CHAPTER 5

Processor Design

5.1 Objectives

This chapter provides an introduction to processor design, breaking down some o the abstractions we have used in previous chapters and preparing for the next tw chapters, which discuss techniques for improving processor performance. Afte completing this chapter, you should

1. Understand the difference between instruction set architecture and proces sor microarchitecture
2. Be familiar with the difference between RISC and CISC instruction set and be able to convert program fragments written for one style fo execution on the other
3. Understand addressing modes and how they impact performance
4. Understand the basics of register file design, and be able to discuss hov trade-offs in register file organization affect the implementation cost of th register file

5.2 Introduction

This chapter begins a three-chapter sequence on processor design by coverin instruction set architecture and the basics of processor microarchitecture. The nex chapter builds on this by discussing pipelining, a technique that improves perfor mance by overlapping the execution of multiple instructions. Chapter 7 complete our coverage of processor architecture by discussing instruction-level parallelism.

Processor design has typically been divided into two subcategories: instructio set architecture and processor microarchitecture. Instruction set architecture refers the design of the set of operations that the processor executes and includes th choice of programming model, number of registers, and decisions about how data accessed. Processor microarchitecture describes how instructions are implemente

Processor Design
↳ ISA
↳ Processor micro arch

ISA → operations executed by processor.
↳ choice of prog model, # of reg, how data is accessed

and includes factors such as how long it takes to execute instructions, how many instructions may be executed at one time, and how processor modules such as the register file are designed. These definitions are somewhat vague, and there is a great deal of overlap between the two areas, so it is often difficult to decide whether a given aspect of computer architecture counts as instruction set architecture or microarchitecture. A good working definition is that any aspect of the processor that an assembly-language programmer needs to know about to write a correct program is part of the instruction set architecture, and that any aspect that only affects performance, not correctness, is part of the microarchitecture.

Processor microarch
how inst are imp
and CPI, length of
clock cycle.
How many inst are
executed in one time.
How are processor
modules designed.

5.3 Instruction Set Architecture

When most computer programming was done in assembly language, instruction set architecture was considered the most important part of computer architecture, because it determined how difficult it was to obtain optimal performance from the system. Over the years, instruction set architecture has become less significant, for several reasons. First, most programming is now done in high-level languages, so the programmer never interacts with the instruction set. Second, and more significant, consumers have come to expect *compatibility* between different generations of a computer system, meaning that they expect programs that ran on their old system to run on their new system without changes. As a result, the instruction set of a new processor is often required to be the same as the instruction set of the company's previous processor, sometimes with a few additional instructions, meaning that most of the design effort for a processor goes into improving the microarchitecture to increase performance.

In the previous chapter, we covered one of the most significant decisions involved in designing a processor's instruction set architecture (ISA)—the selection of a programming model. As was discussed in that chapter, the GPR programming model has become dominant and will be assumed for the remainder of this book. In this chapter, we cover four of the most significant remaining issues in instruction set architecture: the RISC versus CISC debate, selection of addressing modes, the use of fixed- or variable-length instruction encodings, and multimedia vector instructions.

5.3.1 RISC VERSUS CISC

Before the 1980s, there was a great deal of focus on reducing the "semantic gap" between the languages used to program computers and machine languages. It was believed that making machine languages more like high-level programming languages would result in better performance by reducing the number of instructions required to implement a program and would make it easier to compile high-level language programs into machine language. The end result of this was the design of instruction sets that contained very complex instructions.

Moving towards
HLL from ML.

As compiler technology improved, researchers began to consider whether systems with complex instructions, known as complex instruction set computers (CISC), delivered better performance than systems based around simpler instruction sets. This second class of system was known as reduced instruction set computers (RISC).

CISC use lesser instr to process a Program.

The primary argument in favor of CISC computers is that CISC computers generally require fewer instructions than RISC computers to perform a given computation, so a CISC computer will have higher performance than a RISC computer that executes instructions at the same rate. In addition, programs written for CISC architectures tend to take less space in memory than the same programs written for a RISC architecture. The main argument in favor of RISC computers is that their simpler instruction sets often allow them to be implemented at higher clock rates than CISC computers, allowing them to execute more instructions in the same amount of time. If a RISC processor's increased clock rate allows it to execute its programs in less time than a CISC processor takes to execute its programs (which require fewer instructions), the RISC processor will have higher performance.

CISC programs take less space in memory

Risc have higher performance beause it has increased clock Rate.

There was a great deal of controversy in the computer architecture community during the 1980s and early 1990s about which of the two approaches was better, and depending on how you look at it, either approach can be said to have won. The vast majority of ISAs introduced since the 1980s have been RISC architectures, arguing that RISC is superior. On the other hand, the Intel x86 (IA-32) architecture, which uses a CISC instruction set, is the dominant PC/workstation architecture in terms of number of processors sold, arguing that CISC architectures have won.

Over the last 20 years, there has been a fair amount of convergence between RISC and CISC architectures, making it somewhat difficult to determine whether an architecture is RISC or CISC. RISC architectures have incorporated some of the most useful complex instructions from CISC architectures, relying on their micro-architecture to implement these instructions with little impact on the clock cycle, and CISC architectures have dropped complex instructions that were not used sufficiently often to justify their implementation.

Risc are load and store Architectures Only load & stores access memory.

One clear delineation between RISC and CISC architectures is that RISC architectures are *load-store architectures*, meaning that only load and store instructions may access the memory system. The GPR architecture described in Chapter 4 is a load-store architecture.

In CISC, some inst. Can read and write i/p n o/p directly into memory

In many CISC architectures, arithmetic and other instructions may read their inputs from or write their outputs to the memory system, instead of the register file. For example, a CISC architecture might allow an ADD operation of the form ADD (r1), (r2), (r3), where the parentheses around a register name indicates that the register contains the address in memory where the operand can be found or the result should be placed. Using this notation, the instruction ADD (r1), (r2), (r3) instructs the processor to add the value contained in the memory location whose address is stored in r2 to the value contained in the memory location whose address is stored in r3, and store the result into memory at the address contained in r1.

The difference between load-store architectures and architectures that can merge memory references with other operations is an excellent example of the trade-offs

between RISC and CISC architectures. Because RISC architectures are implemented using the load-store model, a RISC processor would require several instructions to implement the single CISC ADD operation described above. However, the hardware required to implement the CISC processor would be more complex, since it would have to be able to fetch instruction operands from memory, so the CISC processor would probably have a longer cycle time (or would require more cycles to execute each instruction) than the RISC processor.

EXAMPLE

In our GPR load-store architecture, how many instructions are required to implement the same function as the CISC ADD operation described above? Assume that the appropriate memory addresses are present in r1, r2, and r3 at the start of the instruction sequence.

Solution

Four instructions are required:

```
LD r4, (r2)
LD r5, (r3)
ADD r6, r4, r5
ST (r1), r6
```

This example shows that a RISC architecture may require many more operations to implement a function than a CISC architecture, although it is something of an extreme example. It also shows that RISC architectures generally require more registers to implement a function than CISC architectures, since all of the inputs of an instruction must be loaded into the register file before the instruction can execute. However, RISC processors have the advantage that breaking a complex CISC operation into multiple RISC operations can allow the compiler to schedule the RISC operations for better performance. For example, if memory references take multiple cycles to execute (as they generally do), a compiler for a RISC architecture can place other instructions between the LD instructions in the example and the ADD. This gives the LD instructions time to complete before their results are needed by the ADD, preventing the ADD from having to wait for its inputs. In contrast, the CISC instruction has no choice but to wait for its inputs to come back from the memory system, potentially delaying other instructions.

5.3.2 ADDRESSING MODES

As we have discussed, one of the major differences between RISC and CISC architectures is the set of instructions that can access memory. A related issue that affects both RISC and CISC architectures is the choice of which *addressing modes* the architecture supports. An architecture's addressing modes are the set of syntaxes and methods that instructions use to specify a memory address, either as the target address of a memory reference or as the address that a branch will jump to.

Set of syntaxes through which target-references are made.

Instruction formats that are readable / understandable by any architecture.

Depending on the architecture, some of the addressing modes may only be available to some of the instructions that reference memory. Architectures that allow any instruction that references memory to use any addressing mode are described as *orthogonal*, because the choice of addressing mode is independent from the choice of instruction.

So far, we have used only two addressing modes: register addressing for load instructions, store instructions, and CISC instructions that reference memory; and label addressing for branch instructions. In register addressing, an instruction reads the value out of a register and uses that as the address of the memory reference or branch target. We use the syntax (rx) to indicate that the register addressing mode is being used. An instruction set that provided only register addressing would be possible, since any address could be computed using arithmetic instructions and stored in a register. Processors provide other addressing modes because they reduce the number of instructions required to compute addresses, thereby improving performance.

The second addressing mode that we have seen so far is label addressing, in which a branch instruction specifies its destination as a label that is placed on an instruction elsewhere in the program. As was discussed in the last chapter, these text labels do not appear in the machine-language version of the program. In fact, most branch instructions do not explicitly contain their destination addresses at all. Instead, the assembler/linker translates the label into an *offset* (which can be either positive or negative) from the location of the branch instruction to the location of its target. In effect, the branch instruction tells the processor how far away the target instruction is from it, rather than exactly where the target instruction is located. The processor adds the offset to the branch instruction's PC to get the destination address of the branch.

Using offsets rather than explicit addresses for label addressing has two advantages. First, it reduces the number of bits required to encode the instruction. Most branches have targets that are relatively close to the branch, so a small number of bits can be used to encode the offset. When a branch has a target that is far away, the target address can be calculated by other instructions and a register or similar addressing mode used. Second, using offsets instead of explicit addresses in branch instructions allows the loader to place the program at different locations in the memory without having to change the program. If explicit addresses were used, the destination address of each branch would have to be recomputed every time the program was loaded. This feature is particularly useful for operating system libraries that are dynamically linked into the program at runtime, as these libraries have no ability to predict which address they will be loaded into.

EXAMPLE

An assembly-language "BR label1" instruction is assembled and linked as part of a larger program. The linker calculates the offset from the branch instruction to label1 as 0x437 bytes. If the branch instruction is loaded into address 0x4000, what is the target address of the branch? How about if the instruction is loaded into address 0x4400?

Solution

The target address of the branch is the sum of the address of the branch (the PC when the branch executes) and the offset. When the branch is loaded into address 0x4000, the target address is 0x4437 (0x4000 + 0x437). When the branch is loaded into address 0x4400, the target address is 0x4837.

Another addressing mode that is provided by many processors is *register plus immediate addressing*. In register plus immediate mode, which is typically expressed as imm(rx), the value of the specified register is added to the immediate (constant) value specified in the instruction to generate a memory address.

EXAMPLE

If the value of r4 is 0x13000, what address is referenced by the instruction LD −0x80(r4)?

Solution:

The label plus immediate addressing mode adds the value of the immediate to the value of the register to get the destination address. Adding −0x80 to 0x13000 gives 0x12F80, the address referenced by the load.

Register plus immediate mode is extremely useful in accessing data structures, which tend to have fields that are located at fixed offsets from the start of the data structure. Using this addressing mode, a program that needs to reference different fields of a data structure can simply load the starting address of the data structure into a register and then use register plus immediate mode to access the different fields of the data structure, reducing the number of instructions required to perform address computations and the number of registers required to store addresses.

Many other addressing modes have been implemented in ISAs over the years. In general, they are variations on the register plus immediate mode. For example, some ISAs allow you to add a label offset and an immediate, or a label offset, a register value, and an immediate.

One problem with all addressing modes that compute their address rather than taking it straight from a register is that these addressing modes increase the execution time of instructions that use them, since the processor must perform a computation before the address can be sent to the memory system. To provide flexibility in addressing without increasing memory latency, some architectures provide *postincrementing* addressing modes instead of register plus immediate-style addressing modes. These addressing modes, which we will use the syntax imm[rx][1] for, read their address out of the specified register, send that address to the memory system, and then add the specified immediate to the value of the register. This result is then written back into the register file. Because the address is sent directly from

[1] Unlike the other syntaxes we use for addressing modes, this syntax is not standard for the industry. Different architectures use different syntaxes to describe postincrement addressing. For example, at least one architecture has separate instructions for postincrementing and nonpostincrementing addressing modes.

the register file to the memory system, these instructions execute more quickly than register plus immediate addressing mode instructions, but they still reduce the number of instructions required to implement a program as compared to ISAs that only provide register addressing.

Effectively, each use of a postincrementing addressing mode computes the address for the next instruction to reference the register. Thus, a sequence of register plus immediate addressing mode references can be easily transformed into a sequence of postincrementing addressing mode references. Postincrementing addressing mode is also useful when accessing an array of equally sized data structures, as the last reference to each address can increment the register containing the address to point at the next data structure.

EXAMPLE

Convert the following sequence of instructions written for an ISA that provides register plus increment addressing mode to run on an ISA that has only register and postincrementing addressing modes. (*Hint*: Exactly one additional instruction is required.)

```
LD r4, 8(r1)
LD r5, 12(r1)
ST 16(r1), r8
```

Solution

The thing to remember here is that postincrementing addressing mode instructions change the value of their address register. Register plus increment addressing mode just computes the address to be sent to the memory system, leaving the value in the address register the same. Thus, each postincrementing instruction needs to increment by the offset from the address it references to the address referenced by the next instruction, which will generally be different than the immediate added to the address register in register plus immediate mode. This gives the following code sequence:

```
  ADD r1, r1, #8 /* increment the original address register
by the offset to the address used by the first LD. */
  LD r4, 4[r1] /* The second LD has an offset of 12 bytes
from the original value of r1. Since we've already offset
r1 by 8 bytes, just increment by 4 more */
  LD r5, 4[r1] /* Just need to add 4 more bytes to get from
the address referenced by this load to the one referenced
by the store */
  ST (r1), r8 /* No need to postincrement */
```

5.3.3 MULTIMEDIA VECTOR INSTRUCTIONS

Many processor families have recently added multimedia vector instructions to their ISAs. These instructions, which include the MMX extensions to the x86 ISA, are

intended to improve performance on multimedia applications, such as video decompression and audio playback. These applications have several traits that make it possible to significantly improve their performance with a small number of new instructions. First, they perform the same sequence of operations on a large number of independent data objects, such as 8 × 8 blocks of compressed pixels. This trait is often described as *data parallelism*, because multiple data objects can be processed at the same time. The second important trait of these applications is that they operate on data that is much smaller than the 32-bit or 64-bit data words found in most modern processors. Video pixels, which are often described by 8-bit red, green, and blue color values, are an example of this. Each of the color values is generally computed independently, meaning that 24 bits of a 32-bit ALU are idle during the computation.

