations of IBM Power processors. All except the Power6, which is static and in-order, were dynamically scheduled; all the processors support two load/store pipelines. The Power6 has the same functional units as the Power5 except for a decimal unit. Power7 and Power8 use embedded DRAM for the L3 cache. Power9 has been described briefly; it further expands the caches and supports off-chip HBM.

of the increase in transistor count (a factor of more than 10 from the Power4 to the Power8) went to increasing the caches and the number of cores per die. Even the expansion in SMT support seems to be more of a focus than is an increase in the ILP throughput: The ILP structure from Power4 to Power8 went from 5 issues to 8, from 8 functional units to 16 (but not increasing from the original 2 load/store units), whereas the SMT support went from nonexistent to 8 threads/processor. A similar trend can be observed across the six generations of i7 processors, where almost all the additional silicon has gone to supporting more cores. The next two chapters focus on approaches that exploit data-level and thread-level parallelism.

3.15 Historical Perspective and References

Section M.5 (available online) features a discussion on the development of pipelining and instruction-level parallelism. We provide numerous references for further reading and exploration of these topics. Section M.5 covers both Chapter 3 and Appendix H.

Case Studies and Exercises by Jason D. Bakos and Robert P. Colwell

Case Study: Exploring the Impact of Microarchitectural Techniques

Concepts illustrated by this case study

- Basic Instruction Scheduling, Reordering, Dispatch
- Multiple Issue and Hazards
- Register Renaming
- Out-of-Order and Speculative Execution
- Where to Spend Out-of-Order Resources

You are tasked with designing a new processor microarchitecture and you are trying to determine how best to allocate your hardware resources. Which of the hardware and software techniques you learned in Chapter 3 should you apply? You have a list of latencies for the functional units and for memory, as well as some representative code. Your boss has been somewhat vague about the performance requirements of your new design, but you know from experience that, all else being equal, faster is usually better. Start with the basics. Figure 3.47 provides a sequence of instructions and list of latencies.

3.1 [10] <3.1, 3.2> What is the baseline performance (in cycles, per loop iteration) of the code sequence in Figure 3.47 if no new instruction's execution could be

Latencies beyond single cycle	
Memory LD	+3
Memory SD	+1
Integer ADD, SUB	+0
Branches	+1
fadd.d	+2
fmul.d	+4
fdiv.d	+10

```
Loop:   fld     f2,0(Rx)
I0:     fmul.d  f2,f0,f2
I1:     fdiv.d  f8,f2,f0
I2:     fld     f4,0(Ry)
I3:     fadd.d  f4,f0,f4
I4:     fadd.d  f10,f8,f2
I5:     fsd     f4,0(Ry)
I6:     addi    Rx,Rx,8
I7:     addi    Ry,Ry,8
I8:     sub     x20,x4,Rx
I9:     bnz     x20,Loop
```

Figure 3.47 Code and latencies for Exercises 3.1 through 3.6.

initiated until the previous instruction's execution had completed? Ignore front-end fetch and decode. Assume for now that execution does not stall for lack of the next instruction, but only one instruction/cycle can be issued. Assume the branch is taken, and that there is a one-cycle branch delay slot.

3.2 [10] <3.1, 3.2> Think about what latency numbers really mean—they indicate the number of cycles a given function requires to produce its output. If the overall pipeline stalls for the latency cycles of each functional unit, then you are at least guaranteed that any pair of back-to-back instructions (a "producer" followed by a "consumer") will execute correctly. But not all instruction pairs have a producer/consumer relationship. Sometimes two adjacent instructions have nothing to do with each other. How many cycles would the loop body in the code sequence in Figure 3.47 require if the pipeline detected true data dependences and only stalled on those, rather than blindly stalling everything just because one functional unit is busy? Show the code with <stall> inserted where necessary to accommodate stated latencies. (*Hint:* an instruction with latency +2 requires two <stall> cycles to be inserted into the code sequence.) Think of it this

way: a one-cycle instruction has latency 1+0, meaning zero extra wait states. So, latency 1+1 implies one stall cycle; latency 1+N has N extra stall cycles.

