H&P 3e

4

Exploiting Instruction-Level Parallelism with Software Approaches

Processors are being produced with the potential for very many parallel operations on the instruction level.... Far greater extremes in instruction-level parallelism are on the horizon.

J. Fisher
[1981], in the paper that inaugurated the term "instruction-level parallelism"

One of the surprises about IA-64 is that we hear no claims of high frequency, despite claims that an EPIC processor is less complex than a superscalar processor. It's hard to know why this is so, but one can speculate that the overall complexity involved in focusing on CPI, as IA-64 does, makes it hard to get high megahertz.

M. Hopkins
[2000], in a commentary on the IA-64 architecture, a joint development of HP and Intel designed to achieve dramatic increases in the exploitation of ILP while retaining a simple architecture, which would allow higher performance

4.1 Basic Compiler Techniques for Exposing ILP

This chapter starts by examining the use of compiler technology to improve the performance of pipelines and simple multiple-issue processors. These techniques are crucial for processors that use static issue, and they are often important even for processors that make dynamic issue decisions but use static scheduling. After applying these concepts to reducing stalls from data hazards in single-issue pipelines, we examine the use of compiler-based techniques for branch prediction. Armed with this more powerful compiler technology, we examine the design and performance of multiple-issue processors using static issuing or scheduling. Sections 4.4 and 4.5 examine more advanced software and hardware techniques, designed to enable a processor to exploit more instruction-level parallelism. Section 4.7, "Putting It All Together," examines the IA-64 architecture and its first implementation, Itanium. Two different static, VLIW-style processors are covered in Section 4.8, "Another View."

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, 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. Throughout this chapter we will assume the FP unit latencies shown in Figure 4.1, 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.

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 4.1 Latencies of FP operations used in this chapter. The first column shows the originating instruction type. The second column is the type of the consuming instruction. 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 zero, since 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.

In this subsection, we look at how the compiler can increase the amount of available ILP by unrolling loops. This example serves both to illustrate an important technique as well as to motivate the more powerful program transformations described later in this chapter. We will rely on an example similar to the one we used in the last chapter, adding a scalar to a vector:

```
for (i=1000; i>0; i=i-1)
      x[i] = x[i] + s;
```

We can see that this loop is parallel by noticing that the body of each iteration is independent. We will formalize this notion later in this chapter and describe how we can test whether loop iterations are independent at compile time. First, let's look at the performance of this loop, showing how we can use the parallelism to improve its performance for a MIPS pipeline with the latencies shown above.

The first step is to translate the above segment to MIPS assembly language. In the following code segment, R1 is initially the address of the element in the array with the highest address, and F2 contains the scalar value s. Register R2 is precomputed, so that 8(R2) is the last element to operate on.

The straightforward MIPS code, not scheduled for the pipeline, looks like this:

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

Let's start by seeing how well this loop will run when it is scheduled on a simple pipeline for MIPS with the latencies from Figure 4.1.

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

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

```
                                        Clock cycle issued
Loop:  L.D     F0,0(R1)          1
       stall                     2
       ADD.D   F4,F0,F2          3
       stall                     4
       stall                     5
       S.D     F4,0(R1)          6
       DADDUI  R1,R1,#-8         7
       stall                     8
       BNE     R1,R2,Loop        9
       stall                    10
```

This code requires 10 clock cycles per iteration. We can schedule the loop to obtain only one stall:

```
Loop:  L.D      F0,0(R1)
       DADDUI   R1,R1,#-8
       ADD.D    F4,F0,F2
       stall
       BNE      R1,R2,Loop ;delayed branch
       S.D      F4,8(R1)   ;altered & interchanged with DADDUI
```

Execution time has been reduced from 10 clock cycles to 6. The stall after ADD.D is for use by the S.D.

Notice that to schedule the delayed branch, the compiler had to determine that it could swap the DADDUI and S.D by changing the address to which the S.D stored: The address was 0(R1) and is now 8(R1). This change is not trivial, since most compilers would see that the S.D instruction depends on the DADDUI and would refuse to interchange them. A smarter compiler, capable of limited symbolic optimization, could figure out the relationship and perform the interchange. The chain of dependent instructions from the L.D to the ADD.D and then to the S.D determines the clock cycle count for this loop. This chain must take at least 6 cycles because of dependencies and pipeline latencies.

In the above example, we complete one loop iteration and store back one array element every 6 clock cycles, but the actual work of operating on the array element takes just 3 (the load, add, and store) of those 6 clock cycles. The remaining 3 clock cycles consist of loop overhead—the DADDUI and BNE—and a stall. To eliminate these 3 clock cycles we need to get more operations within the loop 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 stall 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 register count.

Example Show our loop unrolled so that there are four copies of the loop body, assuming R1 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 DADDUI instructions and dropping the unnecessary BNE operations that are duplicated during unrolling. Note that R2 must now be set so that 32(R2) is the starting address of the last four elements.


```
Loop:  L.D      F0,0(R1)
       ADD.D    F4,F0,F2
       S.D      F4,0(R1)      ;drop DADDUI & BNE
       L.D      F6,-8(R1)
       ADD.D    F8,F6,F2
       S.D      F8,-8(R1)     ;drop DADDUI & BNE
       L.D      F10,-16(R1)
       ADD.D    F12,F10,F2
       S.D      F12,-16(R1)   ;drop DADDUI & BNE
       L.D      F14,-24(R1)
       ADD.D    F16,F14,F2
       S.D      F16,-24(R1)
       DADDUI   R1,R1,#-32
       BNE      R1,R2,Loop
```

We have eliminated three branches and three decrements of R1. The addresses on the loads and stores have been compensated to allow the DADDUI instructions on R1 to be merged. This optimization may seem trivial, but it is not; it requires symbolic substitution and simplification. We will see more general forms of these optimizations that eliminate dependent computations in Section 4.4.