Multimedia vector instructions treat the processor's data word as a collection of smaller data objects, as shown in Fig. 5-1, which shows how a multimedia vector instruction might process a 32-bit data word. Instead of operating on a 32-bit quantity, the data word is treated as a collection of four 8-bit quantities or two 16-bit quantities. Most of the multimedia vector instruction sets can operate on longer data types, such as 64-bit or 128-bit quantities, allowing more operations to be done in parallel.

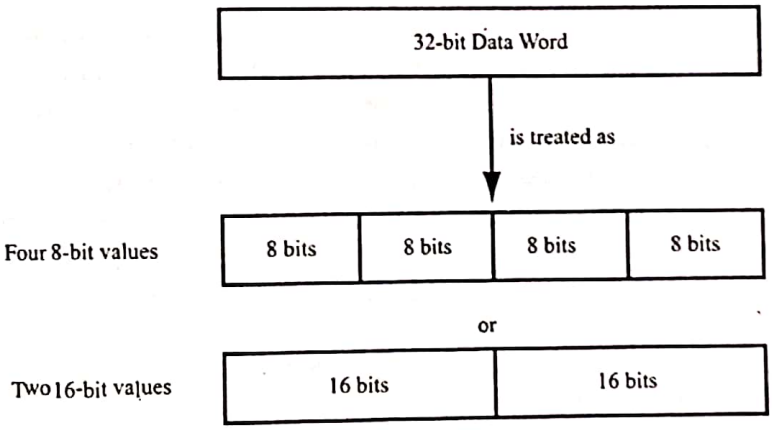


Fig. 5-1. Multimedia vector data types.

Many multimedia vector instructions allow the option to operate in *saturating arithmetic mode*. In saturating arithmetic, computations that overflow the number of bits in their representation return the maximum value that the representation can represent, and computations that underflow return 0. For example, adding 0xaa and 0xbc in 8-bit saturating arithmetic has a result of 0xff instead of 0x66. Saturating arithmetic is useful when it is desirable to have a computation be limited by its maximum value. For example, increasing the amount of red in a pixel that is already extremely red should result in a pixel that has the maximum allowable amount of redness, instead of a pixel that has very little redness because the computation has wrapped around to a small value.

When a multimedia vector instruction executes, it performs its computation in parallel on each of the smaller objects within its input word, as shown in Fig. 5-2, which illustrates a 32-bit vector addition, treating the 32-bit word as four 8-bit

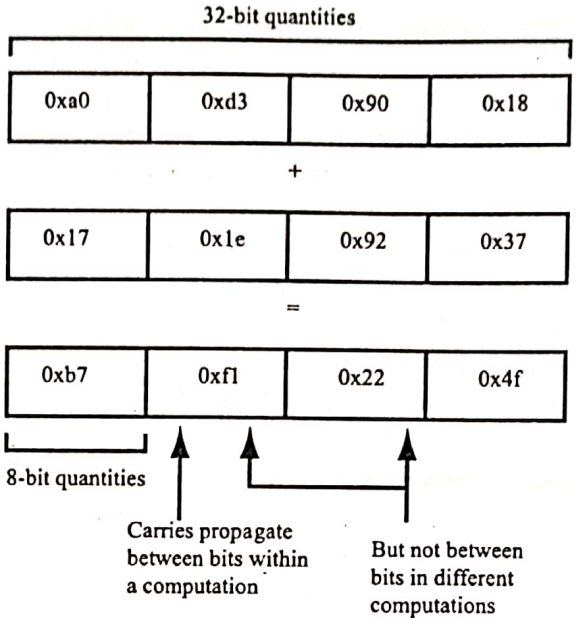


Fig. 5-2. Multimedia vector addition.

unsigned quantities. Saturating arithmetic is not used in this example. Each of the four parallel additions executes in parallel, and the results of any one addition do not affect the other additions. In particular, note that carries propagate normally within each 8-bit computation but do not propagate between different computations.

Multimedia vector instructions can significantly improve a processor's performance on data-parallel applications that operate on small data types by allowing multiple computations to be performed in parallel. For example, 32-bit multimedia vector operations can be used to compute the red color values of four different pixels simultaneously, potentially improving performance by up to a factor of 4. The hardware required to implement multimedia vector operations is typically fairly small, as most of the hardware required to implement a processor's nonvector operations can be reused, making these operations attractive to computer architects who expect their processors to be used for data-parallel applications.

5.3.4 FIXED-LENGTH VERSUS VARIABLE-LENGTH INSTRUCTION ENCODINGS

Once the set of instructions that a processor will support has been selected, a computer architect must select an encoding for the ISA, which is the set of bits that will be used to represent the instructions in the memory of the computer. Generally, architects want to find an encoding that is both compact and requires little logic to decode, meaning that it is simple for the processor to figure out which instruction is represented by a given bit pattern in the program. Unfortunately, these two goals are somewhat in conflict.

Fixed-length instruction set encodings use the same number of bits to encode each instruction in the ISA. Fixed-length encodings have the advantage that they are simple to decode, reducing the amount of decode logic required and the latency of

the decode logic. Also, a processor that uses a fixed-length ISA encoding can easily predict the location of the next instruction to be executed (assuming that the current instruction is not a branch). This makes it easier for the processor to use pipelining, the subject of the next chapter, to improve performance by overlapping the execution of multiple instructions.

Variable-length instruction set encodings use different numbers of bits to encode the instructions in the ISA, depending on the number of inputs to the instruction, the addressing modes used, and other factors. Using a variable-length encoding, each instruction takes only as much space in memory as it requires, although many systems require that all instruction encodings be an integer number of bytes long. Using a variable-length instruction set can reduce the amount of space taken up by a program, but it greatly increases the complexity of the logic required to decode instructions, since parts of the instruction, such as the input operands, may be stored in different bit positions in different instructions. Also, the hardware cannot predict the location of the next instruction until the current instruction has been decoded enough to know how long the current instruction is.

Given the pros and cons of fixed- and variable-length instruction encodings, fixed-length encodings are more common in recent architectures. Variable-length encodings are mainly used in architectures where there is a large variance between the amount of space required for the longest instruction in the ISA and the average instruction in the ISA. Examples of this include stack-based architectures, because many operations do not need to specify their inputs, and CISC architectures, which often contain a few instructions that can take a large number of inputs.

This ends our discussion of instruction set architecture. We have covered the RISC versus CISC debate, one of the most famous controversies in computer architecture, the impact of addressing modes on a processor's ISA, and the pros and cons of fixed-length and variable-length instruction encodings. Also, we have prescribed a brief introduction to multimedia vector instructions, a recent extension to processor ISAs that improves performance on data-parallel applications.

The rest of this chapter will begin our discussion of processor microarchitecture, which will continue in the next two chapters. We will start with a more in-depth discussion of how processors execute instructions than has been presented so far, and we will continue with a discussion of register file design. For simplicity, we will assume a RISC-style processor for the rest of our discussion of processor architecture, although the concepts we will cover are generally applicable to CISC architectures as well.

5.4 Processor Microarchitecture

As described earlier, processor microarchitecture includes all of the details about how a processor is implemented. The ISA specifies how the processor is programmed, and the microarchitecture specifies how it is built. Obviously, the ISA has a great deal of impact on the microarchitecture. An ISA that contains only simple operations can be implemented using a simple, straightforward micro-

architecture, while an ISA containing complex operations generally requires a complex microarchitecture to implement.

In Chapter 3, we presented the processor block diagram that is reproduced here as Fig. 5-3. This diagram breaks the processor down into three main subsystems: the execution units, the register file, and the control logic. Together, the execution units and the register file are often described as the processor's *datapath*, as data and instructions flow through them in a regular fashion. The control logic is more irregular and very processor-specific.

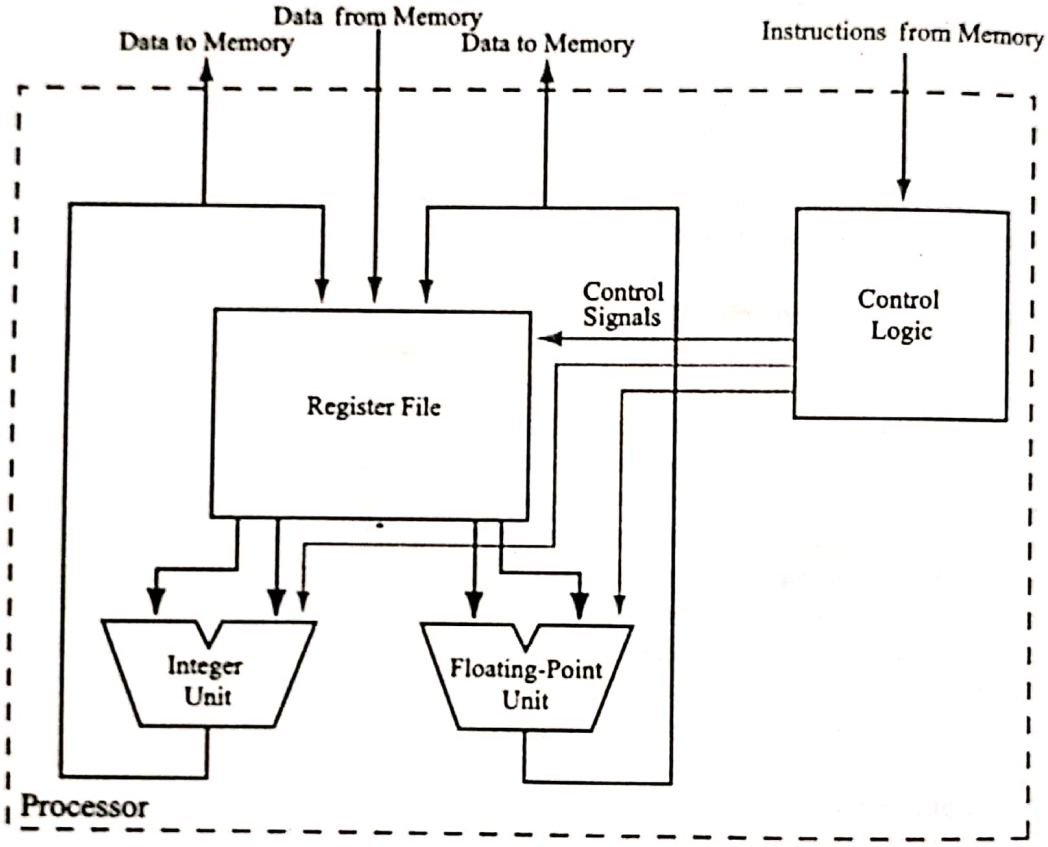


Fig. 5-3. Processor block diagram.

5.4.1 EXECUTION UNITS

Figure 5-4 shows the steps involved in executing an instruction and how the different modules of the processor interact during instruction execution. First, the processor fetches the instruction from the memory. The instruction is then *decoded* to determine what instruction it is and what its input and output registers are. The decoded instruction is represented as a set of bit patterns that tell the hardware how to execute the instruction. These bit patterns are sent on to the next section of the execution unit, which reads the instruction's inputs from the register file. The decoded instruction and the values of the input registers are forwarded to the hardware that computes the result of the instruction, and the result is written back into the register file.

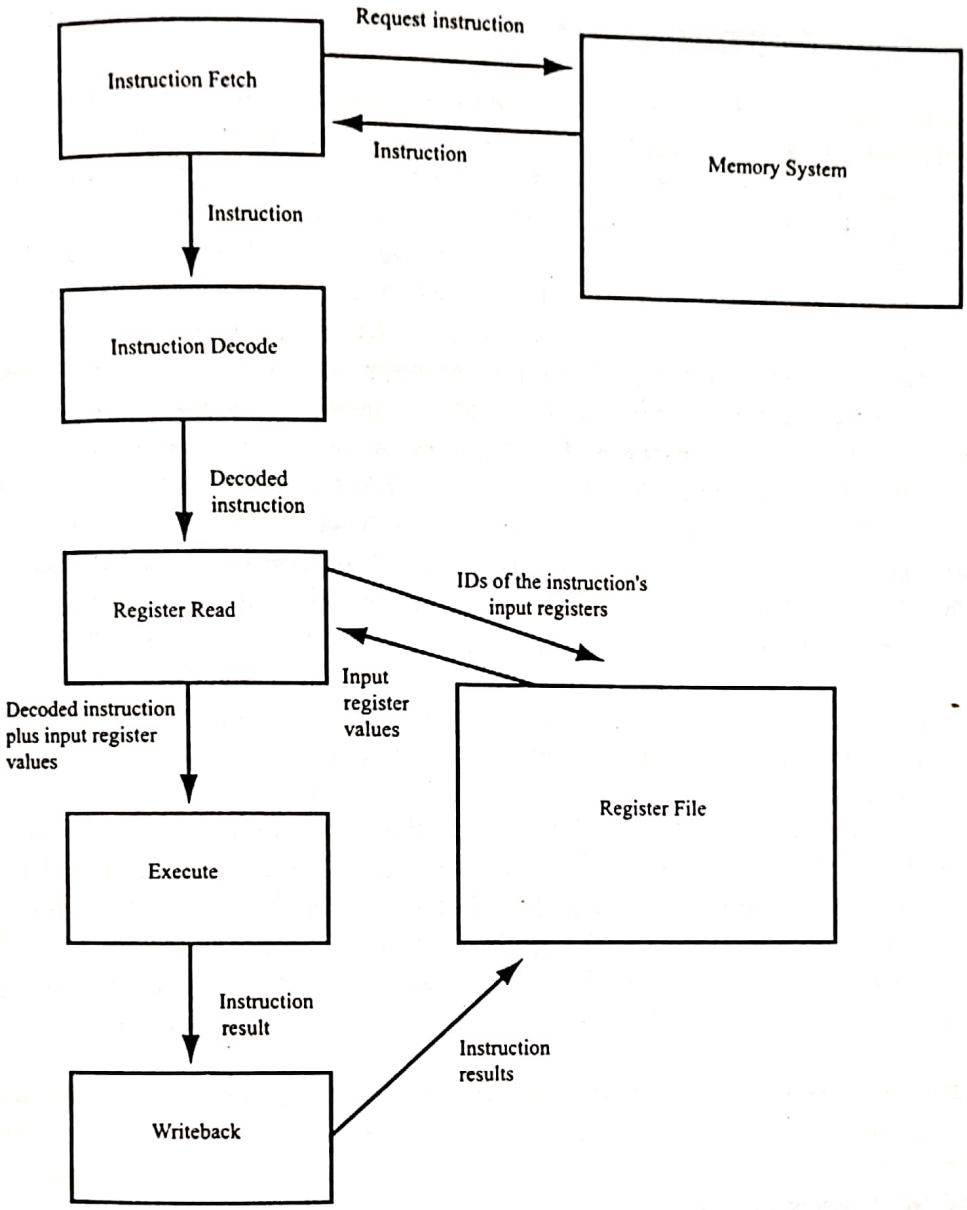


Fig. 5-4. Instruction execution.

