FUTURE VISION BIE

One Stop for All Study Materials

& Lab Programs

Future Vision

By K B Hemanth Raj

Scan the QR Code to Visit the Web Page

Or

Visit : https://hemanthrajhemu.github.io

Gain Access to All Study Materials according to VTU,
CSE – Computer Science Engineering,
ISE – Information Science Engineering,
ECE - Electronics and Communication Engineering
& MORE...

Join Telegram to get Instant Updates: https://bit.ly/VTU_TELEGRAM

Contact: MAIL: futurevisionbie@gmail.com

INSTAGRAM: www.instagram.com/hemanthraj_hemu/

INSTAGRAM: www.instagram.com/futurevisionbie/

WHATSAPP SHARE: https://bit.ly/FVBIESHARE

COMPUTER ORGANIZATION
AND EMBEDDED SYSTEMS

SIXTH EDITION

Carl Hamacher
Queen's University

Zvonko Vranesic
University of Toronto

Safwat Zaky
University of Toronto

Naraig Manjikian
Queen's University

chapter

5

Basic Processing Unit

Chapter Objectives

In this chapter you will learn about:

- Execution of instructions by a processor
- The functional units of a processor and how they are interconnected
- Hardware for generating control signals
- Microprogrammed control

In this chapter we focus on the processing unit, which executes machine-language instructions and coordinates the activities of other units in a computer. We examine its internal structure and show how it performs the tasks of fetching, decoding, and executing such instructions. The processing unit is often called the *central processing unit* (CPU). The term "central" is not as appropriate today as it was in the past, because today's computers often include several processing units. We will use the term *processor* in this discussion.

The organization of processors has evolved over the years, driven by developments in technology and the desire to provide high performance. To achieve high performance, it is prudent to make various functional units of a processor operate in parallel as much as possible. Such processors have a *pipelined* organization where the execution of an instruction is started before the execution of the preceding instruction is completed. Another approach, known as *superscalar* operation, is to fetch and start the execution of several instructions at the same time. Pipelining and superscalar approaches are discussed in Chapter 6. In this chapter, we concentrate on the basic ideas that are common to all processors.

5.1 SOME FUNDAMENTAL CONCEPTS

A typical computing task consists of a series of operations specified by a sequence of machine-language instructions that constitute a program. The processor fetches one instruction at a time and performs the operation specified. Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered. The processor uses the *program counter*, PC, to keep track of the address of the next instruction to be fetched and executed. After fetching an instruction, the contents of the PC are updated to point to the next instruction in sequence. A branch instruction may cause a different value to be loaded into the PC.

When an instruction is fetched, it is placed in the *instruction register*, IR, from where it is interpreted, or decoded, by the processor's control circuitry. The IR holds the instruction until its execution is completed.

Consider a 32-bit computer in which each instruction is contained in one word in the memory, as in RISC-style instruction set architecture. To execute an instruction, the processor has to perform the following steps:

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are the instruction to be executed; hence they are loaded into the IR. In register transfer notation, the required action is

$$IR \leftarrow [[PC]]$$

2. Increment the PC to point to the next instruction. Assuming that the memory is byte addressable, the PC is incremented by 4; that is

$$PC \leftarrow [PC] + 4$$

3. Carry out the operation specified by the instruction in the IR.

Fetching an instruction and loading it into the IR is usually referred to as the *instruction fetch phase.* Performing the operation specified in the instruction constitutes the *instruction execution phase.*

With few exceptions, the operation specified by an instruction can be carried out by performing one or more of the following actions:

- Read the contents of a given memory location and load them into a processor register.
- Read data from one or more processor registers.
- Perform an arithmetic or logic operation and place the result into a processor register.
- Store data from a processor register into a given memory location.

The hardware components needed to perform these actions are shown in Figure 5.1. The processor communicates with the memory through the processor-memory interface, which transfers data from and to the memory during Read and Write operations. The instruction address generator updates the contents of the PC after every instruction is fetched. The register file is a memory unit whose storage locations are organized to form the processor's general-purpose registers. During execution, the contents of the registers named in an instruction that performs an arithmetic or logic operation are sent to the arithmetic and logic

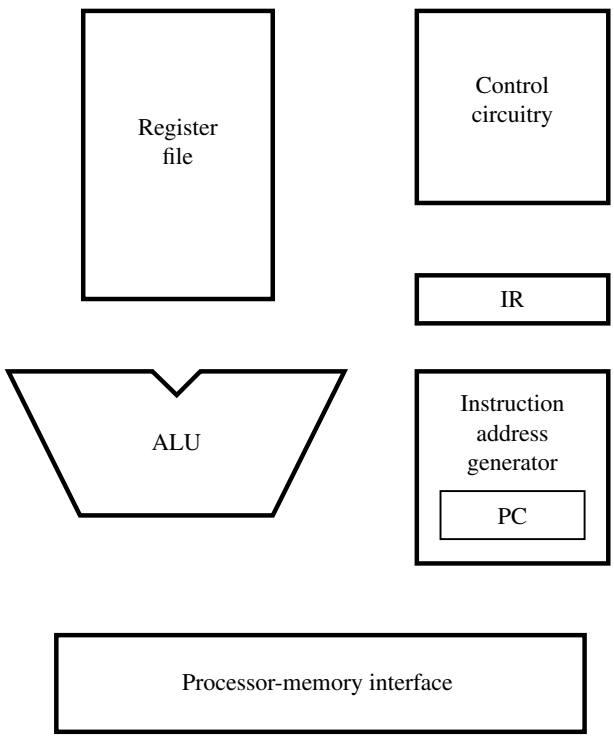


Figure 5.1 Main hardware components of a processor.

unit (ALU), which performs the required computation. The results of the computation are stored in a register in the register file.

Before we examine these units and their interaction in detail, it is helpful to consider the general structure of any data processing system.