Without scheduling, every operation in the unrolled loop is followed by a dependent operation and thus will cause a stall. This loop will run in 28 clock cycles—each L.D has 1 stall, each ADD.D 2, the DADDUI 1, the branch 1, plus 14 instruction issue cycles—or 7 clock cycles for each of the four elements. Although this unrolled version is currently slower than the *scheduled* version of the original loop, this will change when we schedule the unrolled loop. 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 would like 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. 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 shown in Figure 4.1.

Answer

```
Loop:  L.D      F0,0(R1)
       L.D      F6,-8(R1)
       L.D      F10,-16(R1)
       L.D      F14,-24(R1)
       ADD.D    F4,F0,F2
       ADD.D    F8,F6,F2
       ADD.D    F12,F10,F2
       ADD.D    F16,F14,F2
       S.D      F4,0(R1)
       S.D      F8,-8(R1)
       DADDUI   R1,R1,#-32
       S.D      F12,16(R1)
       BNE      R1,R2,Loop
       S.D      F16,8(R1);8-32 = -24
```

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 7 cycles per element before scheduling and 6 cycles when scheduled but not unrolled.

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 code above 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 Example

Throughout this chapter 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:

1. Determine that it was legal to move the `S.D` after the `DADDUI` and `BNE`, and find the amount to adjust the `S.D` offset.
2. Determine that unrolling the loop would be useful by finding that the loop iterations were independent, except for the loop maintenance code.
3. Use different registers to avoid unnecessary constraints that would be forced by using the same registers for different computations.
4. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code.

5. 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.
6. 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 an instruction depends on another and how the instructions can be changed or reordered given the dependences. Before examining how these techniques work for higher issue rate pipelines, let us examine how the loop unrolling and scheduling techniques affect data dependences.

Example Show how the process of optimizing the loop overhead by unrolling the loop actually eliminates data dependences. In this example and those used in the remainder of this chapter, we use nondelayed branches for simplicity; it is easy to extend the examples to use delayed branches.

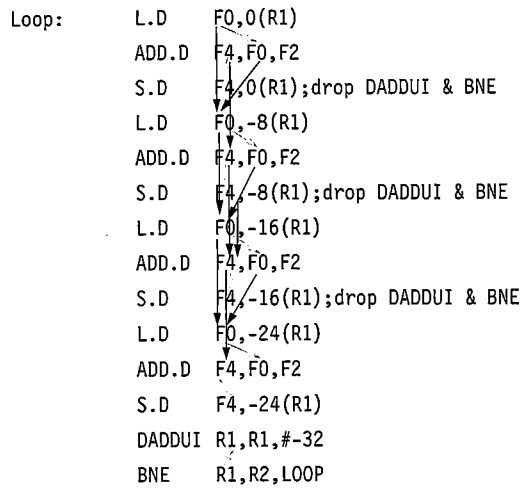
Answer Here is the unrolled but unoptimized code with the extra `DADDUI` instructions, but without the branches. (Eliminating the branches is another type of transformation, since it involves control rather than data.) The arrows show the data dependences that are within the unrolled body and involve the `DADDUI` instructions. The underlined registers are the dependent uses.

```
Loop:  L.D      F0,0(R1)
       ADD.D    F4,F0,F2
       S.D      F4,0(R1)
       DADDUI   R1,R1,#-8;drop BNE
       L.D      F6,0(R1)
       ADD.D    F8,F6,F2
       S.D      F8,0(R1)
       DADDUI   R1,R1,#-8;drop BNE
       L.D      F10,0(R1)
       ADD.D    F12,F10,F2
       S.D      F12,0(R1)
       DADDUI   R1,R1,#-8;drop BNE
       L.D      F14,0(R1)
       ADD.D    F16,F14,F2
       S.D      F16,0(R1)
       DADDUI   R1,R1,#-8
       BNE      R1,R2,LOOP
```

As the arrows show, the DADDUI instructions form a dependent chain that involves the DADDUI, L.D, and S.D instructions. This chain forces the body to execute in order, as well as making the DADDUI instructions necessary, which increases the instruction count. The compiler removes this dependence by symbolically computing the intermediate values of R1 and folding the computation into the offset of the L.D and S.D instructions and by changing the final DADDUI into a decrement by 32. This transformation makes the three DADDUI unnecessary, and the compiler can remove them. There are other types of dependences in this code, as the next few examples show.

Example Unroll our example loop, eliminating the excess loop overhead, but using the same registers in each loop copy. Indicate both the data and name dependences within the body. Show how renaming eliminates name dependences that reduce parallelism.

Answer Here's the loop unrolled but with the same registers in use for each copy. The data dependences are shown with gray arrows and the name dependences with black arrows. As in earlier examples, the direction of the arrow indicates the ordering that must be preserved for correct execution of the code:



The name dependences force the instructions in the loop to be almost completely ordered, allowing only the order of the L.D following each S.D to be interchanged. When the registers used for each copy of the loop body are renamed, only the true dependences within each body remain:


```
Loop:   L.D     F0,0(R1)
        ADD.D   F4,F0,F2
        S.D     F4,0(R1);drop DADDUI & BNE
        L.D     F6,-8(R1)
        ADD.D   F8,F6,F2
        S.D     F8,-8(R1);drop DADDUI & BNE
        L.D     F10,-16(R1)
        ADD.D   F12,F10,F2
        S.D     F12,-16(R1);drop DADDUI & BNE
        L.D     F14,-24(R1)
        ADD.D   F16,F14,F2
        S.D     F16,-24(R1)
        DADDUI  R1,R1,#-32
        BNE     R1,R2,LOOP
```

With the renaming, the copies of each loop body become independent and can be overlapped or executed in parallel. This renaming process can be performed either by the compiler or in hardware, as we saw in the last chapter.