Instructions that access the memory system have a similar execution flow, except the output of the execution unit is sent to the memory system, either as the address of a load operation or as the address and data of a store operation. When the result of a load returns from the memory system, it is written into the register file, similar to the way the result of a computation is written into the register file.

Many execution units are implemented using a physical structure similar to the one shown in the figure. Modules that implement the different steps in instruction execution are physically laid out next to each other in a line, with short wires connecting them. As the instruction executes, data flows through the line from one module to the next, with each module performing its task in sequence.

5.4.2 MICROPROGRAMMING

Microprogramming

In a *microprogrammed* processor, the hardware does not directly execute the instructions in the ISA. Instead, the hardware executes very simple micro-operations, and each instruction specifies a sequence of micro-operations that are used to implement the instruction. Essentially, each instruction in the ISA is translated into a short program of microinstructions by the hardware, similar to the way a compiler translates each instruction in a high-level language program into a sequence of assembly-language instructions. For example, a microprogrammed processor might translate the instruction ADD r1, r2, r3 into six micro-operations: one that reads the value of r2 and sends it to one input of the adder, one that reads the value of r3 and sends it to the other input of the adder, one that performs the actual addition, one that writes the result of the addition into r1, one that increments the PC to point to the next instruction, and one that fetches the next instruction from the memory. Each micro-operation generally takes one processor cycle to execute, so an ADD instruction would require six cycles to complete in such a system.

Microprogrammed processors contain a small memory that holds the sequences of microinstructions used to implement each instruction in the ISA. To execute an instruction, a microprogrammed processor accesses this memory to locate the set of microinstructions required to implement the ISA instruction, and then executes the microinstructions in sequence.

Microprogramming became popular because the technologies used to implement early computers (vacuum tubes, discrete transistors, and small-scale integrated circuits) limited the amount of hardware that could be built into the processor, and computer architects wanted to design ISAs with complex instructions to reduce the number of instructions required to implement a program. By using microprogramming, architects could build simple hardware and then microprogram that hardware to execute complex instructions.

Modern processors tend not to use microprogramming for two reasons. First, it is now practical to implement most processors's ISAs directly in hardware because of advances in VLSI technology, making microcode unnecessary. Second, microprogrammed processors tend to have lower performance than nonmicroprogrammed processors because of the overhead involved in fetching each microinstruction from the microinstruction memory.

5.4.3 REGISTER FILE DESIGN

So far, we have treated the register file as a single device that contains both integer and floating-point data. Most processors do not implement their register files in this fashion. Instead, they implement separate register files for floating-point and integer data. Integer register files are referenced using the "rx" syntax we have used so far for register names, while floating-point registers are referenced as "fx." Using this syntax makes it clear which register file is being referenced for instructions, such as loads and stores, that may need to reference both register files. Arithmetic instructions are generally restricted to accessing the appropriate register file for

the type of computation they perform, although some arithmetic instructions are allowed to transfer data between register files.

Processors implement separate register files for two reasons. First, it allows the register files to be placed physically close to the execution units that need them. The integer register file can be placed close to units that perform integer operations, and the floating-point register file can be placed close to the floating-point execution units. This reduces the length of the wires that connect the register files to the execution units, and therefore the amount of time required to send data from the register file to the execution units.

The second reason why processors implement separate integer and floating-point register files is that separate register files take up less space on processors that execute more than one instruction per cycle. The details of this are beyond the scope of this book, but the size of a register file grows as approximately the square of the number of simultaneous reads and writes that the register file allows. To execute one instruction per cycle, a register file typically needs to allow two reads and one write per cycle, since some arithmetic operations read two registers and write one. Each additional operation that the processor wants to execute in a cycle typically increases the number of simultaneous reads/writes (called ports) by three. Dividing the register file into integer and floating-point register files reduces the number of ports required on each register file. Since the area of a register file grows faster than linearly with the number of ports, the two separate register files take up less area than one register file that provides the same number of ports.

Port.

5.5 Summary

The goal of this chapter has been to provide an introduction to processor design, in preparation for the next two chapters, which provide more in-depth discussions of two techniques that are widely used to improve processor performance: pipelining and instruction-level parallelism. The chapter began with a discussion of the distinction between instruction set architecture and processor microarchitecture. Instruction set architecture is the design of the instructions that a processor provides, including the programming model, the set of operations provided, the addressing modes that the processor supports, and the choice of which instructions may access memory. Processor microarchitecture covers the details of how the processor is implemented. In general, instruction set architecture refers to any aspect of the architecture that is visible to an assembly-language programmer, while processor microarchitecture covers details that affect how quickly a program executes. There is substantial overlap between these two categories. For example, an instruction set architecture that provides a number of complex instructions may require a processor microarchitecture that gives lower performance than a microarchitecture that implements only simpler instructions.

Within instruction set architecture, we discussed the distinction between RISC and CISC instructions, the central element of which is the requirement that RISC architectures be load-store architectures, while most CISC architectures allow other operations to reference memory as well. The impact of addressing modes and

instruction set encodings on program size, performance, and the complexity of the hardware required to implement the processor was discussed. Finally, an introduction to multimedia vector instructions, a relatively new addition to many instruction sets, was provided.

Our introduction to processor microarchitecture included a block diagram of the data flow through a processor, with a discussion of the function of each element in the flow. We then briefly discussed microprogramming, a technique commonly used to implement processors in the past, but one that is rarely used today because of its impact on performance. Our processor microarchitecture discussion then concluded with coverage of the trade-offs involved in register file design.

In the next chapter, we will discuss pipelining, a technique that improves processor performance by overlapping the execution of multiple instructions. This allows a higher clock rate and improves the rate at which instructions are executed. Pipelining is often combined with instruction-level parallelism, the subject of Chapter 7, to produce processors that issue multiple instructions in each cycle and overlap instruction execution to increase the number of clock cycles in a second.

Solved Problems

Instruction Set Architecture

5.1. What is a load-store architecture, and what are the pros and cons of such an architecture as compared to other GPR architectures?

Solution

A load-store architecture is one in which only load and store instructions can access the memory system. In other GPR architectures, some or all of the other instructions may read their operands from or write their results to the memory system. The primary advantage of non-load-store architectures is the reduced number of instructions required to implement a program and lower pressure on the register file. The advantage of load-store architectures is that limiting the set of instructions that can access the memory system makes the microarchitecture simpler, which often allows implementation at a higher clock rate. Depending on whether the clock rate increase or the decrease in number of instructions is more significant, either approach can result in greater performance.

RISC versus CISC (I)

5.2. Rewrite the following CISC-style program fragment so that it executes correctly on a RISC (load-store) processor that executes the GPR ISA outlined in the previous chapter. Assume that the GPR ISA provides only the register addressing mode, and that there are enough registers in the processor to hold any temporary values that you need to generate.

```
ADD r3, (r1), (r2)
```

Solution

This program fragment makes four memory references as part of arithmetic operations. To execute the fragment, we need to replace each of those with an explicit load or store. Here's a code fragment that does that:

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

RISC versus CISC (II)

5.3. Rewrite the following program fragment that is written using the GPR instruction set for execution on a CISC processor that provides the same instruction set as the GPR processor but allows the register addressing mode to be used on the input operands or destination of any instruction. (Yes, the code fragment will execute correctly as written on such a processor. Your goal should be to reduce the number of instructions as much as possible.) Assume that the program ends after the last instruction in the fragment, so that the only goal of the program should be to have the correct value written into memory at the end

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

Solution

The general approach here is to replace all of the load and store instructions with memory references in arithmetic instructions. A program that does this is as follows:

```
DIV r9, (r2), (r4)
ADD r10, r9, (r6)
SUB (r12), (r8), r10
```

Addressing Modes (I)

5.4. Why does adding addressing modes like register plus immediate to an ISA tend to improve performance?

Solution

Adding additional addressing modes to an ISA tends to improve performance by reducing the number of instructions required to compute addresses. For example, if a data structure contains four data words, register plus immediate addressing can be used to access all of them, with only one address computation required to change the pointer to point to the next data structure. If an architecture only provided the register addressing mode, an ADD instruction would be required to calculate the address of each element in the data structure.

Addressing Modes (II)

5.5. Why are postincrementing addressing modes often found on processors that need to have a very short cycle time?

Solution

Postincrementing addressing modes allow different elements of a data structure to be accessed without explicit instructions to compute addresses, but they don't require that the processor perform an addition before sending the address to the memory system, because they add the immediate offset to the contents of the register after the address is sent to the memory system. This allows memory-referencing instructions to compute addresses that will be used by later memory-referencing instructions without increasing the latency to complete a memory reference.

Addressing Modes (III)

5.6. Rewrite the following program fragment to take advantage of register plus immediate addressing mode. Assume that no register values are used outside of the program fragment and that the code fragment will execute on a load-store processor.

```
ADD r2, r3, #8
LD r4, (r2)
ADD r1, r4, r8
ADD r5, r3, #16
LD r6, (r5)
MUL r7, r1, r6
ADD r9, r3, #24
ST (r9), r7
```

Solution

All of the ADD operations that take r3 as an input can be folded into the LD or ST operations that follow them using register plus immediate addressing mode, to give the following program:

```
LD r4, 8(r3)
ADD r1, r4, r8
LD r6, 16(r3)
MUL r7, r1, r6
ST 24(r3), r7
```

Addressing Modes (IV)

5.7. Rewrite the program from Problem 5.6 to take advantage of postincrementing addressing mode. The value in r3 is allowed to be different at the end of the program from the value at the beginning. You may not use register plus immediate addressing mode anywhere in the program, but you may use register addressing mode.

Solution

The first ADD operation is still required to compute the address used by the first LD. The LD operations can then compute the address for the next memory operation. Remember that the increment value for each operation should be the difference between the address referenced by the next operation and the one referenced by the current operation, not the offset from the original value in the register containing the address.

```
ADD r3, r3, #8
LD r4, 8[r3]
ADD r1, r4, r8
LD r6,8[r3]
MUL r7, r1, r6
ST (r3), r7
```

Multimedia Vector Instructions (I)

5.8. If a program operates on 8-bit data types and a processor's multimedia vector instructions operate on 64-bit data words, what is the maximum-possible speedup that can be achieved by using multimedia vector instructions? Assume that all instructions take the same amount of time to execute.

Solution

Eight 8-bit values can fit in a 64-bit word. Therefore, the processor can perform eight 8-bit operations in parallel using multimedia vector instructions. If all instructions in the program were replaced with multimedia vector instructions, the program would require 1/8th the number of instructions as the original program, for a maximum speedup of 8.

Multimedia Vector Instructions (II)

5.9. If two registers contain the values 0xab0890c2 and 0x4598ee50, what is the result of adding them using

a. Multimedia vector operations that operate on 8-bit data?
b. Multimedia vector operations that operate on 16-bit data?

Assume that saturating arithmetic is not being used.

Solution

To find the result of using multimedia vector operations, we simply divide the input data words into chunks of the appropriate size and add them. This gives the following separate additions and final result (all numbers are in hexadecimal):

a. (ab + 45), (08 + 98), (90 + ee), (c2 + 50) → 0xf0a07e12

b. (ab08 + 4598), (90c2 + ee50) → 0xf0a07f12

Fixed-Length versus Variable-Length Encodings (I)

5.10. What are the pros and cons of fixed-length and variable-length instruction encodings?

Solution

Variable-length instruction encodings reduce the amount of memory that programs take up, since each instruction takes only as much space as it requires. Instructions in a fixed-length encoding scheme all take up as much storage space as the longest instruction in the ISA, meaning that there is some number of wasted bits in the encoding of instructions that take fewer operands, don't allow constant inputs, and so forth.

However, variable-length instruction sets require more-complex instruction decode logic than fixed-length instruction sets, and they make it harder to calculate the address of the next instruction in memory. Therefore, processors with fixed-length instruction sets can often be implemented at higher clock rates than processors with variable-length instruction sets.

Fixed-Length versus Variable-Length Encodings (II)

5.11. A given processor has 32 registers, uses 16-bit immediates, and has 142 instructions in its ISA. In a given program, 20 percent of the instructions take one input register and have one output register, 30 percent have two input registers and one output register, 25 percent have one input register, one output register, and take an immediate input as well, and the remaining 25 percent have one immediate input and one output register.

a. For each of the four types of instructions, how many bits are required? Assume that the ISA requires that all instructions be a multiple of 8 bits in length.

b. How much less memory does the program take up if a variable-length instruction set encoding is used as opposed to a fixed-length encoding?

Solution

a. With 142 instructions, 8 bits are required to determine which instruction an instruction is ($128 < 142 < 256$). 32 registers means that 5 bits are required to encode register ID, and we know that 16 bits are required for each immediate. Given that, it's just a matter of adding up the fields required for each type of instruction.

One register input, one register output: $8 + 5 + 5$ bits $= 18$ bits, which rounds up to 24.