Data Processing Hardware

A typical computation operates on data stored in registers. These data are processed by combinational circuits, such as adders, and the results are placed into a register. Figure 5.2 illustrates this structure. A clock signal is used to control the timing of data transfers. The registers comprise edge-triggered flip-flops into which new data are loaded at the active edge of the clock. In this chapter, we assume that the rising edge of the clock is the active edge. The clock period, which is the time between two successive rising edges, must be long enough to allow the combinational circuit to produce the correct result.

The operation performed by the combinational block in Figure 5.2 may be quite complex. It can often be broken down into several simpler steps, where each step is performed by a subcircuit of the original circuit. These subcircuits can then be cascaded into a multi-stage structure as shown in Figure 5.3. Then, if n stages are used, the operation will be completed in n clock cycles. Since these combinational subcircuits are smaller, they can complete their operation in less time, and hence a shorter clock period can be used. A key advantage of the multi-stage structure is that it is suitable for pipelined operation, as will be discussed in Chapter 6. Such a structure is particularly useful for implementing processors that have a RISC-style instruction set. The discussion in the remainder of this chapter focuses on processors that use a multi-stage structure of this type. In Section 5.7 we will consider a more traditional alternative that is suitable for CISC-style processors.

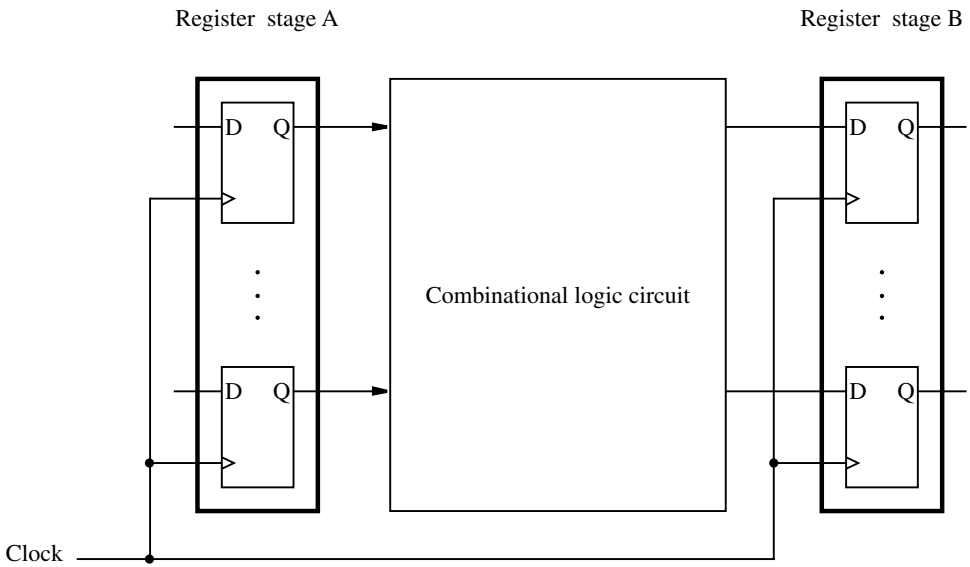


Figure 5.2 Basic structure for data processing.

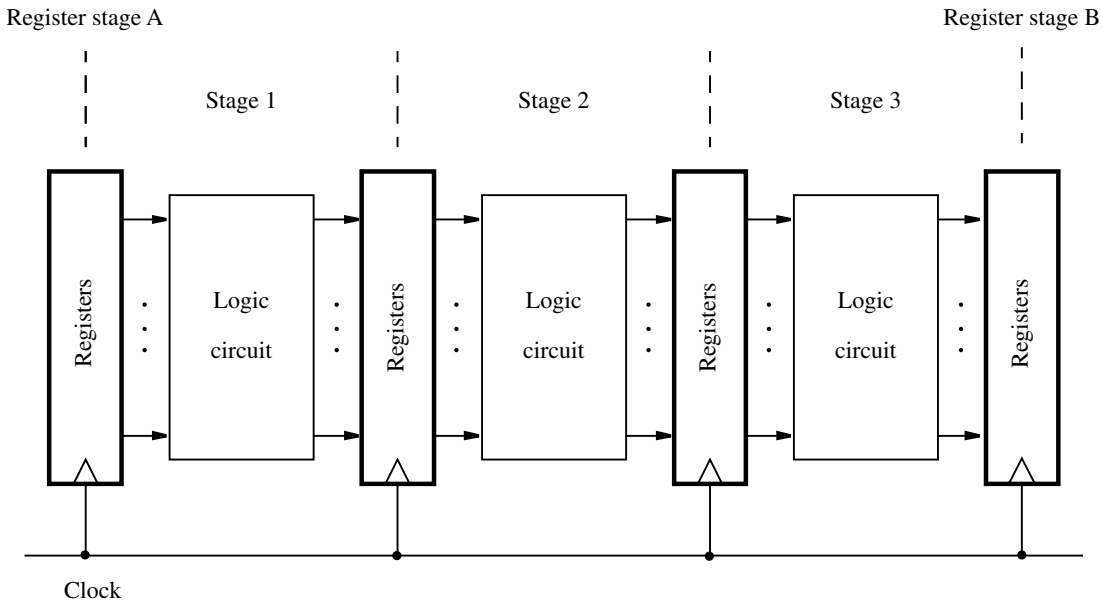


Figure 5.3 A hardware structure with multiple stages.

5.2 INSTRUCTION EXECUTION

Let us now examine the actions involved in fetching and executing instructions. We illustrate these actions using a few representative RISC-style instructions.

5.2.1 LOAD INSTRUCTIONS

Consider the instruction

Load R5, X(R7)

which uses the Index addressing mode to load a word of data from memory location X + [R7] into register R5. Execution of this instruction involves the following actions:

- Fetch the instruction from the memory.
- Increment the program counter.
- Decode the instruction to determine the operation to be performed.
- Read register R7.
- Add the immediate value X to the contents of R7.
- Use the sum X + [R7] as the effective address of the source operand, and read the contents of that location in the memory.
- Load the data received from the memory into the destination register, R5.