There are three different types of limits to the gains that can be achieved by loop unrolling: a decrease in the amount of overhead amortized with each unroll, code size limitations, and 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 DSUBI, 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 original iteration to 1/4. Exercise 4.3 asks you to compute the theoretically optimal number of times to unroll this loop for a random number of iterations.

A second limit to unrolling is the growth in code size that results. For larger loops, the code size growth may be a concern either in the embedded space where memory may be at a premium or if the larger code size causes a decrease in the instruction cache miss rate. We return to the issue of code size when we consider more aggressive techniques for uncovering instruction-level parallelism in Section 4.4.

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 generates 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 machines 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 hard 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 in MIPS to the statically scheduled superscalars we described in the last chapter, as we will see now.

Using Loop Unrolling and Pipeline Scheduling with Static Multiple Issue

We begin by looking at a simple two-issue, statically scheduled superscalar MIPS pipeline from the last chapter, using the pipeline latencies from Figure 4.1 and the same example code segment we used for the single-issue examples above. This processor can issue two instructions per clock cycle: One of the instructions can be a load, store, branch, or integer ALU operation, and the other can be any floating-point operation.

Recall that this pipeline did not generate a significant performance enhancement for the previous example because of the limited ILP in a given loop iteration. Let's see how loop unrolling and pipeline scheduling can help.

Example Unroll and schedule the loop used in the earlier examples and shown on page 305.

Answer To schedule this loop without any delays, we will need to unroll the loop to make five copies of the body. After unrolling, the loop will contain five each of `L.D`, `ADD.D`, and `S.D`; one `DADDUI`; and one `BNE`. The unrolled and scheduled code is shown in Figure 4.2.

This unrolled superscalar loop now runs in 12 clock cycles per iteration, or 2.4 clock cycles per element, versus 3.5 for the scheduled and unrolled loop on the ordinary MIPS pipeline. In this example, the performance of the superscalar MIPS is limited by the balance between integer and floating-point computation. Every floating-point instruction is issued together with an integer instruction, but there are not enough floating-point instructions to keep the floating-point pipeline full. When scheduled, the original loop ran in 6 clock cycles per iteration. We have improved on that by a factor of 2.5, more than half of which came from loop unrolling. Loop unrolling took us from 6 to 3.5 (a factor of 1.7), while superscalar execution gave us a factor of 1.5 improvement.

	Integer instruction		FP instruction	Clock cycle
Loop:	L.D	F0,0(R1)		1
	L.D	F6,-8(R1)		2
	L.D	F10,-16(R1)	ADD.D F4,F0,F2	3
	L.D	F14,-24(R1)	ADD.D F8,F6,F2	4
	L.D	F18,-32(R1)	ADD.D F12,F10,F2	5
	S.D	F4,0(R1)	ADD.D F16,F14,F2	6
	S.D	F8,-8(R1)	ADD.D F20,F18,F2	7
	S.D	F12,-16(R1)		8
	DADDUI	R1,R1,#-40		9
	S.D	F16,16(R1)		10
	BNE	R1,R2,Loop		11
	S.D	F20,8(R1)		12

Figure 4.2 The unrolled and scheduled code as it would look on a superscalar MIPS.

4.2 Static Branch Prediction

In Chapter 3, we examined the use of dynamic branch predictors. Static branch predictors are sometimes used in processors where the expectation is that branch behavior is highly predictable at compile time; static prediction can also be used to assist dynamic predictors.

In Appendix A, we will discuss an architectural feature that supports static branch predication, namely, delayed branches. Delayed branches expose a pipeline hazard so that the compiler can reduce the penalty associated with the hazard. As we saw, the effectiveness of this technique partly depends on whether we correctly guess which way a branch will go. Being able to accurately predict a branch at compile time is also helpful for scheduling data hazards. Loop unrolling is one simple example of this; another example arises from conditional selection branches. Consider the following code segment:

```
        LD      R1,0(R2)
        DSUBU   R1,R1,R3
        BEQZ    R1,L
        OR      R4,R5,R6
        DADDU   R10,R4,R3
L:      DADDU   R7,R8,R9
```

The dependence of the `DSUBU` and `BEQZ` on the `LD` instruction means that a stall will be needed after the `LD`. Suppose we knew that this branch was almost always taken and that the value of R7 was not needed on the fall-through path. Then we could increase the speed of the program by moving the instruction `DADD R7,R8,R9` to the position after the `LD`. Correspondingly, if we knew the branch

was rarely taken and that the value of R4 was not needed on the taken path, then we could contemplate moving the OR instruction after the LD. In addition, we can also use the information to better schedule any branch delay, since choosing how to schedule the delay depends on knowing the branch behavior. We will return to this topic in Section 4.4, when we discuss global code scheduling.

To perform these optimizations, we need to predict the branch statically when we compile the program. There are several different methods to statically predict branch behavior. The simplest scheme is to predict a branch as taken. This scheme has an average misprediction rate that is equal to the untaken branch frequency, which for the SPEC programs is 34%. Unfortunately, the misprediction rate ranges from not very accurate (59%) to highly accurate (9%).