3.3 [15] <3.1, 3.2> Consider a multiple-issue design. Suppose you have two execution pipelines, each capable of beginning execution of one instruction per cycle, and enough fetch/decode bandwidth in the front end so that it will not stall your execution. Assume results can be immediately forwarded from one execution unit to another, or to itself. Further assume that the only reason an execution pipeline would stall is to observe a true data dependency. Now how many cycles does the loop require?

3.4 [10] <3.1, 3.2> In the multiple-issue design of Exercise 3.3, you may have recognized some subtle issues. Even though the two pipelines have the exact same instruction repertoire, they are neither identical nor interchangeable, because there is an implicit ordering between them that must reflect the ordering of the instructions in the original program. If instruction N+1 begins execution in Execution Pipe 1 at the same time that instruction N begins in Pipe 0, and N+1 happens to require a shorter execution latency than N, then N+1 will complete before N (even though program ordering would have implied otherwise). Recite at least two reasons why that could be hazardous and will require special considerations in the microarchitecture. Give an example of two instructions from the code in Figure 3.47 that demonstrate this hazard.

3.5 [20] <3.1, 3.2> Reorder the instructions to improve performance of the code in Figure 3.47. Assume the two-pipe machine in Exercise 3.3 and that the out-of-order completion issues of Exercise 3.4 have been dealt with successfully. Just worry about observing true data dependences and functional unit latencies for now. How many cycles does your reordered code take?

3.6 [10/10/10] <3.1, 3.2> Every cycle that does not initiate a new operation in a pipe is a lost opportunity, in the sense that your hardware is not living up to its potential.

a. [10] <3.1, 3.2> In your reordered code from Exercise 3.5, what fraction of all cycles, counting both pipes, were wasted (did not initiate a new op)?

b. [10] <3.1, 3.2> Loop unrolling is one standard compiler technique for finding more parallelism in code, in order to minimize the lost opportunities for performance. Hand-unroll two iterations of the loop in your reordered code from Exercise 3.5.

c. [10] <3.1, 3.2> What speedup did you obtain? (For this exercise, just color the N+1 iteration's instructions green to distinguish them from the Nth iteration's instructions; if you were actually unrolling the loop, you would have to reassign registers to prevent collisions between the iterations.)

3.7 [15] <3.4> Computers spend most of their time in loops, so multiple loop iterations are great places to speculatively find more work to keep CPU resources busy. Nothing is ever easy, though; the compiler emitted only one copy of that loop's code, so even though multiple iterations are handling distinct data, they will

appear to use the same registers. To keep multiple iterations' register usages from colliding, we rename their registers. Figure 3.48 shows example code that we would like our hardware to rename. A compiler could have simply unrolled the loop and used different registers to avoid conflicts, but if we expect our hardware to unroll the loop, it must also do the register renaming. How? Assume your hardware has a pool of temporary registers (call them `T` registers, and assume that there are 64 of them, `T0` through `T63`) that it can substitute for those registers designated by the compiler. This rename hardware is indexed by the `src` (source) register designation, and the value in the table is the `T` register of the last destination that targeted that register. (Think of these table values as producers, and the `src` registers are the consumers; it doesn't much matter where the producer puts its result as long as its consumers can find it.) Consider the code sequence in Figure 3.48. Every time you see a destination register in the code, substitute the next available `T`, beginning with `T9`. Then update all the `src` registers accordingly, so that true data dependences are maintained. Show the resulting code. (*Hint:* see Figure 3.49.)

