CHAPTER 7

Instruction-Level Parallelism

7.1 Objectives

This chapter covers a variety of techniques for exploiting *instruction* *lelism* (ILP) by executing independent instructions at the same completing this chapter, you should

1. Understand the concepts behind instruction-level parallelism used
2. Understand and be able to discuss the differences between s VLIW (very long instruction word) processors
3. Be able to predict the execution time of short instruction processors that exploit instruction-level parallelism
4. Be able to predict the impact that out-of-order execution w execution time of programs
5. Be familiar with loop unrolling and software pipelining techniques that improve the performance of programs on

7.2 Introduction

In the last chapter, we discussed pipelining, an important techniqu computer performance by overlapping the execution of multiple ins allows instructions to be executed at a higher rate than would be po instruction had to wait for the previous instruction to complete before i execution. In this chapter, we explore techniques to exploit instr parallelism by executing multiple instructions simultaneously, further performance. Modern processors typically employ both pipelining and techniques that exploit instruction-level parallelism, and we will assume that all of the ILP processors discussed in this chapter are pipelined unless otherwise specified.

Pipelining improves performance by increasing the rate at which instructions can be executed. However, as we saw in the last chapter, there are limits to how much pipelining can improve performance. As more and more pipeline stages are added to a processor, the delay of the pipeline register required in each stage becomes a significant component of the cycle time, reducing the benefit of increasing the pipeline depth. More significantly, increasing the pipeline depth increases branch delay and instruction latency, increasing the number of stall cycles that occur between dependent instructions.

Since the combination of technological constraints and diminishing returns from additional pipelining limits the maximum clock rate of a processor in a given fabrication process, architects have turned to parallelism to improve performance by performing multiple tasks at the same time. Parallel computer systems tend to take one of two forms: multiprocessors and instruction-level parallel processors, which vary in the size of the tasks that are executed in parallel. In multiprocessor systems, which are covered in Chapter 12, relatively large tasks, such as procedures or loop iterations, are executed in parallel. In contrast, instruction-level parallel processors execute individual instructions in parallel.

Processors that exploit instruction-level parallelism have been much more successful than multiprocessors in the general-purpose workstation/PC market because they can provide performance improvements on conventional programs, while this has not been possible on multiprocessors. In particular, superscalar processors can achieve speedups when running programs that were compiled for execution on sequential (non-ILP) processors without requiring recompilation. The other architecture that will be covered in this chapter, VLIW processors, requires that programs be recompiled for the new architecture but achieves very good performance on programs written in sequential languages such as C or FORTRAN when these programs are recompiled for a VLIW processor.

A high level block diagram of an instruction-level parallel processor is shown in Fig. 7-1. The processor contains multiple execution units to execute instructions, each of which reads its operands from and writes its results to a single, centralized register file. When an operation writes its result back to the register file, that result becomes visible to all of the execution units on the next cycle, allowing operations to execute on different units from the operations that generate their inputs. Instruction-level parallel processors often have complex bypassing hardware that forwards the results of each instruction to all of the execution units to reduce the delay between dependent instructions.

The instructions that make up a program are handled by the instruction issue logic, which issues instructions to the units in parallel. This allows control flow changes, such as branches, to occur simultaneously across all of the units, making it much easier to write and compile programs for instruction-level parallel processors.

In Fig. 7-1, all of the execution units have been drawn as identical modules. In most actual processors, some or all of the execution units are only able to execute a subset of the processor's instructions. The most common division is between integer

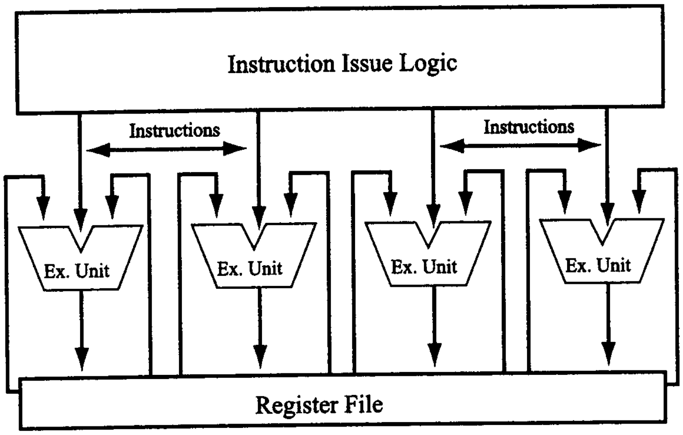

Fig. 7-1. Instruction-level parallel processor.

operations and floating-point operations, because these operations require very different hardware. Implementing these two sets of hardware as separate execution units increases the number of instructions that can be executed simultaneously without significantly increasing the amount of hardware required. On other processors, some of the integer execution units may be constructed to execute only some of the processor's integer operations, generally the most commonly executed operations. This reduces the size of these execution units, although it means that some combinations of independent integer instructions cannot be executed in parallel.

7.3 What is Instruction-Level Parallelism?

Instruction-level parallel processors exploit the fact that many of the instructions in a sequential program do not depend on the instructions that immediately precede them in the program. For example, consider the program fragment in the left side of Fig. 7-2. Instructions 1, 3, and 5 are dependent on each other because instruction 1 generates a data value that is used by instruction 3, which generates a result that is used by instruction 5. Instructions 2 and 4 do not use the results of any other instructions in the fragment and do not generate any results that are used by instructions in the fragment. These dependencies require that instructions 1, 3, and 5 be executed in order to generate the correct result, but instructions 2 and 4 can be

```
1: LD r1, (r2)
2: ADD r5, r6, r7        Cycle 1:  LD r1, (r2)      ADD r5, r6, r7
3: SUB r4, r1, r4   →    Cycle 2:  SUB r4, r1, r4   MUL r8, r9, r10
4: MUL r8, r9, r10       Cycle 3:  ST (r11), r4
5: ST (r11), r4
```

Fig. 7-2. Instruction-level parallelism example.

executed before, after, or in parallel with any of the other instructions without changing the results of the program fragment.

On a processor that executes one instruction at a time, the execution time of this program would be at least five cycles, even on an unpipelined processor with an instruction latency of one cycle. In contrast, an unpipelined processor that is capable of executing two instructions simultaneously could execute the program fragment in three cycles if each instruction had a latency of one cycle, as shown in the right half of the figure. Because instructions 1, 3, and 5 are dependent, it is not possible to reduce the execution time of the fragment any further by increasing the number of instructions that the processor can execute simultaneously.

This example illustrates both the strengths and the weaknesses of instruction-level parallelism. ILP processors can achieve significant speedups on a wide variety of programs by executing instructions in parallel, but their maximum performance improvement is limited by instruction dependencies. In general, as more execution units are added to a processor, the incremental performance improvement that results from adding each execution unit decreases. Going from one execution unit to two gives substantial reductions in execution time. However, as the number of execution units is increased to four, eight, or more, the additional execution units spend most of their time idle, particularly if the program has not been compiled to take advantage of the additional execution units.

7.4 Limitations of Instruction-Level Parallelism

The performance of ILP processors is limited by the amount of instruction-level parallelism that the compiler and the hardware can locate in the program. Instruction-level parallelism is limited by several factors: data dependencies, name dependencies (WAR and WAW hazards), and branches. In addition, a given processor's ability to exploit instruction-level parallelism may be limited by the number and type of execution units present in the processor and by restrictions on which instructions in the program can be examined to locate operations that can be performed in parallel.

RAW dependencies limit performance by requiring that instructions be executed in sequence to generate the correct results, and they represent a fundamental limitation on the amount of instruction-level parallelism available in programs. Instructions with WAW dependencies must also issue sequentially to ensure that the correct instruction writes its destination register last. Instructions with WAR dependencies can issue in the same cycle, but not out of order, because instructions read their inputs from the register file before they issue. Thus, an instruction that reads a register can issue in the same cycle as an instruction that writes the register and appears later in the program, because the reading instruction will read its input registers before the writing instruction generates the new value of its destination register. Later in this chapter, we will discuss *register renaming*, a hardware technique that allows instructions with WAR and WAW dependencies to be executed out of order without changing the results of the program.