A better alternative is to predict on the basis of branch direction, choosing backward-going branches to be taken and forward-going branches to be not taken. For some programs and compilation systems, the frequency of forward taken branches may be significantly less than 50%, and this scheme will do better than just predicting all branches as taken. In the SPEC programs, however, more than half of the forward-going branches are taken. Hence, predicting all branches as taken is the better approach. Even for other benchmarks or compilers, direction-based prediction is unlikely to generate an overall misprediction rate of less than 30% to 40%. An enhancement of this technique was explored by Ball and Larus [1993]; their approach uses program context information and generates more accurate predictions than a simple scheme based solely on branch direction.

A still more accurate technique is to predict branches on the basis of profile information collected from earlier runs. The key observation that makes this worthwhile is that the behavior of branches is often bimodally distributed; that is, an individual branch is often highly biased toward taken or untaken. Figure 4.3 shows the success of branch prediction using this strategy.The same input data were used for runs and for collecting the profile; other studies have shown that changing the input so that the profile is for a different run leads to only a small change in the accuracy of profile-based prediction.

Although we can derive the prediction accuracy of a predicted-taken strategy and measure the accuracy of the profile scheme, as in Figure 4.3, the wide range of frequency of conditional branches in these programs, from 3% to 24%, means that the overall frequency of a mispredicted branch varies widely. Figure 4.4 shows the number of instructions executed between mispredicted branches for both a profile-based and a predicted-taken strategy. The number varies widely, both because of the variation in accuracy and the variation in branch frequency. On average, the predicted-taken strategy has 20 instructions per mispredicted branch and the profile-based strategy has 110. These averages, however, are very different for integer and FP programs, as the data in Figure 4.4 show.

Static branch behavior is useful for scheduling instructions when the branch delays are exposed by the architecture (either delayed or canceling branches), for assisting dynamic predictors (as we will see in the IA-64 architecture in Section 4.7), and for determining which code paths are more frequent, which is a key step in code scheduling (see Section 4.4, page 332).

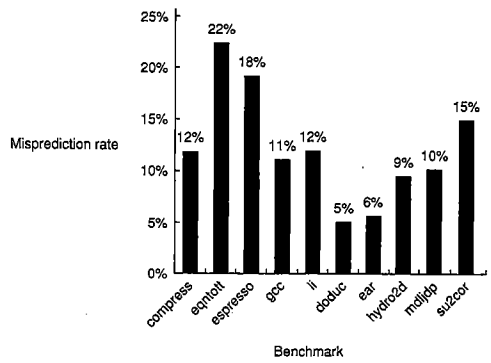


Figure 4.3 Misprediction rate on SPEC92 for a profile-based predictor varies widely but is generally better for the FP programs, which have an average misprediction rate of 9% with a standard deviation of 4%, than for the integer programs, which have an average misprediction rate of 15% with a standard deviation of 5%. The actual performance depends on both the prediction accuracy and the branch frequency, which varies from 3% to 24%; we will examine the combined effect in Figure 4.4.

4.3 Static Multiple Issue: The VLIW Approach

Superscalar processors decide on the fly how many instructions to issue. A statically scheduled superscalar must check for any dependences between instructions in the issue packet as well as between any issue candidate and any instruction already in the pipeline. As we have seen in Section 4.1, a statically scheduled superscalar requires significant compiler assistance to achieve good performance. In contrast, a dynamically scheduled superscalar requires less compiler assistance, but has significant hardware costs.

An alternative to the superscalar approach is to rely on compiler technology not only to minimize the potential data hazard stalls, but to actually format the instructions in a potential issue packet so that the hardware need not check explicitly for dependences. The compiler may be required to ensure that dependences within the issue packet cannot be present or, at a minimum, indicate when a dependence may be present. Such an approach offers the potential advantage of simpler hardware while still exhibiting good performance through extensive compiler optimization.

The first multiple-issue processors that required the instruction stream to be explicitly organized to avoid dependences used wide instructions with multiple operations per instruction. For this reason, this architectural approach was named VLIW (very long instruction word), denoting that the instructions, since they contained several instructions, were very wide (64 to 128 bits, or more). The basic architectural concepts and compiler technology are the same whether

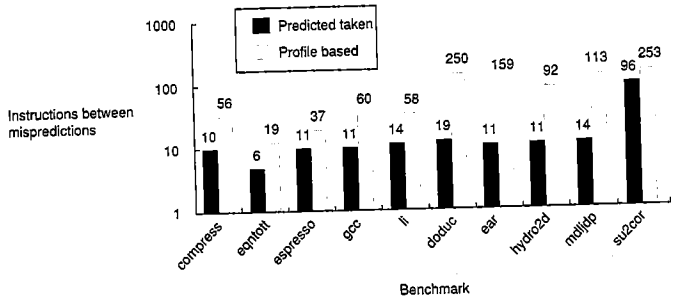


Figure 4.4 Accuracy of a predicted-taken strategy and a profile-based predictor for SPEC92 benchmarks as measured by the number of instructions executed between mispredicted branches and shown on a log scale. The average number of instructions between mispredictions is 20 for the predicted-taken strategy and 110 for the profile-based prediction; however, the standard deviations are large: 27 instructions for the predicted-taken strategy and 85 instructions for the profile-based scheme. This wide variation arises because programs such as su2cor have both low conditional branch frequency (3%) and predictable branches (85% accuracy for profiling), although eqntott has eight times the branch frequency with branches that are nearly 1.5 times *less* predictable. The difference between the FP and integer benchmarks as groups is large. For the predicted-taken strategy, the average distance between mispredictions for the integer benchmarks is 10 instructions, and for the FP programs it is 30 instructions. With the profile scheme, the distance between mispredictions for the integer benchmarks is 46 instructions, and for the FP benchmarks it is 173 instructions.

multiple operations are organized into a single instruction, or whether a set of instructions in an issue packet is preconfigured by a compiler to exclude dependent operations (since the issue packet can be thought of as a very large instruction). Early VLIWs were quite rigid in their instruction formats and effectively required recompilation of programs for different versions of the hardware.