3.8 [20] <3.4> Exercise 3.7 explored simple register renaming: when the hardware register renamer sees a source register, it substitutes the destination `T` register of the last instruction to have targeted that source register. When the rename table sees a destination register, it substitutes the next available `T` for it, but superscalar designs need to handle multiple instructions per clock cycle at every stage in the machine, including the register renaming. A SimpleScalar processor would therefore look up both `src` register mappings for each instruction and allocate a new `dest` mapping per clock cycle. Superscalar processors must be able to do that as well, but they must also ensure that any `dest`-to-`src` relationships between the two concurrent instructions are handled correctly. Consider the sample code sequence in Figure 3.50. Assume that we would like to simultaneously

```
Loop:           fld             f2,0(Rx)
I0:             fmul.d          f5,f0,f2
I1:             fdiv.d          f8,f0,f2
I2:             fld             f4,0(Ry)
I3:             fadd.d          f6,f0,f4
I4:             fadd.d          f10,f8,f2
I5:             sd              f4,0(Ry)
```

Figure 3.48 Sample code for register renaming practice.

```
I0:           fld             T9,0(Rx)
I1:           fmul.d          T10,F0,T9
...
```

Figure 3.49 Expected output of register renaming.

```
I0:        fmul.d      f5,f0,f2
I1:        fadd.d      f9,f5,f4
I2:        fadd.d      f5,f5,f2
I3:        fdiv.d      f2,f9,f0
```

Figure 3.50 Sample code for superscalar register renaming.

rename the first two instructions. Further assume that the next two available T registers to be used are known at the beginning of the clock cycle in which these two instructions are being renamed. Conceptually, what we want is for the first instruction to do its rename table lookups and then update the table per its destination's T register. Then the second instruction would do exactly the same thing, and any inter-instruction dependency would thereby be handled correctly. But there's not enough time to write that T register designation into the renaming table and then look it up again for the second instruction, all in the same clock cycle. That register substitution must instead be done live (in parallel with the register rename table update). Figure 3.51 shows a circuit diagram, using multiplexers and comparators, that will accomplish the necessary on-the-fly register renaming. Your task is to show the cycle-by-cycle state of the rename table for every instruction of the code shown in Figure 3.50. Assume the table starts out with every entry equal to its index (T0=0; T1=1, ...) (Figure 3.51).

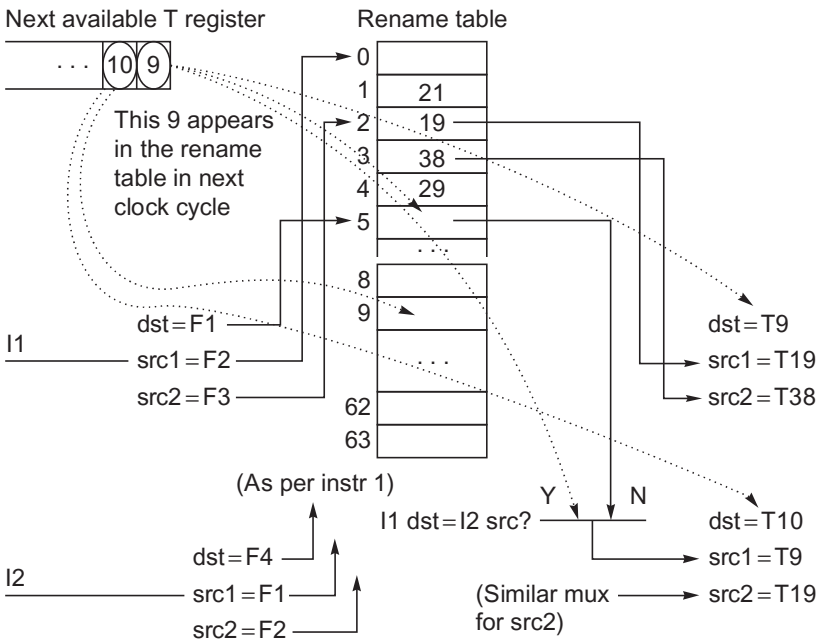

Figure 3.51 Initial state of the register renaming table.