Branches limit instruction-level parallelism because the processor does not know which instructions will be executed after a branch until the branch has completed. This requires the processor to wait for the branch to complete before any instructions after the branch can be executed. As mentioned in the last chapter, many processors incorporate branch prediction hardware to reduce the impact of branches on execution time by predicting the destination address of a branch before the branch is executed.

EXAMPLE

Consider the following program fragment :

```
ADD r1, r2, r3
LD r4, (r5)
SUB r7, r1, r9
MUL r5, r4, r4
SUB r1, r12, r10
ST (r13), r14
OR r15, r14, r12
```

How long would this program take to issue on a processor that allows two instructions to be executed simultaneously? How about on a processor that allows four instructions to be executed simultaneously? Assume that the processor can execute instructions in any order that does not violate data dependencies, that all instructions have latencies of one cycle, and that all of the processor's execution units can execute any of the instructions in the fragment.

Solution

On a processor that allows two instructions to be executed simultaneously, this program will take four cycles to issue. One sample sequence is shown in the following, but there are several sequences that take the same number of cycles.

```
Cycle 1: ADD r1, r2, r3     LD r4, (r5)
Cycle 2: SUB r7, r1, r9     MUL r5, r4, r4
Cycle 3: SUB r1, r12, r10   ST (r13), r14
Cycle 4: OR r15, r14, r12
```

If the processor can execute four instructions simultaneously, the program can issue in two cycles, as follows:

```
Cycle 1: ADD r1, r2, r3  LD r4, (r5)  ST (r13), r14  OR r15, r14, r12
Cycle 2: SUB r7, r1, r9  MUL r5, r4, r4  SUB r1, r12, r10
```

Note that, regardless of the number of instructions that the processor can execute simultaneously, it is not possible to issue this program fragment in only one cycle, because of the RAW dependencies between the ADD r1, r2, r3 and SUB r7, r1, r9 instructions and between the LD r4, (r5) and MUL r5, r4, r4 instructions. Also, note that the SUB r7, r1, r9 and SUB r1, r12, r10 instructions, which have a WAR dependence, are issued in the same cycle.

7.5 Superscalar Processors

Superscalar processors rely on hardware to extract instruction-level parallelism from sequential programs. During each cycle, the instruction issue logic of a superscalar processor examines the instructions in the sequential program to determine which instructions may be issued on that cycle. If enough instruction-level parallelism exists within a program, a superscalar processor can execute one instruction per execution unit per cycle, even if the program was originally compiled for execution on a processor that could only execute one instruction per cycle.

This capability is one of the greatest advantages of superscalar processors and is the reason why virtually all workstation and PC CPUs are superscalar processors. Superscalar processors can run programs that were originally compiled for purely sequential processors, and they can achieve better performance on these programs than processors that are incapable of exploiting instruction-level parallelism. Thus, users who buy systems containing superscalar CPUs can install their old programs on those systems and see better performance on those programs than was possible on their old systems.

The ability of superscalar processors to exploit instruction-level parallelism on sequential programs does not mean that compilers are irrelevant for systems built around superscalar processors. In fact, good compilers are even more critical to the performance of superscalar systems than they are on purely sequential processors. Superscalar processors can only examine a small *window* of the instructions in a program at one time to determine which instructions can be executed in parallel. If a compiler is able to schedule a program's instructions so that large numbers of independent instructions occur within this window, a superscalar processor will be able to achieve good performance on the program. If most of the instructions within the window at any time are dependent on each other, a superscalar processor will not be able to run the program much faster than a sequential processor would. In Section 7.9, we will discuss techniques that a compiler can use to improve the performance of programs on superscalar processors.

7.6 In-Order versus Out-of-Order Execution

One of the significant complexity/performance trade-offs in the design of a superscalar processor is whether the processor is required to execute instructions in the order that they appear in the program (in-order execution), or whether the processor can execute instructions in any order that does not change the result of the program (out-of-order execution). Out-of-order execution can provide much better performance than in-order execution but requires much more complex hardware to implement.

7.6.1 PREDICTING EXECUTION TIMES ON IN-ORDER PROCESSORS

In the previous chapter, we divided the execution time of programs on pipelined processors into the time to issue all of the instructions in the program and the pipeline latency of the processor, giving

$$\text{Execution Time (in Cycles)} = \text{Pipeline Latency} + \text{Issue Time} - 1$$

On pipelined ILP processors, we can use the same expression for the execution time of a program, but calculating the issue time becomes somewhat more complex because the processor can issue more than one instruction in a cycle. Since the pipeline latency of a processor does not vary from program to program, most of the exercises in this chapter will focus on determining the issue time of programs on ILP processors.

On in-order superscalar processors, the issue time of a program can be determined by stepping sequentially through the code to determine when each instruction can issue, similar to the technique used for pipelined processors that execute only one instruction per cycle. The key difference between an in-order superscalar processor and a non-superscalar pipelined processor is that a superscalar processor can issue an instruction in the same cycle as the previous instruction in the program if the data dependencies allow, as long as the number of instructions issued in the cycle does not exceed the number of instructions that the processor can execute simultaneously. On processors where some or all of the execution units can only execute some instructions, the set of instructions issued on a given cycle must match the limitations of the execution units.

EXAMPLE

How long would the following sequence of instructions take to execute on an in-order processor with two execution units, each of which can execute any instruction? Load operations have a latency of two cycles, and all other operations have a latency of one cycle. Assume that the pipeline depth is 5 stages.

```
LD r1, (r2)
ADD r3, r1, r4
SUB r5, r6, r7
MUL r8, r9, r10
```

Solution

The pipeline latency of this processor is five cycles. Assuming that the LD issues on cycle *n*, the ADD cannot issue until cycle $n + 2$ because it is dependent on the LD. The SUB is independent of the ADD and the LD, so it also issues on cycle $n+2$. (It cannot issue in cycle n or $n+1$ because the processor must issue instructions in order.) The MUL is also independent of all previous instructions, but must wait until cycle $n+3$ to issue, because the processor can only issue two instructions per cycle. Therefore, it takes four cycles to issue all of the instructions in the program, and the execution time is $5+4-1=8$ cycles.

7.6.2 PREDICTING EXECUTION TIMES ON OUT-OF-ORDER PROCESSORS

Determining the issue time of a sequence of instructions on an out-of-order processor is significantly more difficult than determining the issue time of the same sequence on an in-order processor, because there are many possible orders in which the instructions could execute. In general, the best approach is to start by examining the sequence of instructions to locate the dependencies between instructions. Once the dependencies between instructions are understood, they can then be assigned to issue cycles to minimize the delay between the execution of the first and last instructions in the sequence.

The effort required to find the best-possible ordering of a set of instructions grows exponentially with the number of instructions in the set, since all possible orderings must potentially be considered. Thus, we will assume that the instruction logic in a superscalar processor places some restrictions on the order in which instructions issue in order to simplify the instruction issue logic. The assumption we will make is that the processor issues an instruction in the first cycle in which the dependencies within the program allow it to issue[1]. If more instructions can issue in a cycle than the processor has execution units, the processor will take a greedy approach and issue the instructions that occur earliest in the program, even if issuing the instructions in a different order would reduce the time required to issue the sequence. When the compiler is able to control when instructions issue, such as in the VLIW processors that are discussed in Section 7.8, we will assume that the compiler considers all possible instruction orderings to find the one with the shortest execution time, since the compiler is able to devote more effort to instruction scheduling than the issue logic.

With this greedy instruction issue assumption, finding the issue time of a sequence of instructions on an out-of-order processor becomes much easier. Starting at the first instruction in the sequence, proceed through the instructions, assigning each instruction to the earliest cycle on which all of its input operands are available, the number of instructions already assigned to issue in the cycle is less than the number of instructions that the processor can issue simultaneously, and the set of instructions to be issued does not violate the limitations of the processor's execution units, even if this means that an instruction issues before an instruction that appears

---

[1] This is a simplifying assumption that may or may not be true on any particular out-of-order processor. The order in which instructions issue on an out-of-order processor is strongly dependent on the details of the processor's instruction issue logic, and different processors may have different policies.

later in the original program. Repeating this process for all of the instructions in the sequence will determine how long the sequence takes to issue.