To reduce this inflexibility and enhance the performance of the approach, several innovations have been incorporated into more recent architectures of this type, while still requiring the compiler to do most of the work of finding and scheduling instructions for parallel execution. This second generation of VLIW architectures is the approach being pursued for desktop and server markets.

In the remainder of this section, we look at the basic concepts in a VLIW architecture. Section 4.4 introduces additional compiler techniques that are required to achieve good performance for compiler-intensive approaches, and Section 4.5 describes hardware innovations that improve flexibility and performance of explicitly parallel approaches. Finally, Section 4.7 describes how the Intel IA-64 supports explicit parallelism.

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 issue packet satisfy the same constraints. Since 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. Since the burden for choosing the instructions to be issued simultaneously falls on the compiler, the hardware in a superscalar to make these issue decisions is unneeded.

Since this 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 saw in the last chapter, this overhead grows with issue width.

Because VLIW approaches make sense for wider processors, we choose to focus our example on such an architecture. For example, a VLIW processor might have 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 to 24 bits per unit, yielding an instruction length of between 112 and 168 bits.

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 requires 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, since moving code across branches is expensive. In the next section, we will discuss *trace scheduling*, one of these global scheduling techniques developed specifically for VLIWs. In Section 4.5, we will examine 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, let's assume we have a technique to generate long, straight-line code sequences, so that we can use local scheduling to build up VLIW instructions and instead 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 305 for the MIPS code) for such a processor. Unroll as many times as necessary to eliminate any stalls. Ignore the branch delay slot.

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

Figure 4.5 VLIW instructions that occupy the inner loop and replace the unrolled sequence. This code takes 9 cycles assuming no branch delay; normally the branch delay would also need to be scheduled. 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 MIPS would normally use in this loop. The VLIW code sequence above requires at least eight FP registers, while the same code sequence for the base MIPS processor can use as few as two FP registers or as many as five when unrolled and scheduled. In the superscalar example in Figure 4.2, six registers were needed.

Answer The code is shown in Figure 4.5. 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. 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 4.1 that used unrolled and scheduled code.

For the original VLIW model, there are both technical and logistical problems. The technical problems are the increase in code size and the limitations of 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 Figure 4.5, we saw that only about 60% of the functional units were used, so almost half of each instruction was empty. In most VLIWs, an instruction may need to be left completely empty if no operations can be scheduled.

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. We will see techniques to reduce code size increases in both Sections 4.7 and 4.8.

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, since all the functional units must be kept synchronized. Although a compiler may be able to schedule the deterministic functional units to prevent stalls, predicting which data accesses will encounter a cache stall and scheduling them is very difficult. Hence, caches needed to be blocking and to cause *all* the functional units to stall. As the issue rate and number of memory references becomes large, this synchronization restriction becomes 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 VLIWs. 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.

One possible solution to this migration problem, and the problem of binary code compatibility in general, is object-code translation or emulation. This technology is developing quickly and could play a significant role in future migration schemes. Another approach is to temper the strictness of the approach so that binary compatibility is still feasible. This later approach is used in the IA-64 architecture, as we will see in Section 4.7.

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 Appendix G). 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 twofold. First, a multiple-issue processor has the potential to extract some amount of parallelism from less regularly structured code, and, second, it has the ability to use a more conventional, and typically less expensive, cache-based memory system. 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.

4.4 Advanced Compiler Support for Exposing and Exploiting ILP

In this section we discuss compiler technology for increasing the amount of parallelism that we can exploit in a program. We begin by defining when a loop is parallel and how a dependence can prevent a loop from being parallel. We also

tural Support for Programming Languages and Operating Systems (October), Boston, IEEE/ACM, 238–247.

Mahlke, S. A., R. E. Hank, J. E. McCormick, D. I. August, and W. W. Hwu [1995]. "A comparison of full and partial predicated execution support for ILP processors," *Proc. 22nd Annual Int'l Symposium on Computer Architecture* (June), Santa Margherita Ligure, Italy, 138–149.

McFarling, S., and J. Hennessy [1986]. "Reducing the cost of branches," *Proc. 13th Symposium on Computer Architecture* (June), Tokyo, 396–403.

Nicolau, A., and J. A. Fisher [1984]. "Measuring the parallelism available for very long instruction word architectures," *IEEE Trans. on Computers* C-33:11 (November), 968–976.

Rau, B. R. [1994]. "Iterative modulo scheduling: An algorithm for software pipelining loops," *Proc. 27th Annual Int'l Symposium on Microarchitecture* (November), San Jose, Calif., 63–74.

Rau, B. R., C. D. Glaeser, and R. L. Picard [1982]. "Efficient code generation for horizontal architectures: Compiler techniques and architectural support," *Proc. Ninth Symposium on Computer Architecture* (April), 131–139.

Rau, B. R., D. W. L. Yen, W. Yen, and R. A. Towle [1989]. "The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs," *IEEE Computers* 22:1 (January), 12–34.

Riseman, E. M., and C. C. Foster [1972]. "Percolation of code to enhance parallel dispatching and execution," *IEEE Trans. on Computers* C-21:12 (December), 1411–1415.

Smith, M. D., M. Horowitz, and M. S. Lam [1992]. "Efficient superscalar performance through boosting," *Proc. Fifth Conf. on Architectural Support for Programming Languages and Operating Systems* (October), Boston, IEEE/ACM, 248–259.

Thorlin, J. F. [1967]. "Code generation for PIE (parallel instruction execution) computers," *Proc. Spring Joint Computer Conf.*, 27.