Depending on how the hardware is organized, some of these actions can be performed at the same time. In the discussion that follows, we will assume that the processor has five hardware stages, which is a commonly used arrangement in RISC-style processors. Execution of each instruction is divided into five steps, such that each step is carried out by one hardware stage. In this case, fetching and executing the Load instruction above can be completed as follows:

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read the contents of register R7 in the register file.
3. Compute the effective address.
4. Read the memory source operand.
5. Load the operand into the destination register, R5.

5.2.2 ARITHMETIC AND LOGIC INSTRUCTIONS

Instructions that involve an arithmetic or logic operation can be executed using similar steps. They differ from the Load instruction in two ways:

- There are either two source registers, or a source register and an immediate source operand.
- No access to memory operands is required.

A typical instruction of this type is

```
Add     R3, R4, R5
```

It requires the following steps:

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read the contents of source registers R4 and R5.
3. Compute the sum [R4] + [R5].
4. Load the result into the destination register, R3.

The Add instruction does not require access to an operand in the memory, and therefore could be completed in four steps instead of the five steps needed for the Load instruction. However, as we will see in the next chapter, it is advantageous to use the same multi-stage processing hardware for as many instructions as possible. This can be achieved if we arrange for all instructions to be executed in the same number of steps. To this end, the Add instruction should be extended to five steps, patterned along the steps of the Load instruction. Since no access to memory operands is required, we can insert a step in which no action takes place between steps 3 and 4 above. The Add instruction would then be performed as follows:

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read registers R4 and R5.
3. Compute the sum [R4] + [R5].

4. No action.
5. Load the result into the destination register, R3.

If the instruction uses an immediate operand, as in

```
Add     R3, R4, #1000
```

the immediate value is given in the instruction word. Once the instruction is loaded into the IR, the immediate value is available for use in the addition operation. The same five-step sequence can be used, with steps 2 and 3 modified as:

2. Decode the instruction and read register R4.
3. Compute the sum [R4] + 1000.

5.2.3 STORE INSTRUCTIONS

The five-step sequence used for the Load and Add instructions is also suitable for Store instructions, except that the final step of loading the result into a destination register is not required. The hardware stage responsible for this step takes no action. For example, the instruction

```
Store     R6, X(R8)
```

stores the contents of register R6 into memory location X + [R8]. It can be implemented as follows:

1. Fetch the instruction and increment the program counter.
2. Decode the instruction and read registers R6 and R8.
3. Compute the effective address X + [R8].
4. Store the contents of register R6 into memory location X + [R8].
5. No action.

After reading register R8 in step 2, the memory address is computed in step 3 using the immediate value, X, in the IR. In step 4, the contents of R6 are sent to the memory to be stored. No action is taken in step 5.

In summary, the five-step sequence of actions given in Figure 5.4 is suitable for all instructions in a RISC-style instruction set. RISC-style instructions are one word long and only Load and Store instructions access operands in the memory, as explained in Chapter 2. Instructions that perform computations use data that are either stored in general-purpose registers or given as immediate data in the instruction.

The five-step sequence is suitable for all Load and Store instructions, because the addressing modes that can be used in these instructions are special cases of the Index mode. Most RISC-style processors provide one general-purpose register, usually register R0, that always contains the value zero. When R0 is used as the index register, the effective address of the operand is the immediate value X. This is the Absolute addressing mode. Alternatively, if the offset X is set to zero, the effective address is the contents of the index register, R*i*. This is the Indirect addressing mode. Thus, only one addressing mode, the Index mode,

Step	Action
1	Fetch an instruction and increment the program counter.
2	Decode the instruction and read registers from the register file.
3	Perform an ALU operation.
4	Read or write memory data if the instruction involves a memory operand.
5	Write the result into the destination register, if needed.

Figure 5.4 A five-step sequence of actions to fetch and execute an instruction.

needs to be implemented, resulting in a significant simplification of the processor hardware. The task of selecting R0 as the index register or setting X to zero is left to the assembler or the compiler. This is consistent with the RISC philosophy of aiming for simple and fast hardware at the expense of higher compiler complexity and longer compilation time. The result is a net gain in the time needed to perform various tasks on a computer, because programs are compiled much less frequently than they are executed.

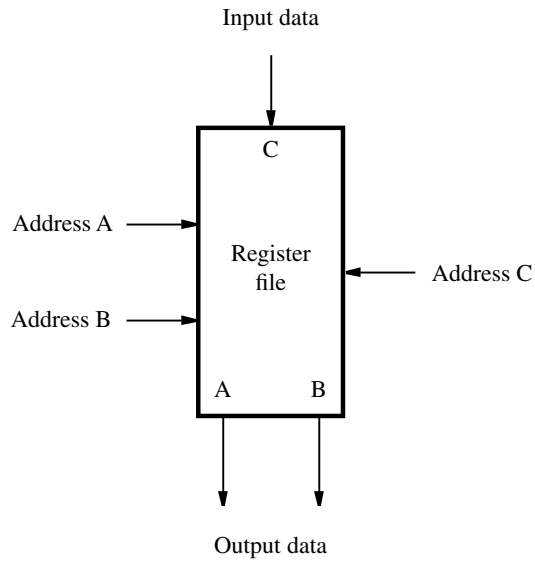
5.3 HARDWARE COMPONENTS

The discussion above indicates that all instructions of a RISC-style processor can be executed using the five-step sequence in Figure 5.4. Hence, the processor hardware may be organized in five stages, such that each stage performs the actions needed in one of the steps. We now examine the components in Figure 5.1 to see how they may be organized in the multi-stage structure of Figure 5.3.