EXAMPLE

How long would the following sequence of instructions take to issue on an out-of-order processor with two execution units, each of which can execute any instruction? Load operations have a latency of 2 cycles, and all other operations have a latency of 1 cycle. (This is the same sequence as the example that was used to illustrate in-order instruction issue.)

```
LD r1, (r2)
ADD r3, r1, r4
SUB r5, r6, r7
MUL r8, r9, r10
```

Solution

The only dependency in this sequence is between the LD and the ADD instructions (a RAW dependency). Because of this dependency the ADD instruction must issue at least two cycles after the LD. The SUB and the MUL could both issue in the same cycle as the LD. Using our greedy assumption, the SUB and the LD issue in cycle n, the MUL issues in cycle $n + 1$, and the ADD issues in cycle $n + 2$, giving a three-cycle issue time for this program.

7.6.3 IMPLEMENTATION ISSUES FOR OUT-OF-ORDER PROCESSORS

On in-order processors, the *instruction window* (the number of instructions the processor examines to select instructions to issue in each cycle) can be relatively small, since the processor is not allowed to issue an instruction until all of the instructions that appear before it in the program have been issued. On a processor with n execution units, only the next n instructions in the program can possibly be issued in a given cycle, so an instruction window length of n instructions is generally sufficient.

Out-of-order processors require much larger instruction windows than in-order processors, to give them as much opportunity as possible to find instructions that can issue in a given cycle. However, the size of the instruction logic grows quadratically with the number of instructions in the instruction window, since each instruction in the window must be compared to all other instructions to determine the dependencies between them. This makes large instruction windows expensive to implement in terms of the amount of hardware required.

The procedure presented earlier for determining the execution time of an instruction sequence on an out-of-order processor assumed that the processor's instruction window was large enough to allow the processor to examine all of the instructions in the sequence simultaneously. If this is not the case, predicting execution time becomes much more difficult, as it becomes necessary to keep track of which instructions are contained within the instruction window on any given cycle and only select instructions to issue from within that set.

Handling interrupts and program exceptions is another difficult implementation issue on out-of-order processors. If instructions can execute out of order, it becomes very difficult to determine exactly which instructions have executed when an instruction takes an exception or when an interrupt occurs. This makes it hard for the programmer to determine the cause of an exception and makes it hard for the system to return to execution of the original program when an interrupt handler completes.

To combat this, virtually all out-of-order processors use a technique called *in-order retirement*. When an instruction generates its result, the result is only written into the register file if all earlier instructions in the program have completed. Otherwise, the result is saved until all earlier instructions have completed, and only then written into the register file. Since results are written into the register file in order, the hardware can simply discard all results that are waiting to be written into the register file when an exception or interrupt occurs. This presents the illusion that instructions are being executed in order, allowing programmers to debug errors relatively easily and making it possible to resume execution of the program at the next instruction when an interrupt handler completes. Processors that use this technique generally use bypassing logic or other techniques to forward the result of an instruction to dependent instructions before the result is written into the register file. This allows dependent instructions to issue as soon as an instruction generates its result, rather than having to wait until the instruction's result is written back into the register file.

7.7 Register Renaming

WAR and WAW dependencies are sometimes referred to as "name dependencies," because they are a result of the fact that programs are forced to reuse registers because of the limited size of the register file. These dependencies can limit instruction-level parallelism on superscalar processors, because it is necessary to ensure that all instructions that read a register complete the register read stage of the pipeline before any instruction overwrites that register.

Register renaming is a technique that reduces the impact of WAR and WAW dependencies on parallelism by dynamically assigning each value produced by a program to a new register, thus breaking WAR and WAW dependencies. Figure 7-3 illustrates register renaming. Each instruction set has an *architectural register file*, which is the set of registers that the instruction set uses. All instructions specify their inputs and outputs out of the architectural register file. On the processor, a larger register file, known as the *hardware register file*, is implemented instead of the architectural register file. Renaming logic tracks mappings between registers in the architectural register file and the hardware register file.

Whenever an instruction reads a register in the architectural register file, the register ID is sent through the renaming logic to determine which register in the hardware register file should be accessed. When an instruction writes a register in

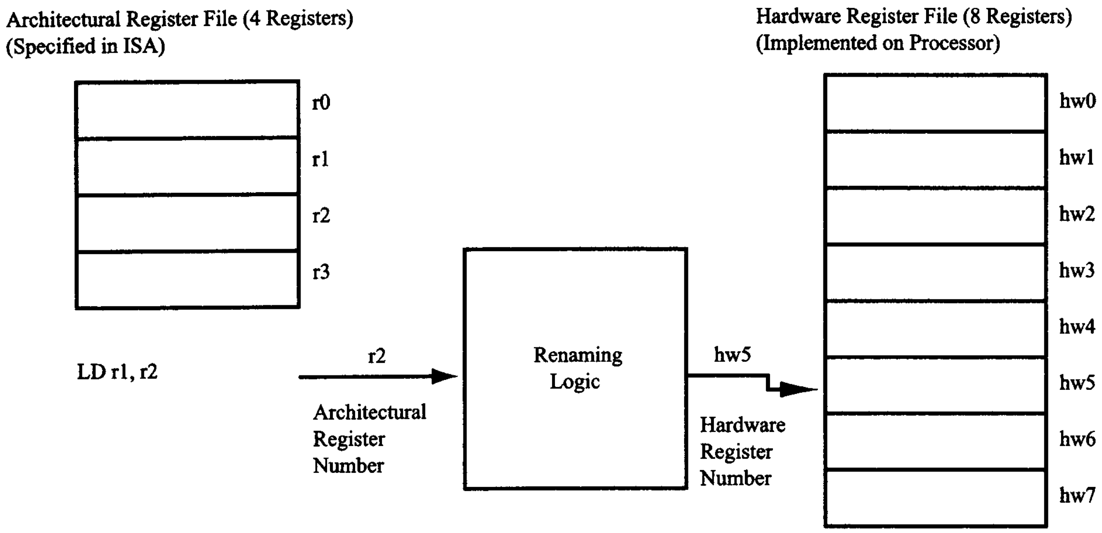

Fig. 7-3. Register renaming.

the architectural register file, the renaming logic creates a new mapping between the architectural register that was written and a register in the hardware register file. Subsequent instructions that read the architectural register access the new hardware register and see the result of the instruction.

Figure 7-4 illustrates how register renaming can improve performance. In the original (before renaming) program, a WAR dependence exists between the LD r7, (r3) and SUB r3, r12, r11 instructions. The combination of RAW and WAR dependencies in the program forces the program to take at least three cycles to issue, because the LD must issue after the ADD, the SUB cannot issue before the LD, and the ST cannot issue until after the SUB.

Before Renaming		After Renaming
ADD r3, r4, r5		ADD hw3, hw4, hw5
LD r7, (r3)	→	LD hw7, (hw3)
SUB r3, r12, r11		SUB hw20, hw12, hw11
ST (r15), r3		ST (hw15), hw20

Fig. 7-4. Register renaming example

With register renaming, the first write to r3 maps to hardware register hw3, while the second maps to hw20 (these are just arbitrary examples). This remapping converts the original four-instruction dependency chain into 2 two-instruction chains, which can then be executed in parallel if the processor allows out-of-order execution. In general, register renaming is of more benefit on out-of-order processors than on in-order processors, because out-of-order processors can reorder instructions once register renaming has broken the name dependencies.

EXAMPLE

On an out-of-order superscalar processor with 8 execution units, what is the execution time of the following sequence with and without register renaming if any execution unit can execute any instruction and the latency of all instructions is one cycle? Assume that the hardware register file contains enough registers to remap each destination register to a different hardware register and that the pipeline depth is 5 stages.

```
LD r7, (r8)
MUL r1, r7, r2
SUB r7, r4, r5
ADD r9, r7, r8
LD r8, (r12)
DIV r10, r8, r10
```

Solution

In this example, WAR dependencies are a significant limitation on parallelism, forcing the DIV to issue 3 cycles after the first LD, for a total execution time of 8 cycles (the MUL and the SUB can execute in parallel, as can the ADD and the second LD). After register renaming, the program becomes

```
LD hw7, (hw8)
MUL hw1, hw7, hw2
SUB hw17, hw4, hw5
ADD hw9, hw17, hw8
LD hw18, (hw 12)
DIV hw10, hw18, hw10
```

(Again, all of the renaming register choices are arbitrary.)