Weiss, S., and J. E. Smith [1987]. "A study of scalar compilation techniques for pipelined supercomputers," *Proc. Second Conf. on Architectural Support for Programming Languages and Operating Systems* (March), IEEE/ACM, Palo Alto, Calif., 105–109.

Wilson, R. P., and M. S. Lam [1995]. "Efficient context-sensitive pointer analysis for C programs," *Proc. ACM SIGPLAN'95 Conf. on Programming Language Design and Implementation*, La Jolla, Calif., June, 1–12.

Exercises

Solutions to the "starred" exercises appear in Appendix B.

✪ 4.1 [15/10] <4.1, A.4> If we assume the set of latencies in Figure 4.1 and that a result can always be forwarded, then a specific structure for some of the CPU pipeline is implied. Assume the CPU uses the standard five-stage IF/ID/EX/Mem/WB pipeline.

a. [15] <4.1, A.4> Using a style similar to that of Figures A.23 and A.31 in Appendix A, draw a block diagram showing only the implied portions of the pipeline. Label each component and data path in your diagram and show the number of clock cycles each functional unit requires.

b. [10] <4.1, A.4> For each functional unit in your part (a) diagram, which row(s) of Figure 4.1 provide the information to determine the number of clock cycles needed for that functional unit to complete its operation?

4.2 [15/15] <4.1> This chapter examines software approaches for exploiting instruction-level parallelism. This exercise asks how well software can find and exploit ILP. Chapter 3 presents hardware techniques for exposing and exploiting ILP. Exercise 3.2 is a reprise of this ILP analysis, but from a hardware perspective.

Consider the following four MIPS code fragments, each containing two instructions:

```
i.      DADDI R1,R1,#4
        LD R2,7(R1)

ii.     DADD R3,R1,R2
        SD 7(R1),R2

iii.    SD 7(R1),R2
        S.D 200(R7),F2

iv.     BEZ R1,place
        SD 7(R1),R1
```

a. [15] <4.1> For each code fragment (i)–(iv) identify each type of dependence that a compiler will find (a fragment may have no dependences) and describe what data flow, name reuse, or control structure causes the dependence.

b. [15] <4.1> Assuming nonspeculative execution, for each code fragment discuss whether a compiler could schedule the two instructions.

4.3 [20/15/15] <2.12, 4.1> Consider the simple loop in Section 4.1. Assume the number of iterations is unknown, but large.

a. [20] <4.1> Find the theoretically optimal number of unrollings using the latencies in Figure 4.1. *Hint:* Recall that you will need two loops: one unrolled and one not!

b. [15] <2.12, 4.1> What is the actual maximum number of times the simple loop in Section 4.1 can be unrolled using the given MIPS code? What is the limiting resource? Show how to increase the number of times the loop may be unrolled by transforming the MIPS code to make less intensive use of the limiting resource. How much does this transformation improve performance?

c. [15] <2.12, 4.1> For the MIPS instruction set, what additional parameters limit the number of times this loop can be unrolled? *Hint:* When you find one limiting parameter, assume that the resource it defines is unlimited, look for an additional parameter, and repeat as needed.

4.4 [15] <4.1> Section 4.1 presents a technique for unrolling loops where the unrolling factor is not statically known to be a factor of the number of loop iterations. For a factor of k, the technique constructs two consecutive loops that iterate (n mod k) and (n/k) times, respectively. Find a technique to use just a single loop containing the unrolled body iterated $\lceil n/k \rceil$ times. What restrictions are there on the use of this technique? When does this technique perform better than the general, two-consecutive-loops technique? Can a compiler employ this technique?

4.5 [15] <4.1> List all the dependences (output, anti, and true) in the following code fragment. Indicate whether the true dependences are loop carried or not. Show why the loop is not parallel.

```
for (i=2;i<100;i=i+1) {
      a[i] = b[i] + a[i];   /* S1 */
      c[i-1] = a[i] + d[i]; /* S2 */
      a[i-1] = 2 * b[i];    /* S3 */
      b[i+1] = 2 * b[i];    /* S4 */
}
```

4.6 [15] <4.1> Here is an unusual loop. First, list the dependences and then rewrite the loop so that it is parallel.

```
for (i=1;i<100;i=i+1) {
      a[i] = b[i] + c[i];    /* S1 */
      b[i] = a[i] + d[i];    /* S2 */
      a[i+1] = a[i] + e[i];  /* S3 */
}
```

4.7 [20/12] <4.1> The following loop is a dot product (assuming the running sum in F2 is initially 0) and contains a recurrence. Assume the pipeline latencies from Figure 4.1 and a 1-cycle delayed branch.

```
foo:    L.D       F0,0(R1)       ;load X[i]
        L.D       F4,0(R2)       ;load Y[i]
        MUL.D     F0,F0,F4       ;multiply X[i]*Y[i]
        ADD.D     F2,F0,F2       ;add sum = sum + X[i]*Y[i]
        DADDUI    R1,R1,#-8      ;decrement X index
        DADDUI    R2,R2,#-8      ;decrement Y index
        BNEZ      R1,foo         ;loop if not done
```

a. [20] <4.1> Assume a single-issue pipeline. Despite the fact that the loop is not parallel, it can be scheduled with no delays. Unroll the following loop a sufficient number of times to schedule it without any delays. Show the schedule after eliminating any redundant overhead instructions. *Hint:* An additional transformation of the code is needed to schedule without delay.

b. [12] <4.1> Show the schedule of the transformed code from part (a) for the processor in Figure 4.2. For an issue capability that is 100% greater, how much faster is the loop body?