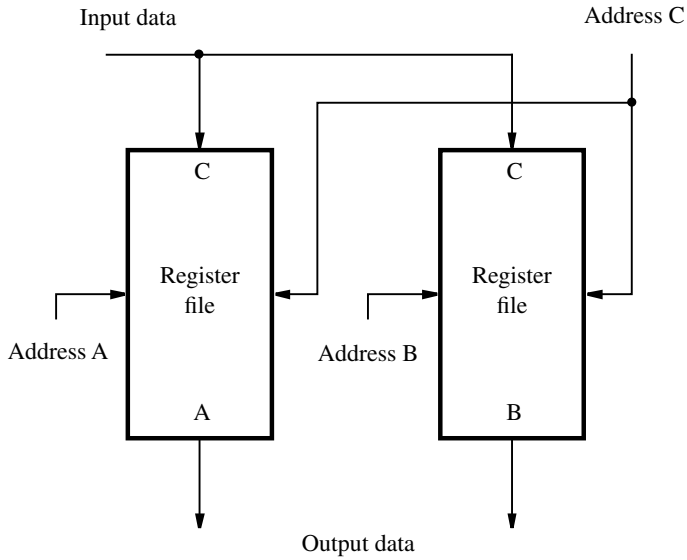
5.3.1 REGISTER FILE

General-purpose registers are usually implemented in the form of a register file, which is a small and fast memory block. It consists of an array of storage elements, with access circuitry that enables data to be read from or written into any register. The access circuitry is designed to enable two registers to be read at the same time, making their contents available at two separate outputs, A and B. The register file has two address inputs that select the two registers to be read. These inputs are connected to the fields in the IR that specify the source registers, so that the required registers can be read. The register file also has a data input, C, and a corresponding address input to select the register into which data are to be written. This address input is connected to the IR field that specifies the destination register of the instruction.

The inputs and outputs of any memory unit are often called input and output *ports*. A memory unit that has two output ports is said to be *dual-ported*. Figure 5.5 shows two ways



(a) Single memory block



(b) Two memory blocks

Figure 5.5 Two alternatives for implementing a dual-ported register file.

of realizing a dual-ported register file. One possibility is to use a single set of registers with duplicate data paths and access circuitry that enable two registers to be read at the same time. An alternative is to use two memory blocks, each containing one copy of the register file. Whenever data are written into a register, they are written into both copies of that register. Thus, the two files have identical contents. When an instruction requires data from two registers, one register is accessed in each file. In effect, the two register files together function as a single dual-ported register file.

5.3.2 ALU

The arithmetic and logic unit is used to manipulate data. It performs arithmetic operations such as addition and subtraction, and logic operations such as AND, OR, and XOR. Conceptually, the register file and the ALU may be connected as shown in Figure 5.6. When an instruction that performs an arithmetic or logic operation is being executed, the contents of the two registers specified in the instruction are read from the register file and become

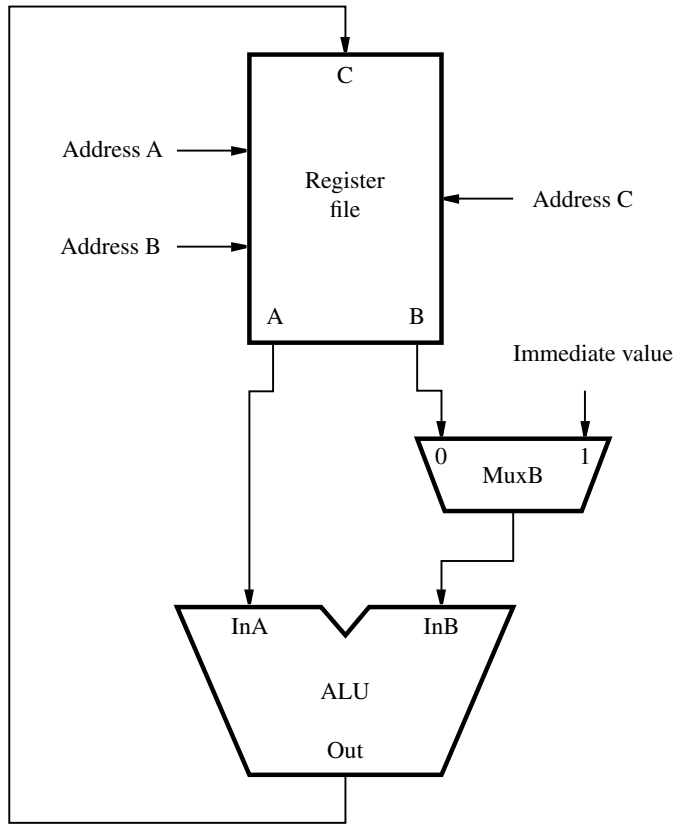


Figure 5.6 Conceptual view of the hardware needed for computation.

available at outputs A and B. Output A is connected directly to the first input of the ALU, InA, and output B is connected to a multiplexer, MuxB. The multiplexer selects either output B of the register file or the immediate value in the IR to be connected to the second ALU input, InB. The output of the ALU is connected to the data input, C, of the register file so that the results of a computation can be loaded into the destination register.

5.3.3 DATAPATH

Instruction processing consists of two phases: the fetch phase and the execution phase. It is convenient to divide the processor hardware into two corresponding sections. One section fetches instructions and the other executes them. The section that fetches instructions is also responsible for decoding them and for generating the control signals that cause appropriate actions to take place in the execution section. The execution section reads the data operands specified in an instruction, performs the required computations, and stores the results.

We now need to organize the hardware into a multi-stage structure similar to that in Figure 5.3, with stages corresponding to the five steps in Figure 5.4. A possible structure is shown in Figure 5.7. The actions taken in each of the five stages are completed in one clock cycle. An instruction is fetched in step 1 by hardware stage 1 and placed into the IR. It is decoded, and its source registers are read in step 2. The information in the IR is used to generate the control signals for all subsequent steps. Therefore, the IR must continue to hold the instruction until its execution is completed.