Two input registers, one output register: $8 + 5 + 5 + 5 = 23$ bits, which rounds up to 24.

One input register, one output register, and an immediate: $8 + 5 + 5 + 16$ bits $= 34$ bits, which rounds up to 40 bits.

One input immediate, one output register: $8 + 16 + 5$ bits $= 29$ bits, which rounds up to 32 bits.

b. Since the largest instruction type requires 40-bit instructions, the fixed-length encoding will have 40 bits per instruction. Each instruction type in the variable encoding will use the number of bits given in Part a. To find the average number of bits per instruction in the variable-length encoding, we multiply the number of bits for each instruction type by that type's frequency and add the results. This gives (20 percent × 24 bits) + 30 percent × 24 bits) + (25 percent × 40 bits) + (25 percent × 32 bits) = 4.8 + 7.2+

$10 + 8 = 30$ bits on average. Therefore, the variable-length encoding requires 25 percent less space than the fixed-length encoding for this program.

Datapath Design

5.12. If it takes 5 ns to read an instruction from memory, 2 ns to decode the instruction, 3 ns to read the register file, 4 ns to perform the computation required by the instruction, and 2 ns to write the result into the register file, what is the maximum clock rate of the processor?

Solution

The time for an instruction to pass through the processor must be greater than the clock cycle time of the processor. The total time to execute an instruction is just the sum of the times to perform each step, or 16 ns. The maximum clock rate is 1/cycle time, or 62.5 MHz.

Microprogramming

5.13. a. Why was microprogramming used on many early processors?
b. Why have modern processors gone away from this technique?

Solution

a. Microprogramming allowed relatively complex instructions to be implemented using small amounts of hardware.

b. Microprogramming has become less commonly used because the increased amount of hardware available to computer architects and the less complex instruction sets used in current processors allow instructions to be directly implemented in hardware. This generally provides better performance than microprogramming, so architects choose to implement instructions directly rather than microprogramming them.

Register File Design

5.14. a. Why do many processors implement integer and floating-point register files as separate register files?
b. Give two ways in which using one register file for both integer and floating-point data might be better than separate register files.

Solution

a. There are two arguments for separate register files. First, on processors that execute multiple instructions in each cycle, having separate register files reduces the total area required for the register file because register files grow quadratically with the number of simultaneous accesses they support (ports). Second, having separate register files allows each of the register files to be located closer to the execution units than access it, reducing wire delay. Since integer and floating-point values are mostly independent—few values are operated on by both integer and floating-point instructions—separate register files are the choice for most processors.

b. One reason is that having a single register file for integer and floating-point data would let the number of registers used for integer and floating-point data vary depending on the needs of the program. With separate register files, the number of registers available for each type of data is fixed—if a program references a large number of integer values but few floating-point values, it cannot easily use the floating-point registers to store integer values that do not fit in the integer register file. If the processor had one combined register file, the program could use it to hold whatever combination of integer and floating-point data gave the best performance.

A second reason is that, while integer and floating-point values are mostly independent, there are some cases where both integer and floating-point instructions operate on a value. In this case, explicit operations are required to move the value between the two register files if the processor implements separate integer and floating-point register files. If the processor implemented a single combined register file, both types of instructions could access the value from the combined register file.

CHAPTER 6

Pipelining

6.1 Objectives

This chapter covers pipelining, a technique for improving processor performance. Pipelining allows a processor to overlap the execution of several instructions so that more instructions can be executed in the same period of time.

After completing this chapter, you should be able to

1. Describe pipelining and how it works
2. Compute the cycle time of a processor with different degrees of pipelining
3. Determine how long (in both processor cycles and time) it takes to execute small code segments on pipelined processors
4. Describe result forwarding and discuss how it affects execution time
5. Compute the execution time of small code segments on pipelined processors with result forwarding

6.2 Introduction

Early computers executed instructions in a very straightforward fashion: The processor fetched an instruction from memory, decoded it to determine what the instruction was, read the instruction's inputs from the register file, performed the computation required by the instruction, and wrote the result back into the register file. Instructions that accessed memory were slightly different, but each instruction was completely finished before execution of the next one began. The problem with this approach is that the hardware needed to perform each of these steps (instruction fetch, instruction decode, register read, instruction execution, and register write-back) is different, so most of the hardware is idle at any given moment, waiting for the other parts of the processor to complete their part of executing an instruction.

In many ways, this is similar to baking several loaves of bread by making the dough for one loaf, letting that loaf rise, baking the loaf, and then repeating the

entire process. While each of the steps in baking each loaf of bread has to be done in order and takes a set amount of time, one person could bake several loaves of bread much faster by making the dough for the second loaf while the dough for the first loaf is rising, making the dough for the third loaf while the second loaf was rising and the first loaf baking, and continuing this process with each loaf so that there were three loaves of bread in progress at any time. Each loaf of bread would take the same amount of time to make, but the number of loaves made in a given amount of time would increase.

- Overlapping instr
- h/w increased but exec. time decreased.

Pipelining is a technique for overlapping the execution of several instructions to reduce the execution time of a set of instructions. Similar to the baking analogy, each instruction takes the same amount of time to execute in a pipelined processor as it would in a nonpipelined processor (longer, actually, because pipelining adds hardware to the processor), but the rate at which instructions can be executed is increased by overlapping instruction execution.

When we discuss pipelining, and computer performance in general, two terms are often used: latency and throughput. *Latency* is the amount of time that a single operation takes to execute. *Throughput* is the rate at which operations get executed (generally expressed as operations/second or operations/cycle). In a nonpipelined processor, throughput = 1/latency, since each operation executes by itself. In a pipelined processor, throughput > 1/latency, since instruction execution is overlapped. The latency of a pipelined processor is still important, however, as it determines how often dependent instructions may be executed.

Latency & Throughput.

6.3 Pipelining

Pipeline Latches are placed b/w each section of pipeline

To implement pipelining, designers divide a processor's datapath into sections, and place *pipeline latches* between each section, as shown in Fig. 6-1. At the start of each cycle, the pipeline latches read their inputs and copy them to their outputs, which then remain constant throughout the rest of the cycle. This breaks the datapath into several sections, each of which has a latency of one clock cycle, since an instruction cannot pass through a pipeline latch until the start of the next cycle.

Stages of PL

The amount of the datapath that a signal travels through in one cycle is called a *stage* of the pipeline, and designers often describe a pipeline that takes n cycles as an n-stage pipeline. In Fig. 6-1, the pipeline has five stages. Stage 1 is the fetch instruction block and its associated pipeline latch, stage 2 is the decode instruction block and its pipeline latch, and stages 3, 4, and 5 are the subsequent blocks of the pipeline. Computer architects differ on whether a pipeline latch is the last part of a stage or the first part of the next stage, so an alternate division of the pipeline into stages would be to count the fetch instruction block as stage 1, the first pipeline latch and the decode instruction block as stage 2, and so on.

Figure 6-2 shows how instructions flow through the pipeline in Fig. 6-1. On cycle 1, the first instruction enters the instruction fetch (IF) stage of the pipeline and stops at the pipeline latch between the instruction fetch and instruction decode (ID) stages of the pipeline. On cycle 2, the second instruction enters the instruction fetch stage, while instruction 1 proceeds to the instruction decode stage. On the third cycle,

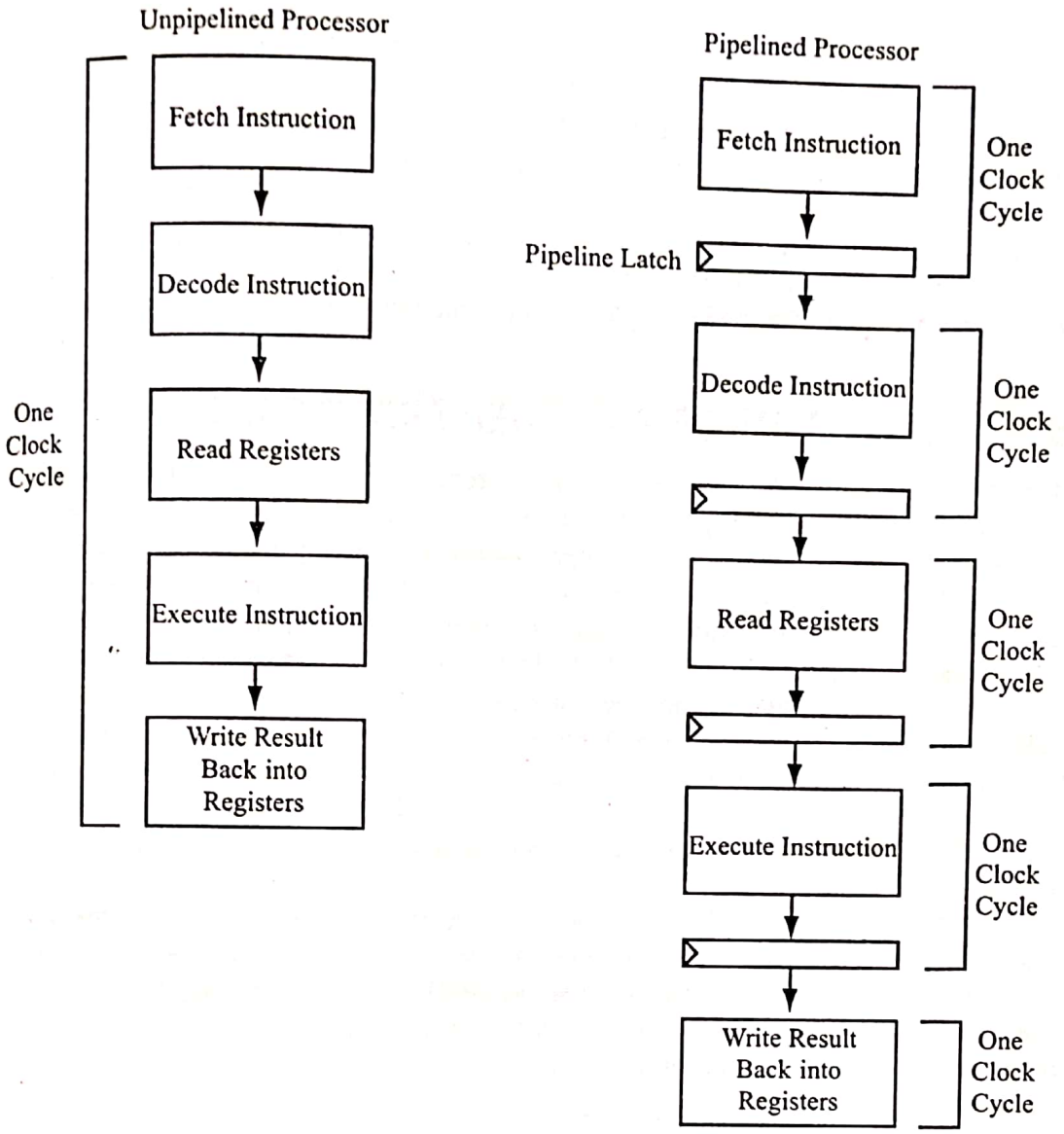


Fig. 6-1. Pipelined versus nonpipelined processor.

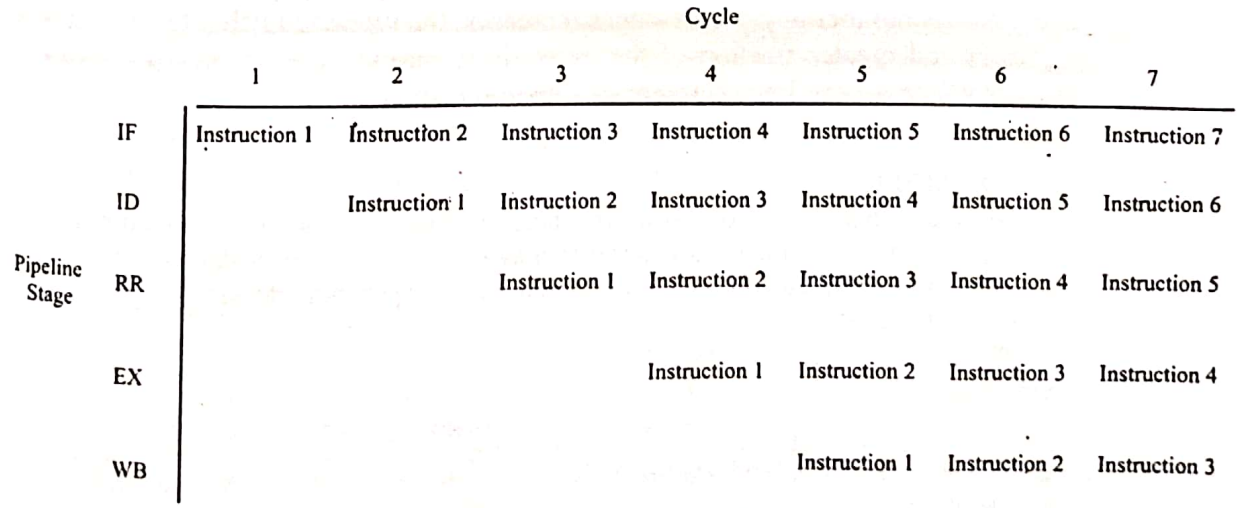


Fig. 6-2. Instruction flow in a pipelined processor.

instruction 1 enters the register read (RR) stage, instruction 2 is in the instruction decode stage, and instruction 3 enters the instruction fetch stage.

Instructions proceed through the pipeline at one stage per cycle until they reach the register write-back (WB) stage, at which point execution of the instruction is complete. Thus, on cycle 6 in the example, instructions 2 through 6 are in the pipeline, while instruction 1 has completed and is no longer in the pipeline. The pipelined processor is still executing instructions at a rate (throughput) of one instruction per cycle, but the latency of each instruction is now 5 cycles instead of 1.

6.3.1 CYCLE TIME OF PIPELINED PROCESSORS

If you consider just the number of cycles required to execute a given set of instructions, it looks like pipelining doesn't increase the performance of the processor. In fact, as we'll see later, pipelining a processor generally increases the number of clock cycles it takes to execute a program, because some instructions get held up in the pipeline waiting for the instructions that generate their inputs to execute. The performance benefit of pipelining comes from the fact that, because less of the logic in the datapath gets executed in a single cycle in a pipelined processor, pipelined processors can have reduced cycle times (more cycles/second) than unpipelined implementations of the same processor. Since the pipelined processor has a throughput of one instruction/cycle, the total number of instructions executed per unit time is higher in the pipelined processor, giving better performance.