With register renaming, the program has been broken into three sets of two dependent instructions (LD and MUL, SUB and ADD, LD and DIV). The SUB and the second LD instruction can now issue in the same cycle as the first LD. The MUL, ADD, and DIV instructions all issue in the next cycle, for a total execution time of 6 cycles.

Adding register renaming to a processor generally gives less of an improvement than changing the instruction set architecture to make the new registers part of the architectural registers, because the compiler cannot use the new registers to store temporary values. However, register renaming allows new processors to remain compatible with programs compiled for older versions of the processor because it does not require changing the ISA. In addition, increasing the number of architectural registers in a processor increases the number of bits required for each instruction, as a larger number of bits is required to encode the operands and destination register.

7.8 VLIW Processors

The superscalar processors that we have discussed so far in this chapter use hardware to exploit ILP by locating instructions that can execute in parallel from within sequential programs. Their ability to achieve performance improvements on old programs and maintain compatibility between generations of a processor family has made them tremendously successful commercially, but achieving good performance on superscalar processors requires a great deal of hardware. Very long instruction word (VLIW) processors take a different approach to instruction-level parallelism, relying on the compiler to determine which instructions may be executed in parallel and provide that information to the hardware.

In a VLIW processor, each instruction specifies several independent operations that are executed in parallel by the hardware, as shown in Figs. 7-5 and 7-6. Each

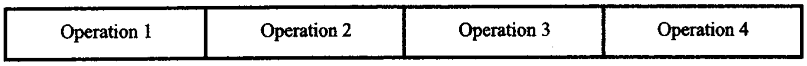

Fig. 7-5. VLIW instruction.

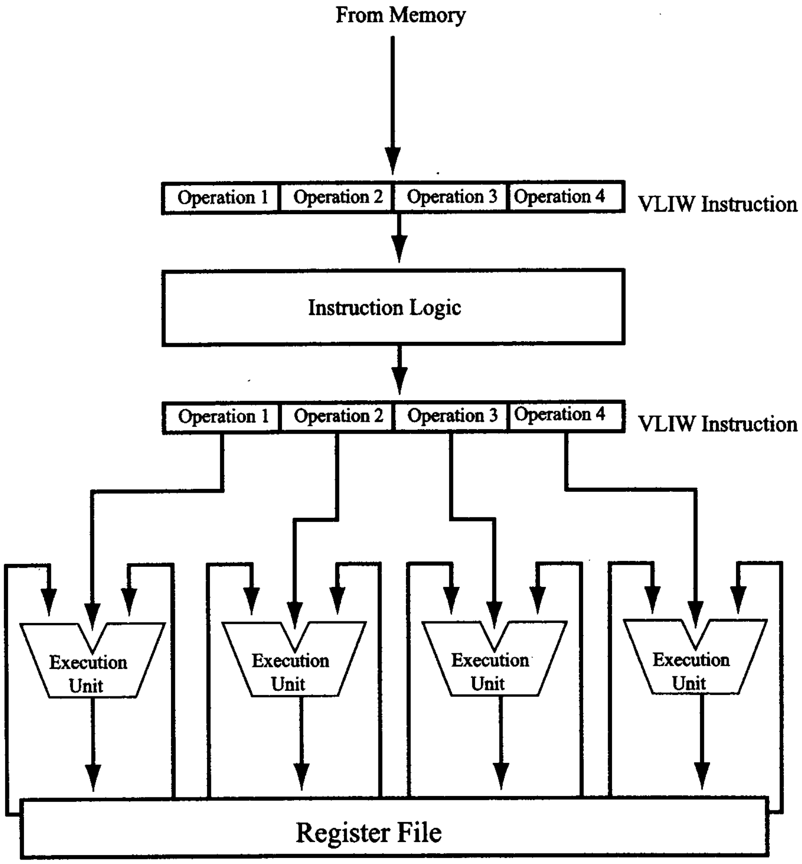

Fig. 7-6. VLIW processor.

operation in a VLIW instruction is equivalent to one instruction in a superscalar or purely sequential processor. The number of operations in a VLIW instruction is equal to the number of execution units in the processor, and each operation specifies the instruction that will be executed on the corresponding execution unit in the cycle that the VLIW instruction is issued. There is no need for the hardware to examine the instruction stream to determine which instructions may be executed in parallel, as the compiler is responsible for ensuring that all of the operations in an instruction can be executed simultaneously. Because of this, the instruction issue logic on a VLIW processor is much simpler than the instruction issue logic on a superscalar processor with the same number of execution units.

Most VLIW processors do not have scoreboards on their register files. Instead, the compiler is responsible for ensuring that an operation is not issued before its operands are ready. Each cycle, the instruction logic fetches a VLIW instruction from the memory and issues it to the execution units for execution. Thus, the compiler can predict exactly how many cycles will elapse between the execution of two operations by counting the number of VLIW instructions between them. In addition, the compiler can schedule instructions with a WAR dependency out of order so long as the instruction that reads the register issues before the instruction that writes the register completes, because the old value in the register is not overwritten until the writing instruction completes. For example, on a VLIW processor with a two-cycle load latency, the sequence ADD r1, r2, r3, LD r2, (r4) could be scheduled so that the ADD operation appeared in the instruction after the load, since the load will not overwrite r2 until two cycles have elapsed.

7.8.1 PROS AND CONS OF VLIW

The main advantages of VLIW architectures are that their simpler instruction logic often allows them to be implemented with shorter clock cycles than superscalar processors and that the compiler has complete control over when operations are executed. The compiler generally has a larger-scale view of the program than the instruction logic in a superscalar processor and is therefore generally better than the issue logic at finding instructions to execute in parallel. Their simpler instruction issue logic also often allows VLIW processors to fit more execution units onto a given amount of chip space than superscalar processors.

The most significant disadvantage of VLIW processors is that VLIW programs only work correctly when executed on a processor with the same number of execution units and the same instruction latencies as the processor they were compiled for, which makes it virtually impossible to maintain compatibility between generations of a processor family. If the number of execution units in a processor increases between generations, the new processor will try to combine operations from multiple instructions in each cycle, potentially causing dependent instructions to execute in the same cycle. Changing instruction latencies between generations of a processor family can cause operations to execute before their inputs are ready or after their inputs have been overwritten, resulting in incorrect behavior. In addition, if the compiler cannot find enough parallel operations to fill all of the slots in an instruction, it must place explicit NOP (no-operation) operations into the corresponding operation slots. This causes VLIW programs to take more memory than equivalent programs for superscalar processors.

Because of their advantages and disadvantages, VLIW processors are often used in digital signal-processing (DSP) applications, where high performance and low cost are critical. They have been less successful in general-purpose computers such as workstations and PCs, because customers demand software compatibility between generations of a processor.

EXAMPLE

Show how a compiler would schedule the following sequence of operations for execution on a VLIW processor with 3 execution units. Assume that all operations have a latency of two cycles, and that any execution unit can execute any operation.

```
ADD r1, r2, r3
SUB r16, r14, r7
LD r2, (r4)
LD r14, (r15)
MUL r5, r1, r9
ADD r9, r10, r11
SUB r12, r2, r14
```

Solution

Figure 7-7 shows how these operations would be scheduled. Note that the LD r14, (r15) is scheduled in the instruction before the SUB r16, r14, r7 operation despite the fact that the SUB instruction appears earlier in the original program and reads the destination register of the LD. Because VLIW operations do not overwrite their register values until they complete, the previous value of r14 remains available until 2 cycles after the instruction containing the LD issues, allowing the SUB to see the old value of r14 and generate the correct result. Scheduling these operations out-of-order in this way allows the program to be scheduled into fewer instructions than would be possible otherwise. Similarly the ADD r9, r10, r11 operation is scheduled ahead of the MUL r5, r1, r9 operation, although these operations could have been placed in the same instruction without increasing the number of instructions required by the program.

Instruction 1	*ADD r1, r2, r3*	*LD r2, (r4)*	*LD r14, (r15)*
Instruction 2	*SUB r16, r14, r7*	*ADD r9, r10, r11*	*NOP*
Instruction 3	*MUL r5, r1, r9*	*SUB r12, r2, r14*	*NOP*

Fig. 7-7. VLIW scheduling example.