It is necessary to insert registers between stages. Inter-stage registers hold the results produced in one stage so that they can be used as inputs to the next stage during the next clock cycle. This leads to the organization in Figure 5.8. The hardware in the figure is often referred to as the *datapath*. It corresponds to stages 2 to 5 in Figure 5.7. Data read from the register file are placed in registers RA and RB. Register RA provides the data to input InA of the ALU. Multiplexer MuxB forwards either the contents of RB or the immediate value in the IR to the ALU's second input, InB. The ALU constitutes stage 3, and the result of the computation it performs is placed in register RZ.

Recall that for computational instructions, such as an Add instruction, no processing actions take place in step 4. During that step, multiplexer MuxY in Figure 5.8 selects register RZ to transfer the result of the computation to RY. The contents of RY are transferred to the register file in step 5 and loaded into the destination register. For this reason, the register file is in both stages 2 and 5. It is a part of stage 2 because it contains the source registers and a part of stage 5 because it contains the destination register.

For Load and Store instructions, the effective address of the memory operand is computed by the ALU in step 3 and loaded into register RZ. From there, it is sent to the memory, which is stage 4. In the case of a Load instruction, the data read from the memory are selected by multiplexer MuxY and placed in register RY, to be transferred to the register file in the next clock cycle. For a Store instruction, data are read from the register file, which is part of stage 2, and placed in register RB. Since memory access is done in stage 4, another inter-stage register is needed to maintain correct data flow in the multi-stage structure. Register RM is introduced for this purpose. The data to be stored are moved from RB to RM in step 3, and from there to the memory in step 4. No action is taken in step 5 in this case.

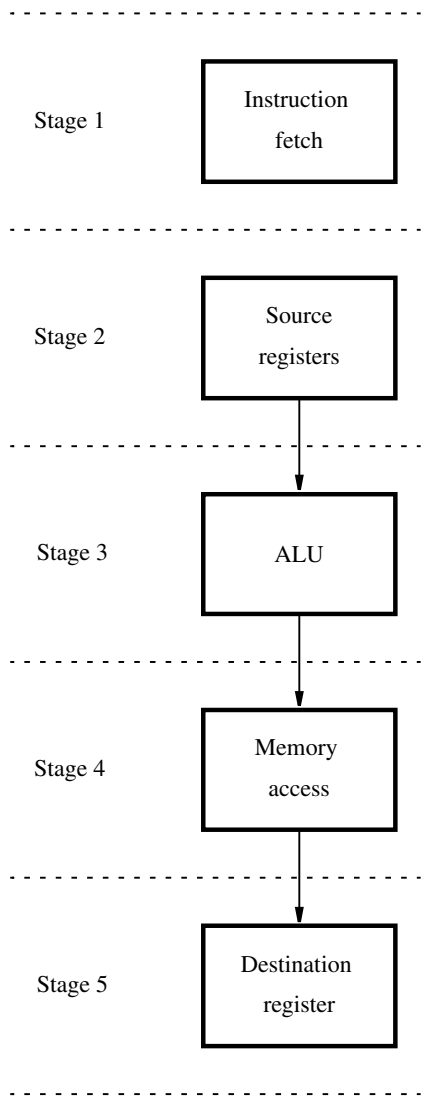


Figure 5.7 A five-stage organization.

The subroutine call instructions introduced in Section 2.7 save the return address in a general-purpose register, which we call LINK for ease of reference. Similarly, interrupt processing requires a return address to be saved, as described in Section 3.2. Assume that another general-purpose register, IRA, is used for this purpose. Both of these actions require the contents of the program counter to be sent to the register file. For this reason, multiplexer MuxY has a third input through which the return address can be routed to register RY, from where it can be sent to the register file. The return address is produced by the instruction address generator, as we will explain later.