The cycle time of a pipelined processor is dependent on four factors: the cycle time of the unpipelined diversion of the processor, the number of pipeline stages, how evenly the datapath logic is divided among the stages, and the latency of the pipeline latches. If the logic can be divided evenly among the pipeline stages, the clock period of the pipelined processor is[1]

$$\text{Cycle Time}_{\text{Pipelined}} = \frac{\text{Cycle Time}_{\text{Unpipelined}}}{\text{Number of Pipeline Stages}} + \text{Pipeline Latch Latency}$$

since each stage contains the same fraction of the original logic, plus one pipeline latch. As the number of pipeline stages increases, the pipeline latch latency becomes a greater and greater fraction of the cycle time, limiting the benefit of dividing a processor into a very large number of pipeline stages.

EXAMPLE

An unpipelined processor has a cycle time of 25 ns. What is the cycle time of a pipelined version of the processor with 5 evenly divided pipeline stages, if each pipeline latch has a latency of 1 ns? What if the processor is divided into 50 pipeline stages?

[1] A slightly higher clock rate can be achieved by taking advantage of the fact that an n-stage pipeline requires only $n - 1$ pipeline latches by assigning enough additional logic to the last stage of the pipeline to make its total latency equal to that of the other pipeline stages including their pipeline latches. See Exercise 6.5 for an example of this.

Solution

Applying the above equation, cycle time for the 5-stage pipeline = (25 ns/5)+ 1 ns = 6 ns. For the 50-stage pipeline, cycle time = (25 ns/50)+ 1 ns = 1.5 ns. In the 5-stage pipeline, the pipeline latch latency is only 1/6th of the overall cycle time, while the pipeline latch latency is 2/3 of the total cycle time in the 50-stage pipeline. Another way of looking at this is that the 50-stage pipeline has a cycle time 1/4 that of the 5-stage pipeline, at a cost of 10 times as many pipeline latches.

Often, the datapath logic cannot easily be divided into equal-latency pipeline stages. For example, accessing the register file in a processor might take 3 ns, while decoding an instruction might take 4 ns. When deciding how to divide a datapath into pipeline stages, designers must balance the desire to have each stage have the same latency with the difficulty of dividing the datapath into pipeline stages at different points and the amount of data that has to be stored in the pipeline latch, which determines the amount of space that the latch takes up on the chip. Some parts of the datapath, such as the instruction decode logic, are irregular, making it hard to split them into stages. Other parts generate a large number of intermediate data values which would have to be stored in the pipeline latch. For these sections, it is often more efficient to place the pipeline latch at a point where there are fewer intermediate results, and thus fewer bits that have to be stored in the latch, than to place the pipeline latch at a point that divides the datapath into more even sections. When a processor cannot be divided into equal-latency pipeline stages, the clock cycle time of the processor is equal to the latency of the longest pipeline stage plus the pipeline latch delay, since the cycle time has to be long enough for the longest pipeline stage to complete and store its result in the pipeline latch between it and the next stage.

Division of data Path Logic

CC time is expressed as Latency + Latch D.

EXAMPLE

Suppose an unpipelined processor with a 25-ns cycle time is divided into 5 pipeline stages with latencies of 5, 7, 3, 6, and 4 ns. If the pipeline latch latency is 1 ns, what is the cycle time of the resulting processor?

Solution

The longest pipeline stage is 7 ns. Adding a 1-ns pipeline latch to this stage gives a total latency of 8 ns, which is the cycle time.

6.3.2 PIPELINE LATENCY

While pipelining can reduce a processor's cycle time and thereby increase instruction throughput, it increases the latency of the processor by at least the sum of all the pipeline latch latencies. The latency of a pipeline is the amount of time that a single instruction takes to pass through the pipeline, which is the product of the number of pipeline stages and the clock cycle time.

Latency of PL = PL stages X CC Time.

EXAMPLE
If an unpipelined processor with a cycle time of 25 ns is evenly divided into 5 pipeline stages using pipeline latches with 1-ns latency, what is the total latency of the pipeline? How about if the processor is divided into 50 pipeline stages?

Solution

This is the same pipeline as the first example in Section 6.3.1, in which we determined that the cycle time of the 5-stage pipeline was 6 ns and the cycle time of the 50-stage pipeline was 1.5 ns. Given that, we can compute the latency of each pipeline by multiplying the cycle time by the number of stages in the pipeline. This gives a latency of 30 ns for the 5-stage pipeline and 75 ns for the 50-stage pipeline.

This example shows the impact pipelining can have on latency, particularly as the number of stages grows. The 5-stage pipeline has a latency of 30 ns, 20 percent longer than the original 25-ns unpipelined processor, while the 50-stage pipeline has a latency of 75 ns, three times that of the original processor!

Pipelines with uneven pipeline stages use the same formula, although they see an even greater increase in latency, because the cycle time must be long enough to accommodate the longest stage of the pipeline, even if the other stages are much shorter.

EXAMPLE
Suppose an unpipelined processor with a 25-ns cycle time is divided into 5 pipeline stages with latencies of 5, 7, 3, 6, and 4 ns. If the pipeline latch latency is 1 ns, what is the latency of the resulting pipeline?

Solution

This is the same pipeline as the second example from Section 6.3.1 and has a cycle time of 8 ns. Since there are 5 stages in the pipeline, the total latency of the pipeline is 40 ns.

6.4 Instruction Hazards and Their Impact on Throughput

As described above, pipelining increases processor performance by increasing instruction throughput. Because several instructions are overlapped in the pipeline, cycle time can be reduced, increasing the rate at which instructions execute. In the ideal case, the throughput of a pipeline is simply 1/cycle time, so a 5-stage pipeline with a 6-ns cycle time and an unpipelined cycle time of 25 ns would have an ideal

throughput of $\frac{1}{6\text{ ns}} = 1.67 \times 10^8$ instructions/s, a more than 4× improvement over the unpipelined processor's throughput of 4×10^7 instructions/s.

However, there are a number of factors that limit a pipeline's ability to execute instructions at its peak rate, including dependencies between instructions, branches, and the time required to access memory. In this chapter, we will discuss how instruction dependencies and branches affect the execution time of programs on pipelined processors. Later chapters will cover techniques for improving memory system performance.

Instruction *hazards* (*dependencies*) occur when instructions read or write registers that are used by other instructions. They are divided into four categories, depending on whether the two instructions involved read or write each other's registers. Read-after-read (RAR) hazards, as shown in Fig. 6-3, occur when two instructions both read from the same register. RAR hazards don't cause a problem for the processor because reading a register doesn't change the register's value. Therefore, two instructions that have an RAR hazard can execute on successive cycles (or on the same cycle, in processors that can execute more than one instruction/cycle).

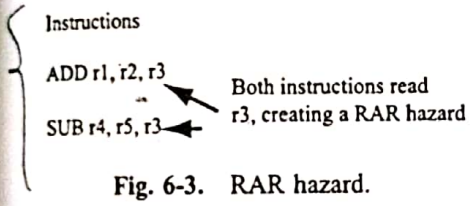


Fig. 6-3. RAR hazard.

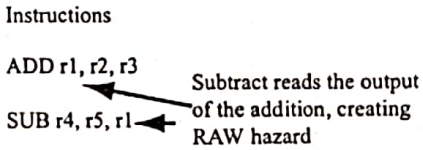


Fig. 6-4. RAW hazard.

Read-after-write (RAW) hazards occur when an instruction reads a register that was written by a previous instruction, as shown in Fig. 6-4. RAW hazards are also known as *data dependencies* or *true dependencies*, because they occur when an instruction needs to use the result of another instruction.

When a RAW hazard occurs, the reading instruction cannot proceed past the register read stage of the pipeline until the writing instruction has passed through the write-back stage, because the data that the reading instruction needs is not available until then. This is called a *pipeline stall* or *bubble*. Note that the reading instruction can proceed through the instruction fetch and instruction decode stages of the pipeline before the writing instruction completes, because the reading instruction does not need the value produced by the writing instruction until it reaches the register read stage.

Figure 6-5 shows how the instructions in Fig. 6-4 would flow through the five-stage example pipeline we've been using. During cycles 1 through 4, both instructions flow through the pipeline at one stage per cycle. In cycle 4, the subtract instruction attempts to read r5 and r1, its input registers, and determines that r1 cannot be read because the ADD has not written its result into r1 yet. (The hardware the subtract uses to determine this will be discussed later in this chapter.)

In cycle 5, the subtract would normally enter the execute stage, but it is prevented from doing so because it was not able to read r1 on cycle 4. Instead, the hardware inserts a special no-operation (NOP) instruction, known as a bubble, into the

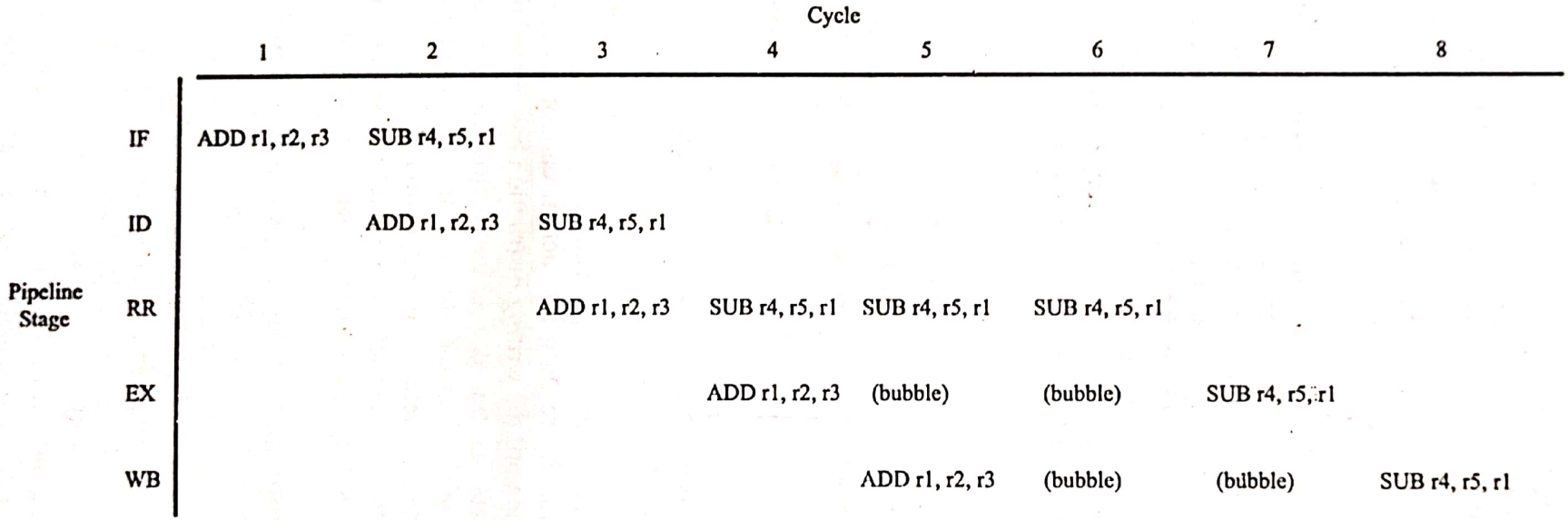


Fig. 6-5. Pipelined execution with stall.

execute stage of the pipeline, and the subtract tries to read its input registers again on cycle 5. The result of the ADD is still unavailable on cycle 5 because the ADD has not completed the writeback stage yet, so the subtract cannot enter the execute stage on cycle 6. On cycle 6, the SUB is able to read r1, and it proceeds into the execute stage on cycle 7. Thus, the RAW dependency between these two instructions has caused a two-cycle delay in the pipeline.

Write-after-read (WAR) hazards, shown in Fig. 6-6, and write-after-write (WAW) hazards (Fig. 6-7) occur when the output register of an instruction has been either read or written by a previous instruction. These hazards are sometimes called *name dependencies*, as they occur because the processor has a finite number of registers. If the processor had an infinite number of registers, it could use a different register for the output of each instruction, and WAW and WAR hazards would never occur.

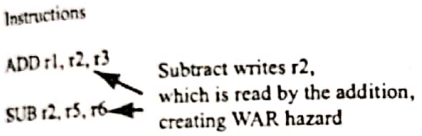


Fig. 6-6. WAR hazard.

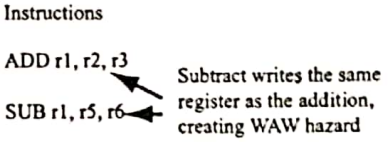


Fig. 6-7. WAW hazard.

If a processor executes instructions in the order that they appear in the program and uses the same pipeline for all instructions, WAR and WAW hazards do not cause delays because of the way instructions flow through the pipeline. Since the output register of an instruction is written in the writeback stage of the pipeline, instructions with WAW hazards will enter the writeback stage in the order in which they appear in the program and write their results into the register in the right order. There is even less of a problem with instructions that have WAR hazards, because the register read stage of the pipeline occurs before the writeback stage. By the time an instruction enters the writeback stage of the pipeline, all previous instructions in the program have already passed through the register read stage and read their input values. Therefore, the writing instruction can go ahead and write its destination register without causing any problems.

If a processor's instructions do not all have the same latency, WAW and WAR hazards can cause problems, because it is possible for a low-latency instruction to complete before a longer-latency instruction that appeared earlier in the program. These processors must keep track of name dependencies between instructions and stall the pipeline as necessary to resolve these hazards. WAR and WAW hazards are also an issue in out-of-order processors, which allow instructions to execute in different orders than the original program to improve performance. These processors will be discussed in more detail in the next chapter, along with register renaming, a hardware technique to reduce the performance impact of name dependencies.