3.9 [5] <3.4> If you ever get confused about what a register renamer has to do, go back to the assembly code you're executing, and ask yourself what has to happen for the right result to be obtained. For example, consider a three-way superscalar machine renaming these three instructions concurrently:

```
addi x1, x1, x1
addi x1, x1, x1
addi x1, x1, x1
```

If the value of `x1` starts out as 5, what should its value be when this sequence has executed?

3.10 [20] <3.4, 3.7> Very long instruction word (VLIW) designers have a few basic choices to make regarding architectural rules for register use. Suppose a VLIW is designed with self-draining execution pipelines: once an operation is initiated, its results will appear in the destination register at most L cycles later (where L is the latency of the operation). There are never enough registers, so there is a temptation to wring maximum use out of the registers that exist. Consider Figure 3.52. If loads have a 1 + 2 cycle latency, unroll this loop once, and show how a VLIW capable of two loads and two adds per cycle can use the minimum number of registers, in the absence of any pipeline interruptions or stalls. Give an example of an event that, in the presence of self-draining pipelines, could disrupt this pipelining and yield wrong results.

3.11 [10/10/10] <3.3> Assume a five-stage single-pipeline microarchitecture (fetch, decode, execute, memory, write-back) and the code in Figure 3.53. All ops are one cycle except `LW` and `SW`, which are 1 + 2 cycles, and branches, which are 1 + 1 cycles. There is no forwarding. Show the phases of each instruction per clock cycle for one iteration of the loop.

a. [10] <3.3> How many clock cycles per loop iteration are lost to branch overhead?

b. [10] <3.3> Assume a static branch predictor, capable of recognizing a backward branch in the Decode stage. Now how many clock cycles are wasted on branch overhead?

```
Loop:   lw      x1,0(x2);     lw      x3,8(x2)
        <stall>
        <stall>
        addi    x10,x1,1;     addi    x11,x3,1
        sw      x1,0(x2);     sw      x3,8(x2)
        addi    x2,x2,8
        sub     x4,x3,x2
        bnz     x4,Loop
```

Figure 3.52 Sample VLIW code with two adds, two loads, and two stalls.

```
Loop:   lw x1,0(x2)
        addi    x1,x1, 1
        sw      x1,0(x2)
        addi    x2,x2,4
        sub     x4,x3,x2
        bnz     x4,Loop
```

Figure 3.53 Code loop for Exercise 3.11.

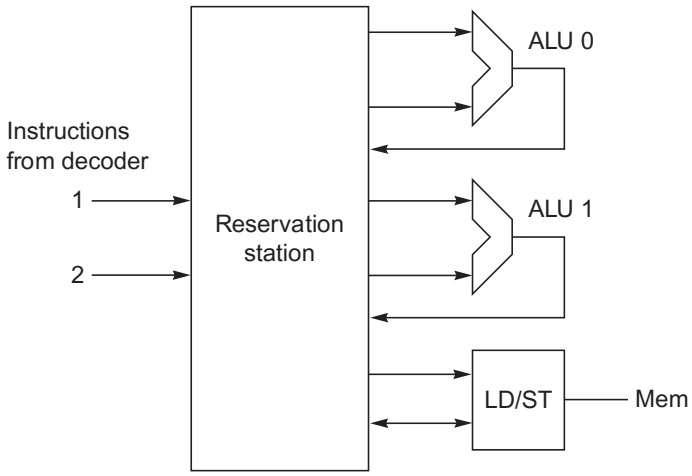

Figure 3.54 Microarchitecture for Exercise 3.12.

c. [10] <3.3> Assume a dynamic branch predictor. How many cycles are lost on a correct prediction?