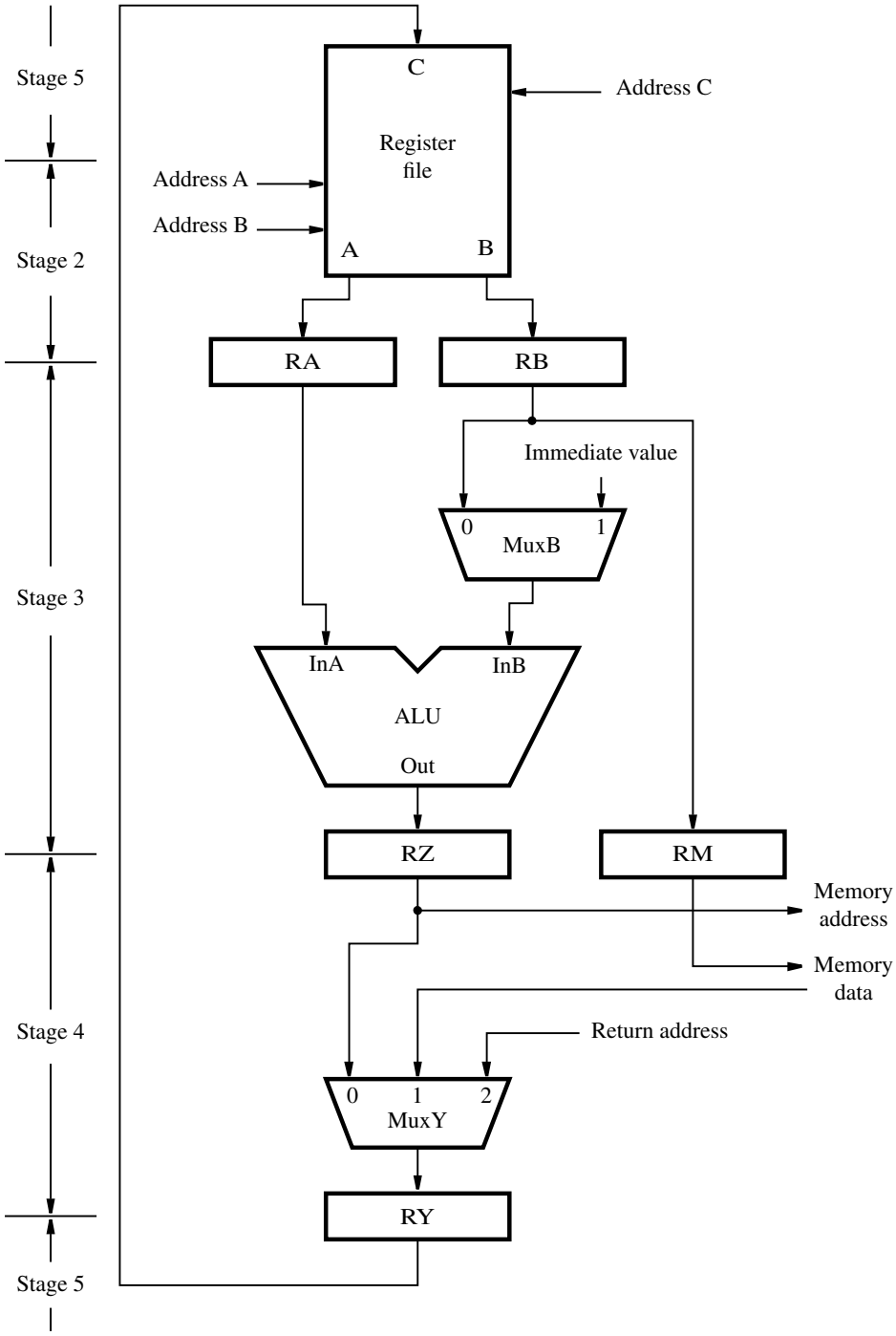


Figure 5.8 Datapath in a processor.

5.3.4 INSTRUCTION FETCH SECTION

The organization of the instruction fetch section of the processor is illustrated in Figure 5.9. The addresses used to access the memory come from the PC when fetching instructions and from register RZ in the datapath when accessing instruction operands. Multiplexer MuxMA selects one of these two sources to be sent to the processor-memory interface. The PC is included in a larger block, the instruction address generator, which updates the contents of the PC after each instruction is fetched. The instruction read from the memory is loaded into the IR, where it stays until its execution is completed and the next instruction is fetched.

The contents of the IR are examined by the control circuitry to generate the signals needed to control all the processor's hardware. They are also used by the block labeled Immediate. As described in Chapter 2, an immediate value may be included in some instructions. A 16-bit immediate value is extended to 32 bits. The extended value is then used either directly as an operand or to compute the effective address of an operand. For some instructions, such as those that perform arithmetic operations, the immediate value is sign-extended; for others, such as logic instructions, it is padded with zeros. The Immediate block in Figure 5.9 generates the extended value and forwards it to MuxB in Figure 5.8 to be used in an ALU computation. It also generates the extended value to be used in computing the target address of branch instructions.

The address generator circuit is shown in Figure 5.10. An adder is used to increment the PC by 4 during straight-line execution. It is also used to compute a new value to be

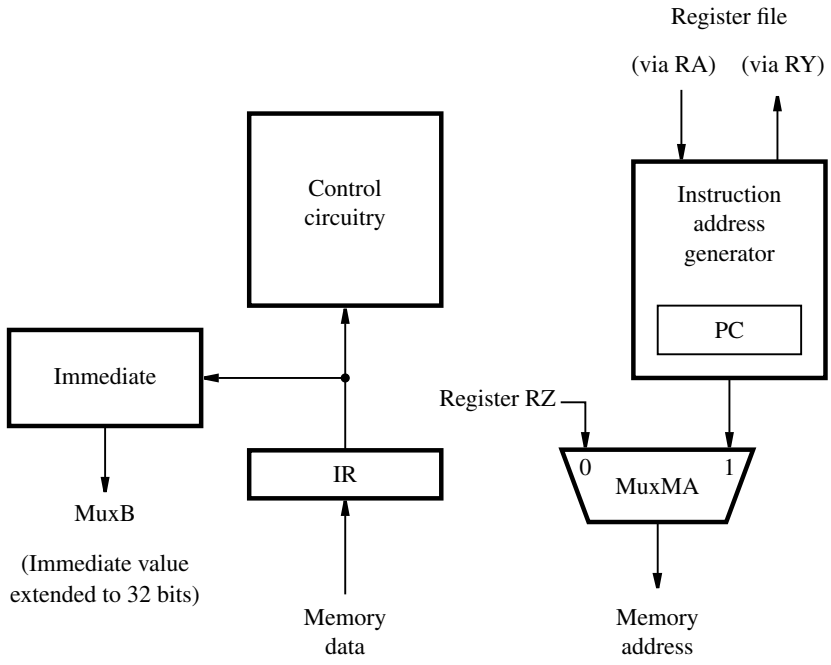


Figure 5.9 Instruction fetch section of Figure 5.7.

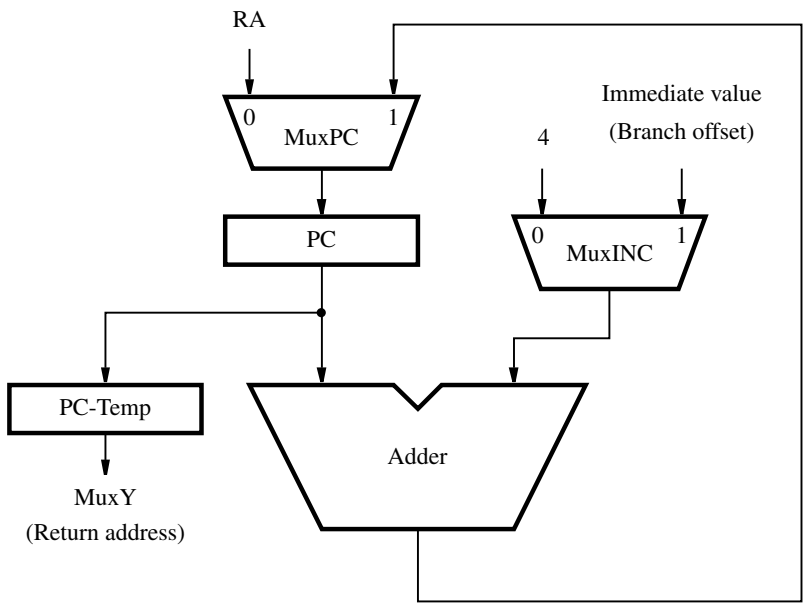


Figure 5.10 Instruction address generator.