6.4.1 BRANCHES

Branch instructions can also cause delays in pipelined processors, because the processor cannot determine which instruction to fetch next until the branch has

executed. Effectively, branch instructions, particularly conditional branches, create data dependencies between the branch instruction and the instruction fetch stage of the pipeline, since the branch instruction computes the address of the next instruction that the instruction fetch stage should fetch. Figure 6-8 shows how a branch instruction would execute on our five-stage pipeline. The PC is updated at the end of the cycle that the branch instruction is in the execute stage, allowing the next instruction to be fetched on the following cycle.

Processor's Branch Delay
Control Hazards

The delay between when a branch instruction enters the pipeline and the time at which the next instruction enters the pipeline is often called the processor's *branch delay*. Branch delays are sometimes called *control hazards*, because the delay is due to the control flow of the program. The pipeline illustrated in Fig. 6-8 has a four-cycle branch delay.

Branch delays have a significant impact on the performance of modern processors, and a number of techniques have been developed to address them. One technique is to add hardware to allow the result of a branch instruction to be computed earlier in the pipeline. For example, if our pipeline computed the new value of the PC in the register read stage instead of the execute stage, the branch delay would be reduced to three cycles. Another technique is to add hardware that predicts the destination address of each branch before the branch completes, allowing the processor to begin fetching instructions from that address earlier in the pipeline. These *branch prediction* techniques are beyond the scope of this book, but they significantly improve the performance of modern processors.

6.4.2 STRUCTURAL HAZARDS

Structural Hazard

A final cause of stalls in pipelined processors are *structural hazards*. Structural hazards occur when the processor's hardware is not capable of executing all the instructions in the pipeline simultaneously. For example, if the register file did not have enough ports to allow an instruction in the WB stage to write its result into the register file in the same cycle that another instruction in the RR stage read from the register file, it would be necessary to stall any instruction in the RR stage if there was also an instruction in the WB stage on that cycle. (Choosing to stall the instruction in the WB stage to allow the instruction in the RR stage to proceed would be a poor choice, as the stall in the WB stage would prevent instructions in the EX stage from advancing.)

Structural hazards within a single pipeline are relatively rare on modern processors, because their hardware and instruction sets have been designed to support pipelining. However, processors that execute more than one instruction in a cycle, which are covered in the next chapter, often have restrictions on the types of instructions that the hardware can execute simultaneously. For example, a processor might be able to execute two instructions in each cycle, but only if one of the instructions was an integer operation and the other a floating-point computation.

6.4.3 SCOREBOARDING

Pipelined processors need to keep track of which registers will be written by instructions that are already in the pipeline, so that subsequent instructions can

Fig. 6-8. Branch instruction in pipeline.

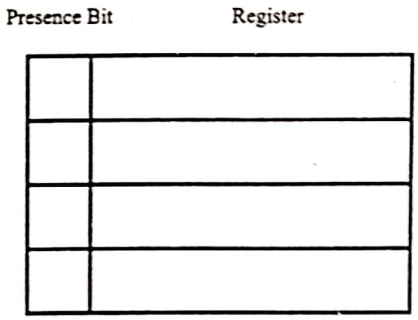


Fig. 6-9. Register scoreboard.

determine whether their input registers are available when they reach the register read stage. To do this, most processors use a technique called *register scoreboarding*. In scoreboarding, a bit, known as the *presence* bit, is added to each register in the register file, as shown in Fig. 6-9. The presence bit records whether the register is available for reading (full) or waiting for an instruction to write its output value (empty).

When an instruction enters the register read stage, the hardware checks to see if all of its input registers are full. If so, the hardware reads the values of all the input registers, marks the output register of the instruction empty, and allows the instruction to proceed to the execute stage on the next cycle. If not, the hardware holds the instruction in the register read stage until its input values become full, inserting bubbles into the execute stage on each cycle until this happens. When an instruction reaches the writeback stage and writes its result into its destination register, that register is marked full, allowing operations that read the register to proceed.

6.5 Predicting Execution Time in Pipelined Processors

An unpipelined processor must complete the execution of each instruction before it begins the execution of the next. This means that dependencies between instructions generally do not affect the execution time of a program on an unpipelined processor, because the result of each instruction has been fully computed before any later instruction in the program begins execution. Thus, on an unpipelined processor, the execution time of a program can be computed by simply adding together the execution times of all of the instructions that the processor executes in running the program.

On a pipelined processor, computing the execution time of a program is more complicated because dependencies between instructions affect a program's execution time. In the ideal case, no pipeline stalls occur, and one instruction passes from the register read stage to the execute stage in each cycle. In this case, the execution time (in cycles) of a program is equal to the depth of the pipeline plus the number of instructions in the program minus 1, because the first instruction passes through the

pipeline in a number of cycles equal to the pipeline depth, and the other instructions proceed through at one instruction per cycle. Multiplying the execution time in cycles by the clock cycle time gives the execution time of the program in seconds. When programs contain dependencies that cause pipeline stalls, however, we need to be able to determine how many stalls occur to predict the execution time of a program.

One way to compute the execution time of a program on a pipelined processor is to draw a pipeline diagram, similar to Fig. 6-5, for the program, but this becomes impractical for large programs. Instead, a better approach is to separate the execution time of the program into two parts: the pipeline latency and the time required to issue all of the instructions in the program. An instruction is said to have issued when it passes from the register read stage into the execute stage, because the register read stage is generally the last point in the pipeline where an instruction can stall. Once an instruction enters the execute stage, it is guaranteed to proceed through the pipeline at one stage per cycle until it reaches the last stage and completes. The time to issue all of the instructions begins in the cycle when the first instruction issues, ends in the cycle when the last instruction issues, and includes all of the cycles during which pipeline stalls cause bubbles to be issued into the execute stage.

Using this model, the execution time (in cycles) of a program is equal to the pipeline latency plus the time to issue all of the instructions minus 1 (again, because the first instruction travels through the pipeline in a number of cycles equal to the pipeline depth). In the ideal case, the number of cycles required to issue all of the instructions in a program is equal to the number of instructions in the program, because one instruction issues each cycle. In almost all cases, however, dependencies cause stalls during the execution of a program, which increase the number of cycles required to issue the instructions in the program.

To help compute the time required to issue the instructions in a program, architects define the *instruction latency* of each instruction type in a pipeline as the delay between the time at which an instruction of that type issues and the time at which a dependent instruction may issue. For example, the pipeline in Fig. 6-5 has an instruction latency of three cycles for non-branch instructions, because a dependent instruction may enter the execute stage three cycles after the instruction which generates its data. Branch instructions have instruction latencies of four cycles, as shown in Fig. 6-8. (Think of the instruction that executes after a branch as being dependent on the result of the branch.)

In a pipelined processor, each instruction will issue on either the cycle after the instruction before it in the program issues or the cycle after the latencies of all of the instructions it is dependent on complete, whichever is later. Thus, the number of cycles required to issue all of the instructions in a program can be computed by proceeding sequentially through the program to determine when each instruction can issue.

EXAMPLE

On the sample pipeline we've been using, which has instruction latencies of 3 cycles for non-branch instructions and 4 cycles for branch instructions, what is the execution time of the following instruction sequence?

execution time (in cycles) X CC time = Execution time of program in sec.

issue time = instead - inst issue + cycles included of stall.

exec time (in cycles) = Pipeline Latency + time to issue all inst - 1

inst. Latency

```
ADD r1, r2, r3
SUB r4, r5, r6
MUL r8, r2, r1
ASH r5, r2, r1
OR r10, r11, r4
```

Solution

Our sample pipeline has five stages, so the pipeline latency is 5 cycles. To compute the issue time of the instruction sequence, assume that the ADD issues on cycle n. The SUB is independent of the ADD, so it can issue on cycle $n + 1$, the cycle after the previous instruction in the program issues. The MUL depends on the ADD, so it can't issue until cycle $n + 3$, because the ADD has a latency of 3 cycles. The ASH is also dependent on the ADD, but it can't issue until cycle $n + 4$, because the MUL issues on cycle $n + 3$. The OR is independent of the previous instructions, so it issues on cycle $n + 5$. Therefore, it takes 6 cycles to issue all of the instructions in the program. Using the formula, the execution time of the program is 5 cycles (pipeline latency) + 6 cycles (time to issue the instructions in the program) $- 1 = 10$ cycles. Figure 6-10 shows a pipeline diagram of the execution of this set of instructions, confirming the 10-cycle execution time.

This example also illustrates an important factor in achieving good performance on pipelined processors—scheduling instructions to avoid pipeline stalls. Because the SUB instruction did not use the result of the ADD, it was able to execute on the cycle immediately following the ADD. If the SUB and MUL instructions had been reversed, the MUL instruction would still have had to wait until three cycles after the ADD executed for its input data to be ready, and the SUB would have been unable to issue until four cycles after the ADD. Compilers for pipelined processors must understand the details of the pipeline to be able to place instructions in an order which maximizes performance.

EXAMPLE

What is the execution time of this sequence on a 7-stage pipeline with a 2-cycle instruction latency for non-branch instructions, but a 5-cycle branch instruction latency? Assume the branch is not taken, so the DIV is the next instruction executed after it.

```
BNE r4, #0, r5
DIV r2, r1, r7
ADD r8, r9, r10
SUB r5, r2, r9
MUL r10, r5, r8
```

Solution

The pipeline has 7 stages, so the pipeline latency is 7 cycles. To compute the number of cycles required to issue the program, assume the BNE executes on cycle n. The pipeline has a 5-cycle branch latency, so the DIV executes on

Pipeline Stage	Cycle 1	2	3	4	5	6	7	8	9	10
IF	ADD r1, r2, r3	SUB r4, r5, r6	MUL r8, r2, r1	ASH r5, r2, r1	OR r10, r11, r4					
ID		ADD r1, r2, r3	SUB r4, r5, r6	MUL r8, r2, r1	ASH r5, r2, r1	OR r10, r11, r4				
RR			ADD r1, r2, r3	SUB r4, r5, r6	MUL r8, r2, r1	MUL r8, r2, r1	ASH r5, r2, r1	OR r10, r11, r4		
EX				ADD r1, r2, r3	SUB r4, r5, r6	(bubble)	MUL r8, r2, r1	ASH r5, r2, r1	OR r10, r11, r4	
WB					ADD r1, r2, r3	SUB r4, r5, r6	(bubble)	MUL r8, r2, r1	ASH r5, r2, r1	OR r10, r11, r4

Fig. 6-10. Pipelined execution example.

cycle $n + 5$. The ADD has no data dependency on the DIV, so it executes on cycle $n + 6$. The SUB has a data dependency on the DIV, so it can't execute before cycle $n + 7$, which is the first cycle on which it is possible for the SUB to execute, because the ADD issued on cycle $n + 6$.

The MUL has data dependencies on both the SUB and the ADD. The ADD issued on cycle $n + 6$, so an instruction that depended only on the ADD could issue on cycle $n + 8$. However, the SUB issued on cycle $n + 7$, so instructions that depend on it can't issue until cycle $n + 9$. Therefore, the MUL issues on cycle $n + 9$, and it takes a total of 10 cycles to issue all of the instructions in the program.

Therefore, the total execution time of this program is 7 cycles (pipeline latency) + 10 cycles (time to issue) − 1 = 16 cycles.

6.6 Result Forwarding (Bypassing)

As illustrated in Fig. 6-11, much of the delay caused by data dependencies is due to the amount of time required to write the result of an instruction into the register file and then read it out as the input to another instruction. The result of the ADD instruction is computed in the execute stage of the pipeline in cycle 4, but the SUB instruction is unable to issue until cycle 7 because the result of the ADD is not written into the register file until cycle 5, allowing the subtract to read it on cycle 6. If the result of the ADD could be sent to the subtract instruction directly, without going through the register file, the subtract could issue on cycle 5 without any pipeline stalls.

Virtually all pipelined processors incorporate a technique known as result forwarding, or *bypassing*, that forwards the results of the execute stage(s) directly to instructions in the previous stages of the pipeline, allowing these instructions to proceed without waiting for the result to be written into the register file. In general, the instruction latency of non-branch instructions in a pipeline with bypassing is equal to the number of execute stages in the pipeline, because the output of an instruction is not computed until it completes the last execute stage, but an instruction's inputs are required when it enters the first execute stage. Bypassing does not usually improve the latency of branch operations, because the results of branch instructions are not written into the register file.

Figure 6-12 shows how bypassing would be implemented on our five-stage pipeline. In addition to the conventional writeback path, connections are added that send the output of the execute stage directly to the input of the execute stage and to the register read stage. If the instruction in the register read stage depends on the output of the instruction in the execute stage, it obtains its input from the bypass path at the start of the next cycle, as it enters the execute stage. Similarly, an instruction in the instruction decode stage that depends on the instruction in the execute stage obtains its input from the bypass path on the next cycle, while it is in register read stage, instead of having to wait until its input can be read out of the register file. There is no need to forward the result of an instruction to the instruction

Fig. 6-11. Data delay.