3.12 [15/20/20/10/20] <3.4, 3.6> Let's consider what dynamic scheduling might achieve here. Assume a microarchitecture as shown in Figure 3.54. Assume that the arithmetic-logical units (ALUs) can do all arithmetic ops (`fmul.d`, `fdiv.d`, `fadd.d`, `addi`, `sub`) and branches, and that the Reservation Station (RS) can dispatch, at most, one operation to each functional unit per cycle (one op to each ALU plus one memory op to the `fld`/`fsd`).

a. [15] <3.4> Suppose all of the instructions from the sequence in Figure 3.47 are present in the RS, with no renaming having been done. Highlight any instructions in the code where register renaming would improve performance. (*Hint:* look for read-after-write and write-after-write hazards. Assume the same functional unit latencies as in Figure 3.47.)

b. [20] <3.4> Suppose the register-renamed version of the code from part (a) is resident in the RS in clock cycle *N*, with latencies as given in Figure 3.47. Show how the RS should dispatch these instructions out of order, clock by clock, to

obtain optimal performance on this code. (Assume the same RS restrictions as in part (a). Also assume that results must be written into the RS before they're available for use—no bypassing.) How many clock cycles does the code sequence take?

c. [20] <3.4> Part (b) lets the RS try to optimally schedule these instructions. But in reality, the whole instruction sequence of interest is not usually present in the RS. Instead, various events clear the RS, and as a new code sequence streams in from the decoder, the RS must choose to dispatch what it has. Suppose that the RS is empty. In cycle 0, the first two register-renamed instructions of this sequence appear in the RS. Assume it takes one clock cycle to dispatch any op, and assume functional unit latencies are as they were for Exercise 3.2. Further assume that the front end (decoder/register-renamer) will continue to supply two new instructions per clock cycle. Show the cycle-by-cycle order of dispatch of the RS. How many clock cycles does this code sequence require now?

d. [10] <3.4> If you wanted to improve the results of part (c), which would have helped most: (1) Another ALU? (2) Another LD/ST unit? (3) Full bypassing of ALU results to subsequent operations? or (4) Cutting the longest latency in half? What's the speedup?

e. [20] <3.6> Now let's consider speculation, the act of fetching, decoding, and executing beyond one or more conditional branches. Our motivation to do this is twofold: the dispatch schedule we came up with in part (c) had lots of nops, and we know computers spend most of their time executing loops (which implies the branch back to the top of the loop is pretty predictable). Loops tell us where to find more work to do; our sparse dispatch schedule suggests we have opportunities to do some of that work earlier than before. In part (d) you found the critical path through the loop. Imagine folding a second copy of that path onto the schedule you got in part (b). How many more clock cycles would be required to do two loops' worth of work (assuming all instructions are resident in the RS)? (Assume all functional units are fully pipelined.)

Exercises

3.13 [25] <3.7, 3.8> In this exercise, you will explore performance trade-offs between three processors that each employ different types of multithreading (MT). Each of these processors is superscalar, uses in-order pipelines, requires a fixed three-cycle stall following all loads and branches, and has identical L1 caches. Instructions from the same thread issued in the same cycle are read in program order and must not contain any data or control dependences.

- Processor A is a superscalar simultaneous MT architecture, capable of issuing up to two instructions per cycle from two threads.
- Processor B is a fine-grained MT architecture, capable of issuing up to four instructions per cycle from a single thread and switches threads on any pipeline stall.

- Processor C is a coarse-grained MT architecture, capable of issuing up to eight instructions per cycle from a single thread and switches threads on an L1 cache miss.

Our application is a list searcher, which scans a region of memory for a specific value stored in R9 between the address range specified in R16 and R17. It is parallelized by evenly dividing the search space into four equal-sized contiguous blocks and assigning one search thread to each block (yielding four threads). Most of each thread's runtime is spent in the following unrolled loop body:

```
loop: lw x1,0(x16)
      lw x2,8(x16)
      lw x3,16(x16)
      lw x4,24(x16)
      lw x5,32(x16)
      lw x6,40(x16)
      lw x7,48(x16)
      lw x8,56(x16)
      beq x9,x1,match0
      beq x9,x2,match1
      beq x9,x3,match2
      beq x9,x4,match3
      beq x9,x5,match4
      beq x9,x6,match5
      beq x9,x7,match6
      beq x9,x8,match7
      DADDIU x16,x16,#64
      blt x16,x17,loop
```

Assume the following:

- A barrier is used to ensure that all threads begin simultaneously.
- The first L1 cache miss occurs after two iterations of the loop.
- None of the BEQAL branches is taken.
- The BLT is always taken.
- All three processors schedule threads in a round-robin fashion.