loaded into the PC when executing branch and subroutine call instructions. One adder input is connected to the PC. The second input is connected to a multiplexer, MuxINC, which selects either the constant 4 or the branch offset to be added to the PC. The branch offset is given in the immediate field of the IR and is sign-extended to 32 bits by the Immediate block in Figure 5.9. The output of the adder is routed to the PC via a second multiplexer, MuxPC, which selects between the adder and the output of register RA. The latter connection is needed when executing subroutine linkage instructions. Register PC-Temp is needed to hold the contents of the PC temporarily during the process of saving the subroutine or interrupt return address.

5.4 INSTRUCTION FETCH AND EXECUTION STEPS

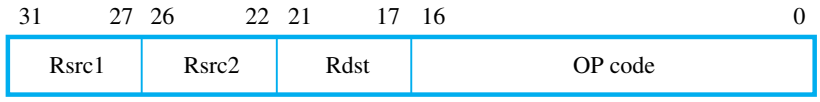
We now examine the process of fetching and executing instructions in more detail, using the datapath in Figure 5.8. Consider again the instruction

Add R3, R4, R5

The steps for fetching and executing this instruction are given in Figure 5.11. Assume that the instruction is encoded using the format in Figure 2.32, which is reproduced here as Figure 5.12. After the instruction has been fetched from the memory and placed in the IR, the source register addresses are available in fields IR_{31-27} and IR_{26-22}. These two fields

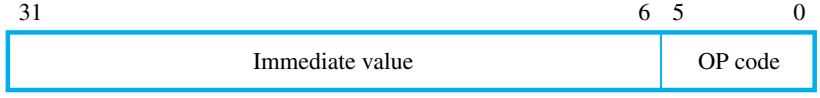
Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R4], RB ← [R5]
3	RZ ← [RA] + [RB]
4	RY ← [RZ]
5	R3 ← [RY]

Figure 5.11 Sequence of actions needed to fetch and execute the instruction: Add R3, R4, R5.



(a) Register-operand format

(b) Immediate-operand format



(c) Call format

Figure 5.12 Instruction encoding.

are connected to the address inputs for ports A and B of the register file. As a result, registers R4 and R5 are read and their contents placed in registers RA and RB, respectively, at the end of step 2. In the next step, the control circuitry sets MuxB to select input 0, thus connecting register RB to input InB of the ALU. At the same time, it causes the ALU to perform an addition operation. Since register RA is connected to input InA, the ALU produces the required sum [RA] + [RB], which is loaded into register RZ at the end of step 3.

In step 4, multiplexer MuxY selects input 0, thus causing the contents of RZ to be transferred to RY. The control circuitry connects the destination address field of the Add instruction, IR_{21-17}, to the address input for port C of the register file. In step 5, it issues

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R7]
3	RZ ← [RA] + Immediate value X
4	Memory address ← [RZ], Read memory, RY ← Memory data
5	R5 ← [RY]

Figure 5.13 Sequence of actions needed to fetch and execute the instruction: Load R5, X(R7).

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R8], RB ← [R6]
3	RZ ← [RA] + Immediate value X, RM ← [RB]
4	Memory address ← [RZ], Memory data ← [RM], Write memory
5	No action

Figure 5.14 Sequence of actions needed to fetch and execute the instruction: Store R6, X(R8).

a Write command to the register file, causing the contents of register RY to be written into register R3.

Load and Store instructions are executed in a similar manner. In this case, the address of the destination register is given in bit field IR_{26-22}. The control hardware connects this field to the address input corresponding to input C of the register file. The steps involved in executing these instructions are given in Figures 5.13 and 5.14. In both examples, the memory address is specified using the Index mode, in which the index value X is given as an immediate value in the instruction. The immediate field of IR, extended as appropriate by the Immediate block in Figure 5.9, is selected by MuxB in step 3 and added to the contents of register RA. The resulting sum is the effective address of the operand.

Some Observations

In the discussion above, we assumed that memory Read and Write operations can be completed in one clock cycle. Is this a realistic assumption? In general, accessing the main memory of a computer takes significantly longer than reading the contents of a register in the register file. However, most modern processors use cache memories, which will be discussed in detail in Chapter 8. A cache memory is much faster than the main memory.

It is usually implemented on the same chip as the processor, making it about as fast as the register file. Thus, a memory Read or Write operation can be completed in one clock cycle when the data involved are available in the cache. When the operation requires access to the main memory, the processor must wait for that operation to be completed. We will discuss how slower memory accesses are handled in Section 5.4.2.

We also assumed that the processor reads the source registers of the instruction in step 2, while it is still decoding the OP code of the instruction that has just been loaded into the IR. Can these two tasks be completed in the same step? How can the control hardware know which registers to read before it completes decoding the instruction? This is possible because source register addresses are specified using the same bit positions in all instructions. The hardware reads the registers whose addresses are in these bit positions once the instruction is loaded into the IR. Their contents are loaded into registers RA and RB at the end of step 2. If these data are needed by the instruction, they will be available for use in step 3. If not, they will be ignored by subsequent hardware stages.

Note that the actions described in Figures 5.11, 5.13, and 5.14 do not show two registers being read in step 2 in every case. To avoid confusion, only the registers needed by the specific instruction described in the figure are mentioned, even though two registers are always read.

5.4.1 BRANCHING

Instructions are fetched from sequential word locations in the memory during straight-line program execution. Whenever an instruction is fetched, the processor increments the PC by 4 to point to the next word. This execution pattern continues until a branch or subroutine call instruction loads a new address into the PC. Subroutine call instructions also save the return address, to be used when returning to the calling program. In this section we examine the actions needed to implement these instructions. Interrupts from I/O devices and software interrupt instructions are handled in a similar manner.