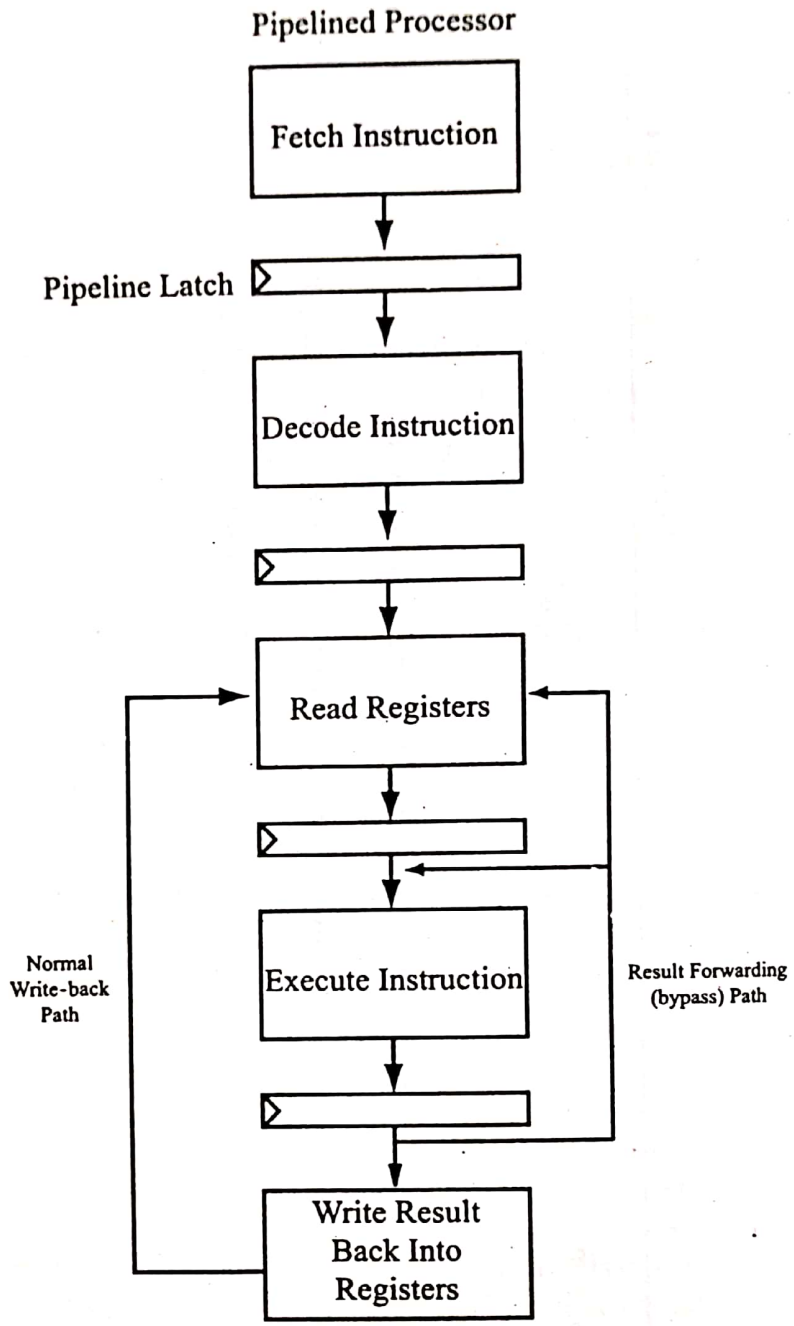


Fig. 6-12. Result forwarding (bypassing).

decode stage, because the result of an instruction in the execute stage will have been written into the register file by the time the instruction in the instruction fetch stage reaches the register read stage.

Different processors implement bypassing in different ways, although the basic idea remains the same. For example, some processors eliminate the need for the bypass path that connects the output of the execute stage to the register read stage by using register files that are written in the first half of a clock cycle and read in the second. If such a register file is used, an instruction in the write-back stage writes

its result during the first half of the clock cycle, allowing an instruction in the register read stage to read it during the second half of the clock cycle. This optimization alone would reduce the non-branch instruction latency of our sample pipeline to two cycles, and adding the bypass path from the output of the execute stage to the input of the execute stage would then reduce the non-branch instruction latency to one cycle.

EXAMPLE

What is the execution time (in cycles) of the code fragment from Fig. 6-5 if result forwarding is added to our 5-stage pipeline?

Solution

Result forwarding reduces the latency of non-branch instructions to one cycle, becuase our 5-stage pipeline has only one execute stage. This reduces the issue time of the code fragment to two cycles. The total execution time becomes 5 cycles (pipeline latency) + 2 cycles (issue time) − 1 = 6 cycles.

6.7 Summary

Pipelining improves processor performance by overlapping the execution of multiple instructions. While one instruction is being executed, the next instruction is reading its input registers, another instruction is being decoded, and so on. Since each stage of instruction execution requires different hardware, pipelining can greatly improve performance at a relatively low hardware cost.

The peak performance of a pipelined system is determined by how many pipeline stages the system contains, how even the division of execution into pipeline stages is, and how much delay is added for the pipeline registers in each pipeline stage. If the division of the processor into pipeline stages is uneven, the clock rate is limited by the latency of the longest stage. Even if the processor can be divided into even pipeline stages, the impact of pipelining diminishes as the number of pipeline stages increases, because the delay added by the pipeline latches becomes a significant portion of the cycle time.

The actual performance of a pipelined system is generally limited by data dependencies within a program. We have discussed three types of data dependencies: read-after-write, write-after-read, and write-after-write. WAR and WAW dependencies are also known as name dependencies, as they only occur because the processor has a limited number of registers to store results in, which must be reused over the course of a program's execution. Branches also limit a pipeline's performance, because the processor must stall until the branch has completed execution.

Result forwarding, or bypassing, is used to reduce the delay caused by RAW dependencies. In addition to writing the result of an instruction back into the register

file, bypassing sends the result of an instruction directly to instructions in the pipeline that need it, reducing the latency of non-branch instructions.

In the next chapter, we will discuss instruction-level parallelism, which further improves processor performance by allowing independent instructions to execute simultaneously. Pipelining and instruction-level parallelism combine well, and most modern processors employ both techniques to improve performance.

Solved Problems

Pipelining (I)

6.1. Why does pipelining improve performance?

Solution

In an unpipelined processor, each instruction is executed completely before execution of the next instruction begins. In a pipelined processor, instruction execution is divided into stages, and execution of the next instruction starts as soon as the current instruction has completed the first stage. This increases the rate at which instructions can be executed, improving performance.

Another way to describe this is that pipelining divides the processor's datapath into stages that are separated by pipeline latches. In an unpipelined processor, an instruction must be able to get all the way through the datapath within a single clock cycle. In a pipelined processor, an instruction must only be able to get through one stage of the pipeline in each cycle, allowing the clock cycle to be much shorter than in an unpipelined processor. Since a pipelined processor can still start executing one instruction during each clock cycle, shortening the clock cycle increases the rate at which instructions can be executed, improving performance.

Pipelining (II)

6.2. What are the limits on how much a processor's performance can be improved using pipelining?

Solution

There are two main limitations. First, as the number of pipeline stages increases, the fraction of the latency of each stage that is due to the pipeline latch increases. In the extreme, pipelining can't reduce the clock cycle time down below the latency of the pipeline latch required for each stage.

The second limitation comes from data dependencies and branch delays. Instructions that depend on the results of other instructions have to wait for those instructions to complete, creating pipeline stalls (bubbles), and instructions that follow branches have to wait for the branch to complete so that the processor knows which instruction to execute next. This means that the pipeline will execute less than one instruction per cycle on average. As the pipeline gets deeper, the delay between dependent instructions will get longer, meaning that more and more of the processor's time will be spent waiting for pipeline stalls. Bypassing can reduce this problem somewhat, but it cannot eliminate it completely, since very deep pipelines will have to have more than one execute stage, meaning that the delay between the execution of dependent instructions will be more than one cycle.

Even Pipelining

6.3. Given an unpipelined processor with a 10 ns cycle time and pipeline latches with 0.5 ns latency, what are the cycle times of pipelined versions of the processor with 2, 4, 8, and 16 stages if the datapath logic is evenly divided among the pipeline stages? Also, what is the latency of each of the pipelined versions of the processor?

Solution

$$\text{Cycle Time}_{\text{Pipelined}} = \frac{\text{Cycle Time}_{\text{Unpipelined}}}{\text{Number of Pipeline Stages}} + \text{Pipeline Latch Latency (from Sect. 6.3.1)}$$

Applying this formula, we get cycle times of 5.5, 3, 1.75, and 1.125 ns, showing the diminishing returns of pipelining as the pipeline latch latency becomes a significant part of the overall cycle time.

To compute the latency of each processor, we simply multiply the cycle time by the number of pipeline stages, giving latencies of 11, 12, 14, and 18 ns.

Pipeline to Achieve Clock Rate

6.4. For the processor from the last exercise, how many stages of pipelining are required to achieve a cycle time of 2 ns? 1 ns?

Solution

Here, we want to solve for the number of pipeline stages, so we rewrite the cycle time formula to get the following:

$$\text{Number of Stages} = \frac{\text{Cycle Time}_{\text{Unpipelined}}}{\text{Cycle Time}_{\text{Pipelined}} - \text{Pipeline Latch Latency}}$$

Applying this gives 6.67 as the number of pipeline stages required to achieve a 2 ns cycle time. Since you can't have a fractional number of pipeline stages, this rounds up to 7. The formula gives 20 as the number of pipeline stages required to achieve a 1 ns clock rate.

Minimum Cycle Time

6.5. For the processor and pipeline latch latencies in Problem 6.3, what is the minimum cycle time achievable with a 4-stage pipeline if additional logic is assigned to the final stage to balance the additional latency of the pipeline latches in the other stages?

Solution

In this pipeline, the total latency of each stage will be the same, even though some stages contain pipeline latches and others do not. A simple way to compute the cycle time in this case is to find the total latency of the original datapath plus the pipeline latches, and then divide by the number of stages. The latency of the original datapath is 10 ns. A 4-stage pipeline requires 3 pipeline latches. Each pipeline latch has a latency of 0.5 ns, so the pipeline latches add 1.5 ns to the datapath latency, giving a total latency of 11.5 ns. Dividing this by the number of stages (4) gives 2.875 ns as the clock time of the pipeline processor with this design.

Even Pipelining

6.3. Given an unpipelined processor with a 10 ns cycle time and pipeline latches with 0.5 ns latency, what are the cycle times of pipelined versions of the processor with 2, 4, 8, and 16 stages if the datapath logic is evenly divided among the pipeline stages? Also, what is the latency of each of the pipelined versions of the processor?

Solution

$$\text{Cycle Time}_{\text{Pipelined}} = \frac{\text{Cycle Time}_{\text{Unpipelined}}}{\text{Number of Pipeline Stages}} + \text{Pipeline Latch Latency (from Sect. 6.3.1)}$$

Applying this formula, we get cycle times of 5.5, 3, 1.75, and 1.125 ns, showing the diminishing returns of pipelining as the pipeline latch latency becomes a significant part of the overall cycle time.

To compute the latency of each processor, we simply multiply the cycle time by the number of pipeline stages, giving latencies of 11, 12, 14, and 18 ns.

Pipeline to Achieve Clock Rate

6.4. For the processor from the last exercise, how many stages of pipelining are required to achieve a cycle time of 2 ns? 1 ns?

Solution

Here, we want to solve for the number of pipeline stages, so we rewrite the cycle time formula to get the following:

$$\text{Number of Stages} = \frac{\text{Cycle Time}_{\text{Unpipelined}}}{\text{Cycle Time}_{\text{Pipelined}} - \text{Pipeline Latch Latency}}$$

Applying this gives 6.67 as the number of pipeline stages required to achieve a 2 ns cycle time. Since you can't have a fractional number of pipeline stages, this rounds up to 7. The formula gives 20 as the number of pipeline stages required to achieve a 1 ns clock rate.

Minimum Cycle Time

6.5. For the processor and pipeline latch latencies in Problem 6.3, what is the minimum cycle time achievable with a 4-stage pipeline if additional logic is assigned to the final stage to balance the additional latency of the pipeline latches in the other stages?

Solution

In this pipeline, the total latency of each stage will be the same, even though some stages contain pipeline latches and others do not. A simple way to compute the cycle time in this case is to find the total latency of the original datapath plus the pipeline latches, and then divide by the number of stages. The latency of the original datapath is 10 ns. A 4-stage pipeline requires 3 pipeline latches. Each pipeline latch has a latency of 0.5 ns, so the pipeline latches add 1.5 ns to the datapath latency, giving a total latency of 11.5 ns. Dividing this by the number of stages (4) gives 2.875 ns as the clock time of the pipeline processor with this design.

Uneven Pipelining

6.6. Suppose that an unpipelined processor has a cycle time of 25 ns, and that its datapath is made up of modules with latencies of 2, 3, 4, 7, 3, 2, and 4 ns (in that order). In pipelining this processor, it is not possible to rearrange the order of the modules (for example, putting the register read stage before the instruction decode stage) or to divide a module into multiple pipeline stages (for complexity reasons). Given pipeline latches with 1 ns latency:

a. What is the minimum cycle time that can be achieved by pipelining this processor?
b. If the processor is divided into the fewest number of pipeline stages that allow it to achieve the minimum latency from part 1, what is the latency of the pipeline?
c. If you are limited to a 2-stage pipeline what is the minimum cycle time?
d. What is the latency for the pipelines from part 3?

Solution

a. If there is no limit on the number of pipeline stages, then the minimum cycle time is determined by the latency of the longest module in the datapath plus the pipeline latch time. This gives a cycle time of 7 ns + 1 ns = 8 ns.
b. To answer this, we need to know how many pipeline stages the processor requires to operate at a cycle time of 8 ns. We can group any set of adjacent modules with total latencies of 7 ns or less into a single stage. Doing this gives 5 pipeline stages. 5 stages × 8 ns cycle time = 40 ns latency.
c. For minimum cycle time, we want to divide the modules into stages with as even latencies as possible. For two stages, this gives stage latencies of 16 ns and 9 ns (or the reverse order). Since we only need one pipeline latch between the two stages, we can divide the logic into a 9 ns first stage and a 16 ns second stage. Adding the pipeline latch to the first stage gives us a 16 ns clock rate.
d. 16 ns × 2 stages = 32 ns latency.

Instruction Hazards (I)

6.7. **a.** Identify all of the RAW hazards in this instruction sequence:

```
DIV r2, r5, r8
SUB r9, r2, r7
ASH r5, r14, r6
MUL r11, r9, r5
BEQ r10, #0, r12
OR r8, r15, r2
```

b. Identify all of the WAR hazards in the previous instruction sequence.
c. Identify all of the WAW hazards in the instruction sequence.
d. Identify all of the control hazards in this instruction sequence.

Solution

a. RAW hazards exist between the DIV instruction and the SUB instruction, between the ASH and the MUL, between the SUB and MUL, and between the DIV and the OR.
b. WAR hazards exist between the DIV and the ASH instructions, and between the DIV and the OR instructions.
c. There are no WAW hazards in this instruction sequence.
d. There is only one control hazard in this sequence, between the BEQ instruction and the OR instruction.

Instruction Hazards (II)

6.8. When reordering instructions to improve performance, which types of instruction hazards represent ordering constraints that must be maintained if the reordered program is to generate the same result as the original program, and why?