Determine how many cycles are required for each processor to complete the first two iterations of the loop.

3.14 [25/25/25] <3.2, 3.7> In this exercise, we look at how software techniques can extract instruction-level parallelism (ILP) in a common vector loop. The following loop is the so-called DAXPY loop (double-precision aX plus Y) and is the central operation in Gaussian elimination. The following code implements the DAXPY operation, $Y = aX + Y$, for a vector length 100. Initially, R1 is set to the base address of array X and R2 is set to the base address of Y:

```
addi   x4,x1,#800 ; x1 = upper bound for X
```

```
foo: fld      F2,0(x1)   ;(F2) = X(i)
     fmul.d   F4,F2,F0   ;(F4) = a*X(i)
     fld      F6,0(x2)   ;(F6) = Y(i)
     fadd.d   F6,F4,F6   ;(F6) = a*X(i) + Y(i)
     fsd      F6,0(x2)   ;Y(i) = a*X(i) + Y(i)
     addi     x1,x1,#8   ;increment X index
     addi     x2,x2,#8   ;increment Y index
     sltu     x3,x1,x4   ;test: continue loop?
     bnez     x3,foo     ;loop if needed
```

Assume the functional unit latencies as shown in the following table. Assume a one-cycle delayed branch that resolves in the ID stage. Assume that results are fully bypassed.

Instruction producing result	Instruction using result	Latency in clock cycles
FP multiply	FP ALU op	6
FP add	FP ALU op	4
FP multiply	FP store	5
FP add	FP store	4
Integer operations and all loads	Any	2

a. [25] <3.2> Assume a single-issue pipeline. Show how the loop would look both unscheduled by the compiler and after compiler scheduling for both floating-point operation and branch delays, including any stalls or idle clock cycles. What is the execution time (in cycles) per element of the result vector, Y, unscheduled and scheduled? How much faster must the clock be for processor hardware alone to match the performance improvement achieved by the scheduling compiler? (Neglect any possible effects of increased clock speed on memory system performance.)

b. [25] <3.2> Assume a single-issue pipeline. Unroll the loop as many times as necessary to schedule it without any stalls, collapsing the loop overhead instructions. How many times must the loop be unrolled? Show the instruction schedule. What is the execution time per element of the result?

c. [25] <3.7> Assume a VLIW processor with instructions that contain five operations, as shown in Figure 3.20. We will compare two degrees of loop unrolling. First, unroll the loop 6 times to extract ILP and schedule it without any stalls (i.e., completely empty issue cycles), collapsing the loop overhead instructions, and then repeat the process but unroll the loop 10 times. Ignore the branch delay slot. Show the two schedules. What is the execution time per element of the result vector for each schedule? What percent of the operation slots are used in each schedule? How much does the size of the code differ between the two schedules? What is the total register demand for the two schedules?

3.15 [20/20] <3.4, 3.5, 3.7, 3.8> In this exercise, we will look at how variations on Tomasulo's algorithm perform when running the loop from Exercise 3.14. The functional units (FUs) are described in the following table.

FU type	Cycles in EX	Number of FUs	Number of reservation stations
Integer	1	1	5
FP adder	10	1	3
FP multiplier	15	1	2

Assume the following:

- Functional units are not pipelined.
- There is no forwarding between functional units; results are communicated by the common data bus (CDB).
- The execution stage (EX) does both the effective address calculation and the memory access for loads and stores. Thus, the pipeline is IF/ID/IS/EX/WB.
- Loads require one clock cycle.
- The issue (IS) and write-back (WB) result stages each require one clock cycle.
- There are five load buffer slots and five store buffer slots.
- Assume that the Branch on Not Equal to Zero (BNEZ) instruction requires one clock cycle.