7.9 Compilation Techniques for Instruction-Level Parallelism

Compilers use a wide variety of techniques to improve the performance of compiled programs, including constant propagation, dead code elimination, and register allocation. A general discussion of compiler optimizations is beyond the scope of this book, but this section will cover loop unrolling, an optimization that significantly increases instruction-level parallelism, in detail and will briefly describe software pipelining, another compilation technique used to improve performance.

7.9.1 LOOP UNROLLING

Individual loop iterations tend to have relatively low instruction-level parallelism because they often contain chains of dependent instructions, and because of the limited number of instructions between branches. Loop unrolling addresses this limitation by transforming a loop with N iterations into a loop with N/M iterations, where each iteration in the new loop does the work of M iterations of the old loop. This increases the number of instructions between branches, giving the compiler and the hardware more opportunity to find instruction-level parallelism. In addition, if the iterations of the original loop are independent or contain only a few dependent computations, loop unrolling can create multiple chains of dependent instructions where only one chain existed before unrolling, also increasing the ability of the system to exploit instruction-level parallelism.

Figure 7-8 shows a C-language example of loop unrolling. The original loop iterates through the source arrays one element at a time, computing the sum of corresponding elements in the source arrays and storing the result in the destination array. The unrolled loop steps through the arrays at two elements per iteration, performing the work of two iterations of the original loop in each iteration of the unrolled loop. In this example, the loop has been unrolled two times.[2] The original loop could have been unrolled four times by adding 4 to the loop index i in each iteration and performing the work of four original iterations in each iteration of the unrolled loop.

Original Loop

```
for (i = 0; i < 100; i++){
      a[i] = b[i] + c[i];
}
```

→

Unrolled Loop

```
for (i = 0; i < 100; i += 2) {
      a[i] = b[i] + c[i];
      a[i + 1] = b[i + 1] + c[i + 1];
}
```

Fig. 7-8. C-language loop unrolling example.

This example also illustrates another advantage of loop unrolling: reduction in loop overhead. By unrolling the original loop two times, we reduce the number of iterations of the loop from 100 to 50. This halves the number of conditional branch

---

[2] A loop is said to have been unrolled *n* times if each iteration of the unrolled loop performs the work of *n* iterations of the original loop.

instructions that must be executed at the end of loop iterations, reducing the total number of instructions that the system needs to execute during the loop in addition to exposing more instruction-level parallelism. Thus, loop unrolling can be of some benefit even on purely sequential processors, although its benefits are most significant on ILP processors.

Figure 7-9 shows how the loop from Fig. 7-8 might be implemented in assembly language, and how a compiler might unroll the loop and schedule it for fast execution on a superscalar processor with 32-bit data types. Even in the original loop, the compiler has scheduled the code to expose as much ILP as possible by placing both of the initial loads ahead of either of the ADDs that increment pointers, and by performing the pointer increments ahead of the ADD that implements $a[i] = b[i] + c[i]$. Arranging instructions such that independent instructions are close together in the program makes it easier for the hardware in a superscalar processor to locate instruction-level parallelism, while placing the pointer increments between the loads and the computation of $a[i]$ increases the number of operations between the loads and the use of their results, making it more likely that the loads will complete before their results are needed.

The unrolled loop begins with three adds to generate pointers to $a[i + 1]$, $b[i + 1]$, and $c[i + 1]$. Keeping these pointers in separate registers from the pointers to $a[i]$, $b[i]$, and $c[i]$ allows the loads and stores to the ith and $i + 1$th elements of each array to be done in parallel, rather than having to increment each pointer between memory references. This initial block of setup instructions for the unrolled loop is called the *preamble* to the loop. In the body of the loop, the compiler has placed all of the loads in one block, followed by all of the pointer increments, then the computation of $a[i]$ and $a[i + 1]$, and finally the stores and loopback branch. This maximizes both parallelism and the time for the loads to complete.

In the example we have studied so far, the number of iterations of the original loop was evenly divisible by the degree of loop unrolling, making it easy to unroll the loop. In many cases, however, the number of loop iterations is not divisible by the degree of unrolling, or it is not known at compile time, making it harder to unroll the loop. For example, the compiler might want to unroll the loop of Fig. 7-8 eight times, or the number of iterations might be an input parameter to the procedure containing the loop.

In these cases, the compiler generates two loops. The first loop is an unrolled version of the original loop, and it executes until the number of iterations remaining is less than the degree of loop unrolling. The second loop then executes the remaining iterations one at a time. In loops where the number of iterations is not known at the start of the loop, such as a loop that iterates through a string looking for the end-of-string character, it becomes much harder to unroll the loop and more sophisticated techniques are required.

Figure 7-10 shows how the loop of Fig. 7-8 would be unrolled eight times. The first loop steps through the iterations eight at a time, until there are fewer than eight iterations remaining (detected when $i + 8 >= 100$). The second loop starts at the next iteration and steps through the remaining iterations one at a time. Because i is an integer variable, the computation $i = ((100/8) \times 8)$ does not set i to 100. The integer computation 100/8 generates only the integer portion of the quotient

Original Loop

```
      MOV r31, #0  /* initialize i */
loop: LD r1,( r2);    /* r2 contains pointer to b[i] */
      LD r3, (r4);    /* r4 contains pointer to c[i] */
      ADD r2, #4, r2;         /* Increment to b[i + 1] */
      ADD r4, #4, r4;         /* Increment to c[i + 1] */
      ADD r6, r1, r3;
      ST (r7), r6;    /* r7contains pointer to a[i] */
      ADD r7, #4, r7;         /* Increment to a[i + 1] */
      ADD r31, #1, r31;       /* Increment i */
      BNE loop, r31, #100; /* Jump to next iteration if not done */
```

Unrolled Loop

```
      MOV r31, #0
      ADD r8, #4, r2;       /* r8 contains pointer to b[i +1] */
      ADD r10, #4, r4;      /* r10 contains pointer to c[i + 1] */
      ADD r13, #4, r6;      /* r13 contains pointer to a[i + 1] */
loop: LD r1, (r2);
      LD r9, (r8);
      LD r3, (r4);
      LD r11, (r10);
      ADD r2, #8, r2;       /* Note: increment by 2x32 bits */
      ADD r4, #8, r4;
      ADD r8, #8, r8;
      ADD r10, #8, r10;
      ADD r6, r1, r3;
      ADD r12, r9, r11;
      ST (r7), r6;
      ST (r13), r12;
      ADD r7, #8, r7;
      ADD r13, #8, r13;
      ADD r31, #2, r31;     /* r31 gets incremented by 2 instead of 1 */
      BNE loop, r31, #100;
```

Fig. 7-9. Assembly-language loop unrolling example.

Original Loop

```
for (i = 0; i < 100; i++){
      a[i] = b[i] + c[i];
}
```

Unrolled Loop

```
for (i = 0; i < 100; i += 8) {
      a[i] = b[i] + c[i];
      a[i + 1] = b[i + 1] + c[i + 1];
      a[i + 2] = b[i + 2] + c[i + 2];
      a[i + 3] = b[i + 3] + c[i + 3];
      a[i + 4] = b[i + 4] + c[i + 4];
      a[i + 5] = b[i + 5] + c[i + 5];
      a[i + 6] = b[i + 6] + c[i + 6];
      a[i + 7] = b[i + 7] + c[i + 7];
}
for (i = ((100 / 8)x8); i < 100; i++){
      a[i] = b[i] + c[i];
}
```

Fig. 7-10. Uneven loop unrolling.

(dropping the remainder), and multiplying that result by 8 gives the largest multiple of 8 that is less than 100 (96), which is where the second loop should begin its iterations.

7.9.2 SOFTWARE PIPELINING

Loop unrolling improves performance by increasing the number of independent operations within a loop iteration. Another optimization, *software pipelining*, improves performance by distributing each iteration of the original loop over multiple iterations of the pipelined loop, so that each iteration of the new loop performs some of the work of multiple iterations of the original loop. For example, a loop that fetched $b[i]$ and $c[i]$ from memory, added them together to generate $a[i]$, and wrote $a[i]$ back to memory might be transformed so that each interaction first wrote $a[i-1]$ back to memory, then computed $a[i]$ based on the values of $b[i]$ and $c[i]$ that were fetched in the last iteration, and finally fetched $b[i+1]$ and $c[i+1]$ from memory to prepare for the next iteration. Thus, the work of computing a given element of the $a[\ ]$ array is distributed across three iterations of the new loop.