Solution

Control, RAW, WAW, and WAR hazards represent ordering constraints. RAW hazards indicate that the reading instruction uses the result of the writing instruction, and moving the reading instruction before the writing instruction will cause the reading instruction to see the wrong value of the writing instruction's output register. WAW hazards occur when multiple instructions write the same register. Changing the order of two instructions with a WAW hazard will cause a different instruction to write the output register last, leaving a different value in the register for any subsequent readers. WAR hazards occur when a register is reused. Moving the writing instruction before the reading instruction will cause the reading instruction to see the new value of the output register, rather than the old value, which it was intended to see.

Branch (control) hazards result from branch instructions that compute the address of the next instruction fetch, and create ordering constraints. Moving an instruction that was above a branch below the branch causes the instruction to only be executed if the branch is not taken, while moving an instruction that was below a branch above the branch has the reverse effect, causing an instruction that was only intended to be executed when the branch is not taken to be executed each time the instruction sequence executes.

RAR hazards do not represent ordering constraints. Reading a register does not change its value, so multiple reads may be done in any order. Structural hazards arise because of the limitations of the processor, not because of dependencies between instructions, so they do not generally impose constraints on the ordering of instructions in a program.

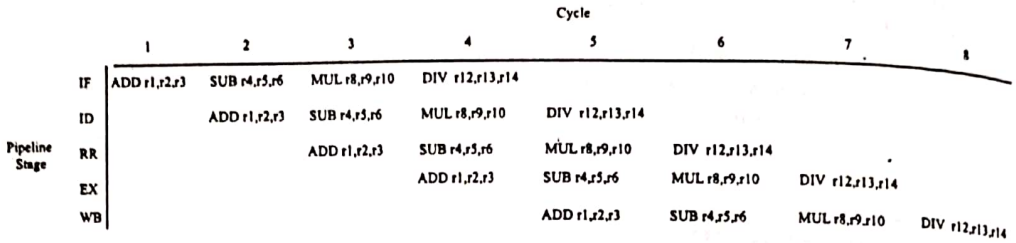
Pipelined Execution

6.9. Assuming no result forwarding and the five-stage sample pipeline of Section 6.3, draw a pipeline execution diagram similar to Fig. 6-2 for the following code fragment:

```
ADD r1, r2, r3
SUB r4, r5, r6
MUL r8, r9, r10
DIV r12, r13, r14
```

Solution

There are no instruction hazards in this code fragment, so instructions proceed through the pipeline at one stage per cycle.



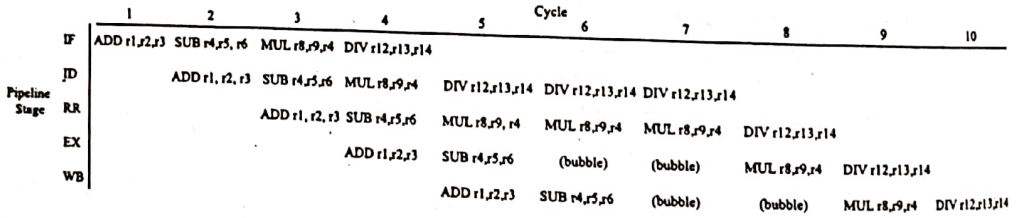
Pipelined Execution With Hazards (I)

6.10. Assuming no result forwarding and the five-stage sample pipeline of Section 6.3, draw a pipeline execution diagram similar to Fig. 6-2 for the following code fragment:

```
ADD r1, r2, r3
SUB r4, r5, r6
MUL r8, r9, r4
DIV r12, r13, r14
```

Solution

Here, there is a RAW hazard between the SUB instruction, which writes r4, and the MUL instruction, which reads r4. Therefore, the MUL instruction will not be able to read its input registers until after the SUB instruction has completed the WB stage, creating a pipeline stall.



Pipelined Execution With Hazards (II)

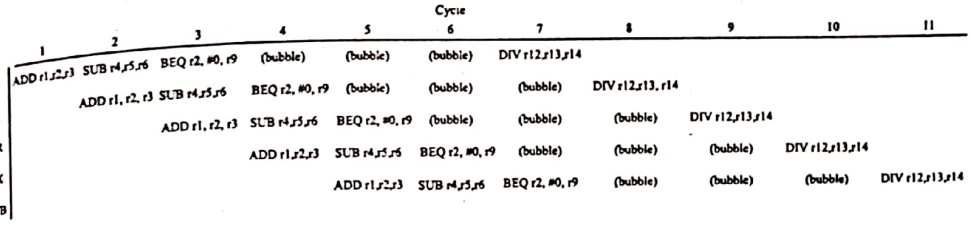
6.11. Assuming no result forwarding and the five-stage sample pipeline of Section 6.3, draw a pipeline execution diagram similar to Fig. 6-2 for the following code fragment. Assume that the branch represented by the BEQ instruction is not taken.

```
ADD r1, r2, r3
SUB r4, r5, r6
BEQ r2, #0, r9
DIV r12, r13, r14
```

Solution

Here, the stall occurs because the BEQ instruction must complete the execution stage before the instruction fetch stage knows what address the next instruction should be fetched

from, causing a stall. In general, processors have a direct path from the execution unit to the instruction fetch stage, allowing them to fetch the next instruction on the cycle after a branch reaches the execution stage.



Execution Time on Pipelines

6.12. What is the execution time (in cycles) of the following instruction sequence on our example five-stage pipeline (without bypassing)? Assume the branch is not taken. If the processor has a 2-ns clock cycle, what is the execution time in ns?

```
ADD r1, r4, r7
BEQ r2, #0, r1
SUB r8, r10, r11
MUL r12, r13, r14
```

Solution

The pipeline has a depth of 5 stages, giving a pipeline latency of 5 cycles. The BEQ instruction depends on the result of the ADD, so it issues on cycle $n + 3$, assuming the ADD issues on cycle n. The SUB has to wait for the four-cycle branch delay of the BEQ, so it issues on cycle $n + 7$, and the MUL issues on cycle $n + 8$, giving 9 cycles as the time to issue this program. Total execution time is $5 + 9 - 1 = 13$ cycles.

At 2 ns/cycle, this is 26 ns.

Instruction Ordering

6.13. a. What is the execution time (in cycles) of the following instruction sequence on our five-stage pipeline (without bypassing)?

```
ADD r3, r4, r5
SUB r7, r3, r9
MUL r8, r9, r10
ASH r4, r8, r12
```

b. Can the execution time of the instruction sequence be improved by reordering the instructions without changing the result of the computation? If so, show the instruction sequence with the shortest execution time and give its execution time.

Solution

a. The five-stage pipeline has a latency of 5 cycles. Assuming the ADD issues on cycle n, the SUB can issue on cycle $n + 3$ because it depends on the ADD, which has a 3-cycle instruction latency. The MUL is independent of the ADD and SUB, so it issues on cycle

$n + 4$, and the ASH issues on cycle $n + 7$, because it depends on the MUL. Therefore, it takes 8 cycles to issue all of the instructions in the program, and the execution time is $5 + 8 - 1 = 12$ cycles.

b. Yes, there is a better ordering. The ordering with the shortest execution time is as follows:

```
ADD r3, r4, r5
MUL r8, r9, r10
SUB r7, r3, r9
ASH r4, r8, r12
```

In this sequence, the only pipeline stall that occurs is between the MUL and the SUB instructions, because the SUB can't execute until 3 cycles after the ADD executes. The dependency between the MUL and the ASH does not cause any stalls because the MUL completes before the cycle after the SUB enters the execute stage, which is the first opportunity for the ASH to enter the execute stage. Therefore, it takes 5 cycles to issue all of the instructions in the sequence. The execution time of this sequence is $5 + 5 - 1 = 9$ cycles.

Instruction Ordering (II)

6.14. Compute the execution time of the following instruction sequence on our five-stage pipeline without bypassing. Then, find the reordering of the instructions that gives the shortest execution time and compute that execution time.

```
MUL r10, r11, r12
SUB r8, r10, r15
ADD r13, r14, r0
ASH r15, r2, r3
OR r7, r5, r6
```

Solution

The execution time of the original sequence is 11 cycles. A reordering that gives minimal execution time is as follows:

```
MUL r10, r11, r12
ADD r13, r14, r0
OR r7, r5, r6
SUB r8, r10, r15
ASH r15, r2, r3
```

The execution time of this ordering is 9 cycles, because no pipeline stalls occur. Note that it would not be possible to move the ASH instruction before the SUB instruction because of the WAR hazard between these instructions.

Bypassing (I)

6.15. Why does bypassing usually eliminate or reduce stalls due to data dependencies, but has no effect on stalls due to control hazards?

Solution

Bypassing eliminates the time required to write a result back into the register file and read the result out of the register file in pipelines without bypassing. For data dependencies, this is a significant fraction of the instruction latency, so bypassing improves performance. Because the transmission of the result address from a branch instruction to the instruction fetch stage does not go through the register file, bypassing does not improve the performance of branch instructions. Another way of looking at this is that the base pipeline already has a bypass path from the execution unit to the instruction fetch stage, so no improvement is gained from adding additional bypass paths.

Bypassing (II)

6.16. What is the execution time of the code sequence in Problem 6.12 if bypassing is added to our base pipeline?

Solution

The execution time would be 11 cycles. Since our base pipeline has only one execute stage, bypassing reduces the instruction latency of non-branch instructions to one cycle. This eliminates the two cycles of delay between the ADD and the branch, but not the branch delay.

Comprehensive Example

6.17. This exercise will consider the seven-stage pipeline shown in Fig. 6-13.

a. If the clock period of this pipeline is 4 ns, what is the latency of the pipeline in cycles and ns?

b. Draw a pipeline execution diagram for this pipeline, showing how each of the following instruction sequences flow through the pipeline. Assume that the pipeline does not implement bypassing.

Sequence 1:

```
ADD r1, r2, r3
SUB r4, r1, r5
```

Sequence 2: (assume branch not taken)

```
BNE r9, #3, r8
OR r12, r14, r15
```

c. Based on these diagrams, what are the non-branch and branch instruction latencies for this pipeline?

d. Given the branch and non-branch instruction latencies for this pipeline, what is the execution time of the code sequence in Problem 6.12 on this pipeline?

e. If result forwarding were implemented on this pipeline, what would the non-branch and branch instruction latencies be?

f. What would the execution time of the code sequence from Problem 6.12 be on this pipeline with result forwarding?

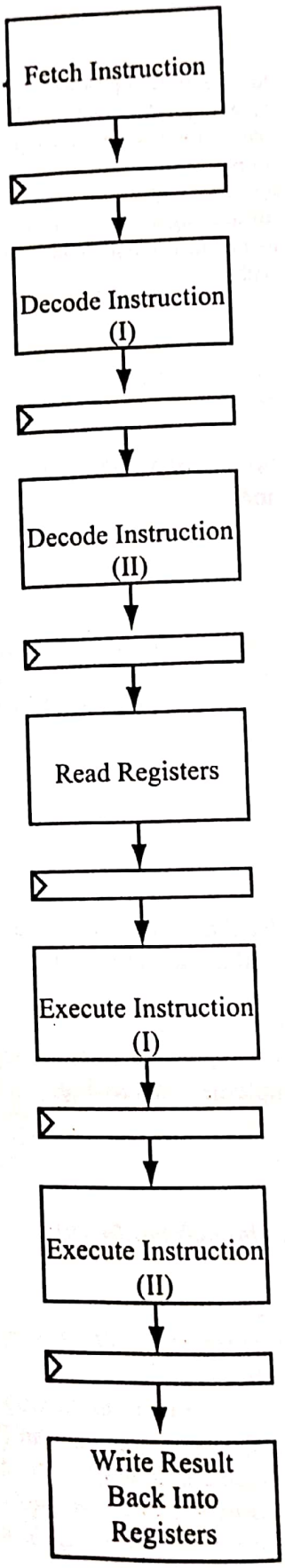
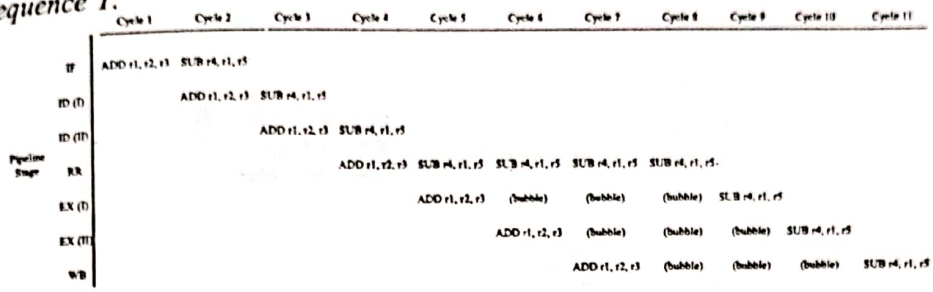


Fig. 6-13. Seven-stage pipeline.

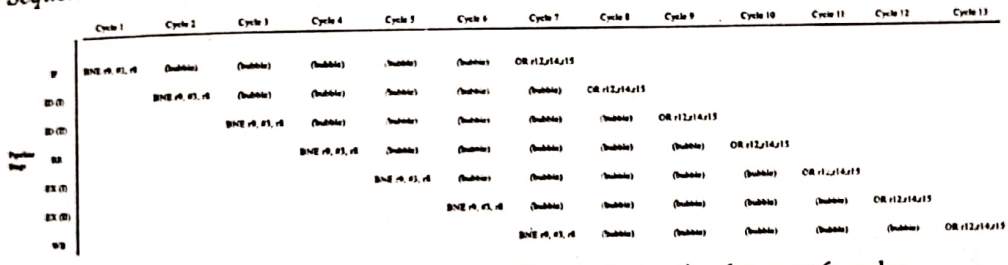
Solution

a. The pipeline has seven stages. Therefore, the latency of the pipeline is seven cycles. At 4ns/cycle, this is 28 ns.

b. *Sequence 1:*



Sequence 2:



c. Non-branch instruction latency: 4 cycles. Branch instruction latency: 6 cycles.

d. The execution time would be 18 cycles.

e. The non-branch latency would become 2 cycles because of the two execute stages. The branch latency would remain 6 cycles because result forwarding does not improve branch latency.

f. 16 cycles.