4.8 [15/15] <4.1> The following loop computes Y[i] = a × X[i] + Y[i], the key step in a Gaussian elimination. Assume the pipeline latencies from Figure 4.1 and a 1-cycle delayed branch.

```
loop:   L.D       F0,0(R1)       ;load X[i]
        MUL.D     F0,F0,F2       ;multiply a*X[i]
        L.D       F4,0(R2)       ;load Y[i]
        ADD.D     F0,F0,F4       ;add a*X[i]+Y[i]
        S.D       0(R2),F0       ;store Y[i]
```



```
        DSUBUI    R1,R1,#8       ;decrement X index
        DSUBUI    R2,R2,#8       ;decrement Y index
        BNEZ      R1,loop        ;loop if not done
```

a. [15] <4.1> Assume a single-issue pipeline. Unroll the loop as many times as necessary to schedule it without any delays, collapsing the loop overhead instructions. Show the schedule. What is the execution time per element?

b. [15] <4.1> Assume a dual-issue processor as in Figure 4.2. Unroll the loop as many times as necessary to schedule it without any delays, collapsing the loop overhead instructions. Show the schedule. What is the execution time per element? How many instruction issue slots are unused?

4.9 [20/15/20/15/15] <1.5, 4.1, 4.3, 4.4> In this exercise, we look at how some software techniques can extract ILP in a common vector loop. The following loop is the so-called DAXPY loop (*d*ouble-precision *aX plus Y;* discussed in Appendix G) and the central operation in Gaussian elimination. The following code implements the DAXPY operation, Y = a × X + Y, for a vector length 100.

```
bar:    L.D       F2,0(R1)       ;load X(i)
        MUL.D     F4,F2,F0       ;multiply a*X(i)
        L.D       F6,0(R2)       ;load Y(i)
        ADD.D     F6,F4,F6       ;add a*X(i) + Y(i)
        S.D       0(R2),F6       ;store Y(i)
        DADDUI    R1,R1,#8       ;increment X index
        DADDUI    R2,R2,#8       ;increment Y index
        DSGTUI    R3,R1,#800     ;test if done
        BEQZ      R3,bar         ;loop if not done
```

For (a)–(e) assume the pipeline latencies from Figure 4.1 and a 1-cycle delayed branch that resolves in the ID stage. Assume that integer operations issue and complete in 1 clock cycle and that their results are fully bypassed.

a. [20] <1.5, 4.1> 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 (see the example on page 305). What is the execution time 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 the possible increase in the number of clock cycles necessary for memory system access effects of higher processor clock speed on memory system performance)?

b. [15] <4.1> 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 vector? What is the major contribution to the reduction in time per element?

c. [20] <4.3> Assume a VLIW processor with instructions that contain five operations, as shown in Figure 4.5. We will compare two degrees of loop

unrolling. First, unroll the loop 4 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?

d. [15] <4.4> Assume a family of VLIW processors designed for different price-performance points in the marketplace. The budget version of the processor has latencies greater than those in Figure 4.1. What will happen if the code for your answer to part (c) is executed on the low-cost processor? How could you eliminate any undesirable behavior?

e. [15] <4.4> Assume a single-issue pipeline. Show the schedule for a software-pipelined version of the DAXPY loop. You may omit the start-up and clean-up code.What is the execution time per element of the result vector?

4.10 [20/20] <4.4> In this exercise we finish the compiler code transformation started in the software-pipelining loop example on page 330.

a. [20] <4.4> Starting with the solution loop body given in the example, write code for the complete software-pipelined loop by adding the start-up and finish-up code. Assume that there will be a large number of iterations executed. You need not show code to initialize the loop induction variable and the scalar increment value. Using the latencies in Figure 4.1 and assuming a 1-cycle branch delay, write an expression for the total time for the software-pipelined loop to increment all elements of the array.

b. [20] <4.4> The original loop in the example may execute only one iteration. Write code for the complete software-pipelined loop that allows one, two, or as many iterations as needed for the array size. Then, for your final answer, schedule and possibly further transform your code so that it can execute with only two stalls, and show where those stalls occur. Assume the latencies in Figure 4.1, and use delayed branches that always execute the instruction in the 1-cycle branch delay slot.

4.11 [20] <4.4> Consider the loop that we software-pipelined on page 330. Suppose the latency of the `ADD.D` was 5 cycles. The software-pipelined loop now has a stall. Show how this loop can be written using both software pipelining and loop unrolling to eliminate any stalls. The loop should be unrolled as few times as possible (once is enough). Show the loop start-up and clean-up code.

✪ 4.12 [15] <4.4> Here is a simple code fragment:

```
for (i=2;i<=100;i+=2)
      a[i] = a[50*i+1];
```

To use the GCD test, this loop must first be "normalized"—written so that the index starts at 1 and increments by 1 on every iteration. Write a normalized version of the loop (change the indices as needed), then use the GCD test to see if there is a dependence.

4.13 [15] <4.4> Here is another loop:

```
for (i=2,i<=n/2;i+=2)
      a[i] = a[i] + a[i + n/2];
```

Normalize the loop. Does the GCD test detect a dependence? Is there a loop-carried, true dependence in this loop? Explain.

4.14 [25] <4.4> Show that if there is a true dependence between the two array elements $A(a \times i + b)$ and $A(c \times i + d)$, then GCD(c,a) divides $(d - b)$.

4.15 [15] <4.5> It is common in scientific codes for array elements to be addressed based on the element values in an index array. Consider the following loop:

```
for (i=1,i<=n,i+=1)
      a[x[i]] = a[x[i]] + b[x[i]];
```

The array subscripts are not affine, so the GCD test cannot be used. However, the loop may still be parallel, so additional compiler tests may be valuable. What condition on the index array x[] will make the loop parallel? Be as general in your answer as you can. *Hint:* Think of the loop as adding the elements of one linked list to the corresponding elements of another linked list.