Interleaving portions of different loop iterations in this way increases instruction-level parallelism in much the same way that loop unrolling does. It also increases the number of instructions between the computation of a value and its use, making it more likely that the value will be ready before it is needed. Many compilers combine software pipelining and loop unrolling to increase instruction-level parallelism further than is possible by applying either optimization individually.

7.10 Summary

Exploiting instruction-level parallelism can greatly improve the performance of a processor by allowing independent instructions to execute at the same time. The performance of an ILP processor on a given program is limited by several factors. The number of execution units in the processor determines the maximum number of instructions that the processor can execute simultaneously. Instruction dependencies limit the amount of instruction-level parallelism available in the program. This limitation is particularly significant for in-order superscalar processors, because one pair of dependent instructions can stall all of the remaining instructions in the program. Finally, instruction-level parallelism can be limited by restrictions on the window of instructions that the system can examine to find instructions that can execute in parallel.

In this chapter, we have covered the two most common architectures for instruction-level parallelism: superscalar processors and very long instruction word processors. VLIW processors rely on the compiler to schedule instructions for parallel execution by placing multiple operations in a single long instruction word. All of the operations in a VLIW instruction execute in the same cycle, allowing the compiler to control which instructions execute in any given cycle. VLIW processors can be relatively simple, allowing them to be implemented at high clock speeds, but they are generally unable to maintain compatibility between generations because any change to the processor implementation requires that programs be recompiled if they are to execute correctly.

Superscalar processors, on the other hand, contain hardware that examines a sequential program to locate instructions that can be executed in parallel. This allows them to maintain compatibility between generations and to achieve speedups on programs that were compiled for sequential processors, but they have a limited window of instructions that the hardware examines to select instructions that can be executed in parallel, which can reduce performance.

Two hardware techniques to improve the performance of superscalar processors were discussed. Out-of-order execution allows the processor to execute instructions in any order that does not change the result of the program. This improves performance by preventing dependent instructions from holding up the execution of later independent instructions. Register renaming breaks WAW and WAR hazards by mapping the architectural register set of the processor onto a larger hardware register set, allowing more instructions to be executed in parallel.

Finally, compiler techniques for ILP processors were briefly discussed. In particular, the loop unrolling optimization, which fuses several iterations of an original loop into one iteration to improve instruction-level parallelism, was discussed in detail. Good compilers are crucial to the performance of both VLIW and superscalar processors.

Instruction-level parallelism will continue to be an important technique for improving performance in the future. As fabrication technologies advance, the amount of time required for each execution unit to communicate with the processor's register file and instruction issue logic may limit performance, requiring more advanced architectures that distribute these resources into several smaller modules that are located close to the individual execution units. This style of processor architecture is an active area of research, and the next decade should see substantial changes in the way instruction-level processors are built.

Solved Problems

Instruction-Level Parallelism

7.1 What is instruction-level parallelism? How do processors exploit it to improve performance?

Solution

Instruction-level parallelism refers to the fact that many of the instructions in a sequential program are independent, meaning that it is not necessary to execute them in the order that they appear in the program to produce the correct result. Processors exploit this by executing these instructions in parallel rather than sequentially, reducing the amount of time that they take to execute programs.

Dependent Operations

7.2 What is the longest chain of dependent operations (include name dependencies) in the following program fragment?

```
LD r7, (r8)
SUB r10, r11, r12
MUL r13, r7, r11
ST (r9), r13
ADD r13, r2, rl
LD r5, (r6)
SUB r3, r4, r5
```

Solution

The longest chain of dependencies is four instructions long.

```
LD r7, (r8)
MUL r13, r7, r11
ST (r9), r13
ADD r13, r2, rl
```

Note that the dependency between the ST and the ADD instructions is a WAR dependency.

Limits of Parallelism

7.3 If the code fragment from Problem 7.2 was executed on a superscalar processor with an infinite number of execution units and one-cycle latencies for all operations, how long would it take to issue? (In other words, what limitations do the dependencies in the program fragment place on the issue time?)

Solution

With an infinite number of execution units, the processor's ability to issue instructions in parallel is limited only by the depth of the chains of dependent instructions in the program.

The longest chain of dependent instructions was identified in the last exercise, and the next-longest chain is only two instructions.

If all of the dependencies in the longest chain were RAW dependencies, the instructions in the chain would have to be issued in sequence, making the issue time four cycles. However, one of the dependencies is a WAR dependency, and instructions with a WAR dependency can be issued in the same cycle. This allows the ST (r9), r13 and ADD r13, r2, r1 instructions to be issued in the same cycle, reducing the issue time to 3 cycles.

In-Order Execution (I)

7.4 How long will the following code fragment take to issue on an in-order superscalar processor with two execution units, where all instructions have latencies of 1 cycle and any execution unit can execute any instruction?

```
LD r1, (r2)
SUB r4, r5, r6
ADD r3, r1, r7
MUL r8, r3, r3
ST (r11), r4
ST (r12), r8
ADD r15, r14, r13
SUB r10, r15, r10
ST (r9), r10
```

Solution

This code fragment takes 6 cycles to issue, as shown below. Note that there are several instructions in the fragment whose data dependencies would allow them to be executed earlier, but that the processor cannot move up any earlier because of the in-order execution requirement.

```
Cycle 1: LD r1, (r2)          SUB r4, r5, r6
Cycle 2: ADD r3, r1, r7
Cycle 3: MUL r8, r3, r3       ST (r11), r4
Cycle 4: ST (r12) r8          ADD r15, r14, r13
Cycle 5: SUB r10, r15, r10
Cycle 6: ST (r9), r10
```

In-Order Execution (II)

7.5 How long will the following code sequence take to issue on an in-order superscalar processor with 4 execution units, where any execution unit can execute any operation, load operations have a 2-cycle latency, and all other operations have a 1-cycle latency?

```
ADD r1, r2, r3
SUB r5, r4, r5
LD r4, (r7)
MUL r4, r4, r4
ST (r7), r4
```

```
LD r9, (r10)
LD r11, (r12)
ADD r11, r11, r12
MUL r11, r11, r11
ST (r12), r11
```

Solution

The code sequence will take 8 cycles to issue, as shown in the following:

```
Cycle 1: ADD r1, r2, r3    SUB r5, r4, r5  LD r4, (r7)
Cycle 2: (nothing)
Cycle 3: MUL r4, r4, r4
Cycle 4: ST (r7), r4       LD r9, (r10)    LD r11, (r12)
Cycle 5: (nothing)
Cycle 6: ADD r11, r11, r12
Cycle 7: MUL r11, r11, r11
Cycle 8: ST (r12), r11
```

In-Order Execution (III)

7.6 How long would the following instructions take to execute on an in-order superscalar processor with two execution units, where each execution unit can execute any operation, load operations have a latency of 3 cycles, and all other operations have a latency of 2 cycles? Assume the processor has a 6-stage pipeline.

```
LD r4, (r5)
LD r7, (r8)
ADD r9, r4, r7
LD r10, (r11)
MUL r12, r13, r14
SUB r2, r3, r1
ST (r2), r15
MUL r21, r4, r7
ST (r22), r23
ST (r24), r21
```

Solution

The pipeline latency is 6 cycles, and it takes 9 cycles to issue all of the instructions, as shown in the following:

```
Cycle 1: LD r4, (r5)       LD r7, (r8)
Cycle 2: (nothing)
Cycle 3: (nothing)
Cycle 4: ADD r9, r4, r7    LD r10, (r11)
Cycle 5: MUL r12, r13, r14 SUB r2, r3, r1
Cycle 6: (nothing)
Cycle 7: ST (r2), r15      MUL r21, r4, r7
Cycle 8: ST (r22), r23
Cycle 9: ST (r24), r21
```

Therefore, the total execution time is $6 + 9 - 1 = 14$ cycles

Restricted Execution Units

7.7 How long would the program from Problem 7.6 take to issue if the processor was limited so that at most one of the instructions issued in a cycle could be a memory (load or store) operation, and at most one of the instructions could be a non-memory operation (i.e., if one of the execution units executed only memory instructions and one of the execution units executed only non-memory instructions?) All other parameters of the processor are the same as in Problem 7.6.