a. [20] <3.4–3.5> For this problem use the single-issue Tomasulo MIPS pipeline of Figure 3.10 with the pipeline latencies from the preceding table. Show the number of stall cycles for each instruction and what clock cycle each instruction begins execution (i.e., enters its first EX cycle) for three iterations of the loop. How many cycles does each loop iteration take? Report your answer in the form of a table with the following column headers:
   - Iteration (loop iteration number)
   - Instruction
   - Issues (cycle when instruction issues)
   - Executes (cycle when instruction executes)
   - Memory access (cycle when memory is accessed)
   - Write CDB (cycle when result is written to the CDB)
   - Comment (description of any event on which the instruction is waiting)

   Show three iterations of the loop in your table. You may ignore the first instruction.

b. [20] <3.7, 3.8> Repeat part (a) but this time assume a two-issue Tomasulo algorithm and a fully pipelined floating-point unit (FPU).

3.16 [10] <3.4> Tomasulo's algorithm has a disadvantage: only one result can compute per clock per CDB. Use the hardware configuration and latencies from the previous question and find a code sequence of no more than 10 instructions where Tomasulo's algorithm must stall due to CDB contention. Indicate where this occurs in your sequence.

3.17 [20] <3.3> An (*m*,*n*) correlating branch predictor uses the behavior of the most recent *m* executed branches to choose from 2^m predictors, each of which is an *n*-bit predictor. A two-level local predictor works in a similar fashion, but only keeps track of the past behavior of each individual branch to predict future behavior.

There is a design trade-off involved with such predictors: correlating predictors require little memory for history, which allows them to maintain 2-bit predictors for a large number of individual branches (reducing the probability of branch instructions reusing the same predictor), while local predictors require substantially more memory to keep history and are thus limited to tracking a relatively small number of branch instructions. For this exercise, consider a (1,2) correlating predictor that can track four branches (requiring 16 bits) versus a (1,2) local predictor that can track two branches using the same amount of memory. For the following branch outcomes, provide each prediction, the table entry used to make the prediction, any updates to the table as a result of the prediction, and the final misprediction rate of each predictor. Assume that all branches up to this point have been taken. Initialize each predictor to the following:

Correlating predictor

Entry	Branch	Last outcome	Prediction
0	0	T	T with one misprediction
1	0	NT	NT
2	1	T	NT
3	1	NT	T
4	2	T	T
5	2	NT	T
6	3	T	NT with one misprediction
7	3	NT	NT

Local predictor

Entry	Branch	Last 2 outcomes (right is most recent)	Prediction
0	0	T,T	T with one misprediction
1	0	T,NT	NT
2	0	NT,T	NT
3	0	NT	T
4	1	T,T	T
5	1	T,NT	T with one misprediction
6	1	NT,T	NT
7	1	NT,NT	NT

Branch PC (word address)	Outcome
454	T
543	NT
777	NT
543	NT
777	NT
454	T
777	NT
454	T
543	T

3.18 [10] <3.9> 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 four cycles and the buffer miss penalty is always three cycles. Assume a 90% hit rate, 90% accuracy, and 15% branch frequency. How much faster is the processor with the branch-target buffer versus a processor that has a fixed two-cycle branch penalty? Assume a base clock cycle per instruction (CPI) without branch stalls of one.

3.19 [10/5] <3.9> Consider a branch-target buffer that has penalties of zero, two, and two clock cycles for correct conditional branch prediction, incorrect prediction, and a buffer miss, respectively. Consider a branch-target buffer design that distinguishes conditional and unconditional branches, storing the target address for a conditional branch and the target instruction for an unconditional branch.

a. [10] <3.9> What is the penalty in clock cycles when an unconditional branch is found in the buffer?

b. [10] <3.9> Determine the improvement from branch folding for unconditional branches. Assume a 90% hit rate, an unconditional branch frequency of 5%, and a two-cycle penalty for a buffer miss. How much improvement is gained by this enhancement? How high must the hit rate be for this enhancement to provide a performance gain?