4.16 [15/15/15] <4.5> Consider the following code fragment from an if-then-else statement of the form

```
if (A==0) A = B; else A = A+4;
```

where A is at `0(R3)` and B is at `0(R2)`:

```
        LD      R1,0(R3)      ;load A
        BNEZ    R1,L1         ;test A
        LD      R1,0(R2)      ;then clause
        J       L2            ;skip else
L1:     DADDI   R1,R1,#4      ;else clause
L2:     SD      0(R3),R1      ;store A
```

In the following assume a standard single-issue MIPS pipeline, branch resolution in the ID stage, delayed branches, and forwarding.

a. [15] <4.5> Assume conditional load instructions `LWZ Rd,x(Rs1),Rs2` and `LWNZ Rd,x(Rs1),Rs2` that do not load unless the value of `Rs2` is zero or not zero, respectively. Compile the code using a conditional load and write it showing any stall cycles that would occur in the pipeline. Compare the clock cycles and register use to that of the original code fragment.

b. [15] <4.5> Boosting supports compiler speculation via a load instruction LW+ that includes a flag of the compiler prediction for the branch on which the load depends. If the prediction flag matches the branch resolution result, then the loaded value is written to the register. Compile the code using a boosted load and write it showing any stall cycles that would occur in the

pipeline. Compare the clock cycles and register use to that of the original code fragment.

c. [15] <4.5> Compile the code using compiler-based speculation for both the then and else clauses and write it showing any stall cycles that would occur in the pipeline. Assume conditional move instructions `CMVZ Rd,Rs1,Rs2` and `CMVNZ Rd,Rs1,Rs2` that move the contents of register `Rs1` to register `Rd` if `Rs2` is equal to or not equal to zero, respectively. Compare the clock cycles and register use to that of the original code fragment.

4.17 [15] <4.5> Perform the same transformation (moving up the branch) as the example on page 342, but using only conditional move. Be careful that your loads, which are no longer control dependent, cannot raise an exception if they should not have been executed!

4.18 [15/15] <4.5> In this exercise we will investigate how predication affects the form and execution of pipelined instructions. Conditional execution of instructions is traditionally implemented with branches. For example, the MIPS instruction

```
DADD R1,R2,R3      ;R1=R2+R3
```

is unconditional. Its execution is made conditional on the value in register R8 by writing

```
            BNEZ R8,place ;if(R8==0)
            DADD R1,R2,R3 ;then{ R1=R2+R3 }
place:
```

If predicated, however, the instruction form could be

```
            (R8) ADD R1,R2,R3 ;if(R8){ R1=R2+R3 }
place:
```

with control of execution of the ADD an integral part of the instruction itself, thus eliminating a control dependence between what was before two instructions. If R8 has been set to the value true, then the ADD computes the sum and places it in the result register; otherwise the ADD behaves like a NOP, computing no result and leaving R1 unchanged.

Assume a set of 1-bit predicate registers that are set by a compare instruction of the form

```
(qp) CMP.NE pT,pF=R8,R0
```

This compare (written with mnemonic CMP) is itself predicated on the truth value of the qualifying predicate, qp. This example uses a not equal (`.NE`) comparison relation to match the code fragment above; other relations may be available. If qp is true, the `CMP.NE` sets the 1-bit predicate registers pT and pF such that `pT=(R8!=R0)` and `pF!=(R8!=R0)`. That is, pT is assigned the truth value of the statement "R8 is not equal to R0," and pF is assigned the complement of the truth value of that same statement. If qp is false, the `CMP.NE` behaves as would a NOP instruction.

a. [15] <4.5> Using predicated instructions, write the following code fragment as a single basic block (assuming the SUB instruction is the only entry point for the code). You may assume that any compare instruction you use is not itself predicated.

```
        DSUB R1,R13,R14
        BNEZ R1,L1
        DADDI R2,R2,#1
        SD 0(R7),R2
        J L2
L1:     MUL.D F0,F0,F2
        ADD.D F0,F4
        S.D 0(R8),F0
L2:
```

b. [15] <4.5> What are all the dependences in the code given in part (a), and what are the dependences in the code for your answer to part (a)? How do they differ, and what is the advantage for performance of the predicated code?

4.19 [15/12/12] <4.5, 4.7> See the description of predicated instructions in the preceding exercise, then answer the following questions.

a. [15] <4.5, 4.7> Write the following code without branches. Use predicated MIPS instructions.

```
if (A>B) then { X=1;}
else {
            if (C<D) then { X=2;}
            else { X=3;} }
```

b. [12] <4.7> Where would an IA-64 compiler place stop(s) in the answer to part (a)?

c. [12] <4.7> What are the possible sequences of instruction bundle templates (see Figure 4.12) for the answer to part (a)? Assume that the first instruction begins a bundle.

4.20 [10] <4.5> Predicated instructions cannot eliminate a branch instruction when that branch is part of what kind of program structure?

4.21 [15]<4.7> A compiler for IA-64 has generated the following sequence of three instructions:

```
            L.D F0,0(R1)  ;F0=Mem[0+R1]
(p1)        DADD R1,R2,R3 ;if (p1) then R1=R2+R3
(p2)        DSUB R5,R1,R4 ;if (p2) then R5=R1-R4
```

where p1 and p2 are two predicate registers that are set earlier in the program. Assume that the three instructions are to form a bundle. What are the possible templates that the compiler could use for the bundle (see Figure 4.12), and under what circumstances would each template be chosen?