Solution:

The program would take 11 cycles to issue:

```
 Cycle 1: LD r4, (r5)
 Cycle 2: LD r7, (r8)
 Cycle 3: (nothing)
 Cycle 4: (nothing)
 Cycle 5: ADD r9, r4, r7    LD r10, (r11)
 Cycle 6: MUL r12, r13, r14
 Cycle 7: SUB r2, r3, r1
 Cycle 8: (nothing)
 Cycle 9: ST (r2), r15      MUL r21, r4, r7
Cycle 10: ST (r22), r23
Cycle 11: ST (r24), r21
```

Out-of-Order Execution (I)

7.8 How long would the code fragment from Problem 7.4 take to issue on an out-of-order superscalar processor with all other parameters the same as the original exercise? Assume that the instruction window of the processor is large enough to cover the entire code fragment and that the processor takes the greedy approach to issuing instructions discussed in the chapter.

Solution

It would take 5 cycles, as shown in the following:

```
Cycle 1: LD r1, (r2)     SUB r4, r5, r6
Cycle 2: ADD r3, r1, r7  ST (r11), r4
Cycle 3: MUL r8, r3, r3  ADD r15, r14, r13
Cycle 4: ST (r12), r8    SUB r10, r15, r10
Cycle 5: ST (r9), r10
```

Out-of-Order Execution (II)

7.9 How long will the code fragment from Problem 7.5 take to issue on an out-of-order processor whose other parameters are the same as the one in the original exercise? Use the greedy scheduling assumption, and assume that the instruction window of the processor is large enough to cover the entire program fragment.

Solution

It would take 6 cycles to issue. Note that the time to issue these instructions could have been reduced by placing the LD r11, (r12) in cycle 1 instead of the LD r9, (r10), but this would have violated our greedy scheduling assumption.

```
Cycle 1: ADD r1, r2, r3    SUB r5, r4, r5    LD r4, (r7)  LD r9, (r10)
Cycle 2: LD r11, (r12)
Cycle 3: MUL r4, r4, r4
Cycle 4: ST (r7), r4       ADD r11, r11, r12
Cycle 5: MUL r11, r11, r11
Cycle 6: ST (r12), r11
```

Out-of-Order Execution (III)

7.10 How long would the instructions from Problem 7.6 take to issue on an out-of-order superscalar processor with 2 execution units, where all operation latencies are the same as in Problem 7.6? Use the greedy scheduling assumption, and assume that the processor's instruction window is large enough to cover the entire program fragment.

Solution

They would take 6 cycles to issue:

```
Cycle 1: LD r4, (r5)       LD r7, (r8)
Cycle 2: LD r10, (r11)     MUL r12, r13, r14
Cycle 3: SUB r2, r3, r1    ST (r22), r23
Cycle 4: ADD r9, r4, r7    MUL r21, r4, r7
Cycle 5: ST (r2), r15
Cycle 6: ST (r24), r21
```

Out-of-Order Execution with Restricted Execution Units

7.11 Suppose the processor from Problem 7.10 had 1 execution unit that executed memory instructions and 1 execution unit that executed non-memory instructions. If all other parameters of the processor remain the same, how long would the code fragment take to issue?

Solution

It would take 8 cycles to issue:

```
Cycle 1: LD r4, (r5)       MUL, r12, r13, r14
Cycle 2: LD r7, (r8)       SUB r2, r3, r1
Cycle 3: LD r10, (r11)
Cycle 4: ST (r2), r15
Cycle 5: ADD r9, r4, r7    ST (r22), r23
Cycle 6: MUL r21, r4, r7
Cycle 7: (nothing)
Cycle 8: ST (r24), r21
```

Register Renaming (I)

7.12 How many hardware registers are required to allow register renaming to break all of the WAR and WAW dependencies in the following set of instructions?

```
LD r1, (r2)
ADD r3, r4, r1
SUB r4, r5, r6
MUL r7, r4, r8
ASH r8, r9, r10
SUB r11, r8, r12
DIV r12, r13, r14
ST (r15), r12
```

Solution

The code fragment uses 15 architectural registers. In addition, there are three WAR dependencies: ADD r3, r4, r1 → SUB r4, r5, r6, MUL r7, r4, r8 → ASH r8, r9, r10, and SUB r11, r8, r12 → DIV r12, r13, r14. There are no WAW dependencies in the code. Therefore, a total of 18 hardware registers are required if register renaming is to break all of the name dependencies in the program (15 for the 15 architectural registers, plus 3 to rename each of the registers involved in the WAR dependencies).

Register Renaming (II)

7.13 Show how register renaming hardware would transform the code fragment from the previous exercise. Assume that the processor has sufficient hardware registers to perform the required renaming.

Solution

(Hardware register numbers are the same as architectural register numbers except when renaming is required to break dependencies.)

```
LD hw1, (hw2)
ADD hw3, hw4, hw1
SUB hw16, hw5, hw6
MUL hw7, hw16, hw8
ASH hw17, hw9, hw10
SUB hw11, hw17, hw12
DIV hw18, hw13, hw14
ST (hw15), hw18
```

Register Renaming (III)

7.14 How long would the original code sequence from Problem 7.12 and the renamed code sequence from the previous exercise take to issue on an out-of-order superscalar processor with 4 execution units, each of which can execute any operation? Assume all instructions have latencies of 1 cycle, use the greedy scheduling assumption, and assume that the processor's instruction window is large enough to cover the entire code sequence.

Solution

Without register renaming the sequence takes 5 cycles to issue, because instructions with a WAR dependency can issue in the same cycle, but not out of order:

```
Cycle 1: LD r1, (r2)
Cycle 2: ADD r3, r4, r1     SUB r4, r5, r6
Cycle 3: MUL r7, r4, r8     ASH r8, r9, r10
Cycle 4: SUB r11, r8, r12   DIV r12, r13, r14
Cycle 5: ST (r15), r12
```

With register renaming, the sequence can be issued in 2 cycles, because we can issue instructions that originally had WAR dependencies out of order:

```
Cycle 1: LD hw1, (hw2)        SUB hw16, hw5, hw6  ASH hw17, hw9, hw10    DIV hw18, hw13, hw14
Cycle 2: ADD hw3, hw4, hw1    MUL hw7, hw16, hw8  SUB hw11, hw17, hw12   ST(hw15), hw18
```

VLIW Scheduling (I)

7.15 Show how a compiler would schedule the code from Problem 7.5 for execution on a VLIW processor with the same number of execution units and instruction latencies as specified in the original exercise. Unlike the out-of-order execution problems, you should assume that the compiler examines all possible instruction orderings to find the best schedule. (This reflects the fact that the compiler can devote greater effort to finding the best schedule than is usually possible in hardware.) Be sure to include the NOPs (no operation instruction) for unused operations.

Solution

The code can be scheduled in 5 instructions. One example of a correct schedule is shown in the following, although other schedules that use the same number of instructions exist:

```
Instruction 1: SUB r4, r5, r5  LD r4, (r7)          LD r9, (r10)  LD r11, (r12)
Instruction 2: ADD r1, r2, r3  NOP                  NOP           NOP
Instruction 3: MUL r4, r4, r4  ADD r11, r11, r12    NOP           NOP
Instruction 4: ST (r7), r4     MUL r11, r11, r11    NOP           NOP
Instruction 5: ST (r12), r11   NOP                  NOP           NOP
```

VLIW Scheduling (II)

7.16 Show how a compiler would schedule the following program for execution on a VLIW with 4 execution units, each of which can execute any instruction type. Load instructions have a latency of 3 cycles, and all other instructions have a latency of 1 cycle. Keep in mind that, in a VLIW, the old value of an operation's destination register remains available to be read until the operation completes.

```
SUB r4, r7, r8
MUL r10, r11, r12
DIV r14, r13, r15
```

```
ADD r9, r3, r2
LD r7, (r20)
LD r8, (r21)
LD r11, (r22)
LD r12, (r23)
LD r13, (r24)
LD r15, (r25)
LD r3, (r30)
LD r2, (r31)
ST (r26), r4
ST (r27), r10
ST (r28), r14
ST (r29), r9
```

Solution

Taking advantage of the fact that old register contents are not overwritten in a VLIW processor until the writing instruction completes, this sequence can be scheduled into 4 instructions. Here, we are deliberately scheduling the SUB r4, r7, r8 after instructions that load r7 and r8, but before those instructions complete. The subtract will see the old value of r7 and r8, which is what we want, since the subtract appears before the loads in the original program. The MUL, DIV and ADD instructions are similarly scheduled during the latencies of instructions that overwrite their input operands so that they see the old values of those registers.

Instruction 1:	`LD r7, (r20)`	`LD r8, (r21)`	`LD r11, (r22)`	`LD r12, (r23)`
Instruction 2:	`LD r13, (r24)`	`LD r15, (r25)`	`LD r3, (r30)`	`LD r2, (r31)`
Instruction 3:	`SUB r4, r7, r8`	`MUL r10, r11, r12`	`DIV r14, r13, r15`	`ADD r9, r3, r2`
Instruction 4:	`ST (r26), r4`	`ST (r27), r10`	`ST (r28), r14`	`ST (r29), r9`

(Note that the load instructions can be placed in the first two instructions in any order without changing the number of instructions that are required.)

Loop Unrolling (I)

7.17 Why does unrolling a loop often improve performance?

Solution

Loop unrolling improves performance because iterations of loops are often independent, or at least contain some operations that do not depend on the previous iteration of the loop. However, the control hazard created by the branch back to the start of the loop makes it hard for processors to issue instructions from multiple loop iterations simultaneously. Unrolling a loop merges several iterations into one straight-line section of code that the processor or compiler can examine to locate independent instructions. This generally increases the amount of instruction-level parallelism in the program (number of instructions that can be executed per cycle), improving performance. Loop unrolling also reduces the number of conditional branch instructions executed during the execution of the loop, further improving performance.

Loop Unrolling (II)

7.18 Show how a compiler would unroll the following infinite loop 4 times. Be sure to include the preamble code (the code that computes all of the pointers required for the operations within each iteration of the unrolled loop). Assume that the processor has as many architectural registers as required.

```
loop:
   LD r1, (r2)
   LD r3, (r4)
   LD r5, (r6)
   ADD r1, r1, r3
   ADD r1, r1, r5
   DIV r1, r1, r7
   ST (r0), r1
   ADD r2, #4, r2
   ADD r4, #4, r4
   ADD r6, #4, r6
   ADD r0, #4, r0
   BR loop
```

Solution

Here is an example of how the compiler might unroll the loop. The key elements in the loop unrolling are the preamble, to generate the pointers required by the unrolled loop, incrementing all of the pointers by 16 instead of 4 in each unrolled iteration because the unrolled iteration contains 4 of the original iterations, and realizing that r7 does not change from iteration to iteration of the original loop, so we do not need multiple registers to hold the value in r7 during different iterations of the original loop. There are many different ways in which this loop could be unrolled. Any solution that incorporates the key elements described above and performs the work of 4 iterations of the old loop in each iteration of the unrolled one is correct.

In addition to the basic unrolling, this example moves all of the loads in the unrolled loop to the beginning of the loop, and it schedules as many operations as possible between the divides and the stores that write the results of the divides to memory. These reorderings will improve the performance of the loop by giving the loads and divides, which are often long-latency operations, more time to complete before their results are needed.

```
preamble:
   ADD r8, #4, r0
   ADD r10, #4, r2
   ADD r12, #4, r4
   ADD r14, #4, r6
   ADD r16, #8, r0
   ADD r18, #8, r2
   ADD r20, #8, r4
   ADD r22, #8, r6
   ADD r24, #12, r0
   ADD r26, #12, r2
   ADD r28, #12, r4
   ADD r30, #12, r6
```

```
loop:
   LD r1, (r2)
   LD r3, (r4)
   LD r5, (r6)
   LD r9, (r10)
   LD r11, (r12)
   LD r13, (r14)
   LD r17, (r18)
   LD r19, (r20)
   LD r21, (r22)
   LD r25, (r26)
   LD r27, (r28)
   LD r29, (r30)
   ADD r1, r1, r3
   ADD r1, r1, r5
   DIV r1, r1, r7
   ADD r9, r9, r11
   ADD r9, r9, r13
   DIV r9, r9, r7
   ADD r17, r17, r19
   ADD r17, r17, r21
   DIV r17, r17, r7
   ADD r25, r25, r27
   ADD r25, r25, r29
   DIV r25, r25, r7
   ADD r2, #16, r2
   ADD r4, #16, r4
   ADD r6, #16, r6
   ADD r10, #16, r10
   ADD r12, #16, r12
   ADD r14, #16, r14
   ADD r18, #16, r18
   ADD r20, #16, r20
   ADD r22, #16, r22
   ADD r26, #16, r26
   ADD r28, #16, r28
   ADD r30, #16, r30
   ST (r0), r1
   ADD r0, #16, r0
   ST (r8), r9
   ADD r8, #16, r8
   ST (r16), r17
   ADD r16, #16, r16
   ST (r24), r25
   ADD r24, #16, r24
   BR loop
```

Impact of Loop Unrolling on Execution Time

7.19 Show how a compiler would schedule the original and unrolled versions of the loop from the previous exercise for execution on a 4-wide VLIW processor that can execute an instruction on any execution unit. Assume latencies of 3 cycles for LD operations, and 2 cycles for DIVs and ADDs. Assume that the branch delay of the processor is long enough that all operations in one iteration complete before the next iteration starts. As in the other VLIW problems, assume that the compiler examines all possible operation orderings to find one that fits into the fewest number of instructions. For the unrolled loop, schedule only the loop body, not the preamble.

Solution

In both parts of this problem, there are many ways to convert the loop for execution on the VLIW in the minimum number of instructions. Here, we present examples of how the loop could be placed in the minimum number of instructions, but any solution that achieves this number of instructions without violating the loop's data dependencies is correct.

Original loop: 10 instructions.

Instruction	op1	op2	op3	op4
1	LD r1, (r2)	LD r3, (r4)	LD r5, (r6)	ADD r2, #4, r2
2	ADD r4, #4, r4	ADD r6, #4, r6	NOP	NOP
3	NOP	NOP	NOP	NOP
4	ADD r1, r1, r3	NOP	NOP	NOP
5	NOP	NOP	NOP	NOP
6	ADD r1, r1, r5	NOP	NOP	NOP
7	NOP	NOP	NOP	NOP
8	DIV r1, r1, r7	NOP	NOP	NOP
9	NOP	NOP	NOP	NOP
10	ST (r0), r1	BR loop	ADD r0, #4, r0	NOP

Unrolled loop: 12 instructions. By unrolling the loop, we have managed to do 4 times as much work in each iteration, with only a 20 percent decrease in the execution time of an iteration.

Instruction	op1	op2	op3	op4
1	LD r1, (r2)	LD r3, (r4)	LD r9, (r10)	LD r11, (r12)
2	LD r17, (r18)	LD r19, (r20)	LD r25, (r26)	LD r27, (r28)
3	LD r5, (r6)	LD r13, (r14)	LD r21, (r22)	LD r29, (r30)
4	ADD r1, r1, r3	ADD r9, r9, r11	ADD r2, #16, r2	ADD r4, #16, r4
5	ADD r17, r17, r19	ADD r25, r25, r27	ADD r10, #16, r10	ADD r12, #16, r12
6	ADD r1, r1, r5	ADD r9, r9, r13	ADD r18, #16, r18	ADD r20, #16, r20
7	ADD r17, r17, r21	ADD r25, r25, r29	ADD r26, #16, r26	ADD r28, #16, r28
8	DIV r1, r1, r7	DIV r9, r9, r7	ADD r6, #16, r6	ADD r14, #16, r14
9	DIV r17, r17, r7	DIV r25, r25, r7	ADD r22, #16, r22	ADD r30, #16, r30
10	ST (r0), r1	ST (r8), r9	ADD r0, #16, r0	ADD r8, #16, r8
11	ST (r16), r17	ST (r24), r25	ADD r16, #16, r16	ADD r24, #16, r24
12	BR loop	NOP	NOP	NOP