Branch instructions specify the branch target address relative to the PC. A branch offset given as an immediate value in the instruction is added to the current contents of the PC. The number of bits used for this offset is considerably less than the word length of the computer, because space is needed within the instruction to specify the OP code and the branch condition. Hence, the range of addresses that can be reached by a branch instruction is limited.

Subroutine call instructions can reach a larger range of addresses. Because they do not include a condition, more bits are available to specify the target address. Also, most RISC-style computers have Jump and Call instructions that use a general-purpose register to specify a full 32-bit address. The details vary from one computer to another, as the example processors introduced in Appendices B to E illustrate.

Branch Instructions

The sequence of steps for implementing an unconditional branch instruction is given in Figure 5.15. The instruction is fetched and the PC is incremented as usual in step 1. After the instruction has been decoded in step 2, multiplexer MuxINC selects the branch offset in

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction
3	PC ← [PC] + Branch offset
4	No action
5	No action

Figure 5.15 Sequence of actions needed to fetch and execute an unconditional branch instruction.

the IR to be added to the PC in step 3. This is the address that will be used to fetch the next instruction. Execution of a Branch instruction is completed in step 3. No action is taken in steps 4 and 5.

We explained in Section 2.13 that the branch offset is the distance between the branch target and the memory location following the branch instruction. The reason for this can be seen clearly in Figure 5.15. The PC is incremented by 4 in step 1, at the time the branch instruction is fetched. Then, the branch target address is computed in step 3 by adding the branch offset to the updated contents of the PC.

The sequence in Figure 5.15 can be readily modified to implement conditional branch instructions. In processors that do not use condition-code flags, the branch instruction specifies a compare-and-test operation that determines the branch condition. For example, the instruction

Branch_if_[R5]=[R6] LOOP

results in a branch if the contents of registers R5 and R6 are identical. When this instruction is executed, the register contents are compared, and if they are equal, a branch is made to location LOOP.

Figure 5.16 shows how this instruction may be executed. Registers R5 and R6 are read in step 2, as usual, and compared in step 3. The comparison could be done by performing the subtraction operation [R5] − [R6] in the ALU. The ALU generates signals that indicate whether the result of the subtraction is positive, negative, or zero. The ALU may also generate signals to show whether arithmetic overflow has occurred and whether the operation produced a carry-out. The control circuitry examines these signals to test the condition given in the branch instruction. In the example above, it checks whether the result of the subtraction is equal to zero. If it is, the branch target address is loaded into the PC, to be used to fetch the next instruction. Otherwise, the contents of the PC remain at the incremented value computed in step 1, and straight-line execution continues.

According to the sequence of steps in Figure 5.16, the two actions of comparing the register contents and testing the result are both carried out in step 3. Hence, the clock cycle must be long enough for the two actions to be completed, one after the other. For this reason, it is desirable that the comparison be done as quickly as possible. A subtraction

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R5], RB ← [R6]
3	Compare [RA] to [RB], If [RA] = [RB], then PC ← [PC] + Branch offset
4	No action
5	No action

Figure 5.16 Sequence of actions needed to fetch and execute the instruction: Branch_if_[R5]=[R6] LOOP.

operation in the ALU is time consuming, and is not needed in this case. A simpler and faster comparator circuit can examine the contents of registers RA and RB and produce the required condition signals, which indicate the conditions greater than, equal, less than, etc. A comparator is not shown separately in Figure 5.8 as it can be a part of the ALU block. Example 5.3 shows how a comparator circuit can be designed.

Subroutine Call Instructions

Subroutine calls and returns are implemented in a similar manner to branch instructions. The address of the subroutine may either be computed using an immediate value given in the instruction or it may be given in full in one of the general-purpose registers. Figure 5.17 gives the sequence of actions for the instruction

Call_Register R9

which calls a subroutine whose address is in register R9. The contents of that register are read and placed in RA in step 2. During step 3, multiplexer MuxPC selects its 0 input, thus transferring the data in register RA to be loaded into the PC.

Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R9]
3	PC-Temp ← [PC], PC ← [RA]
4	RY ← [PC-Temp]
5	Register LINK ← [RY]

Figure 5.17 Sequence of actions needed to fetch and execute the instruction: Call_Register R9.

Assume that the return address of the subroutine, which is the previous contents of the PC, is to be saved in a general-purpose register called LINK in the register file. Data are written into the register file in step 5. Hence, it is not possible to send the return address directly to the register file in step 3. To maintain correct data flow in the five-stage structure, the processor saves the return address in a temporary register, PC-Temp. From there, the return address is transferred to register RY in step 4, then to register LINK in step 5. The address LINK is built into the control circuitry.

Subroutine return instructions transfer the value saved in register LINK back to the PC. The encoding of the Return-from-subroutine instruction is such that the address of register LINK appears in bits IR_{31-27}. This is the field connected to Address A of the register file. Hence, once the instruction is fetched, register LINK is read and its contents are placed in RA, from where they can be transferred to the PC via MuxPC in Figure 5.10. Return-from-interrupt instructions are handled in a similar manner, except that a different register is used to hold the return address.

5.4.2 WAITING FOR MEMORY

The role of the processor-memory interface circuit is to control data transfers between the processor and the memory. We pointed out earlier that modern processors use fast, on-chip cache memories. Most of the time, the instruction or data referenced in memory Read and Write operations are found in the cache, in which case the operation is completed in one clock cycle. When the requested information is not in the cache and has to be fetched from the main memory, several clock cycles may be needed. The interface circuit must inform the processor's control circuitry about such situations, to delay subsequent execution steps until the memory operation is completed.

Assume that the processor-memory interface circuit generates a signal called Memory Function Completed (MFC). It asserts this signal when a requested memory Read or Write operation has been completed. The processor's control circuitry checks this signal during any processing step in which it issues a memory Read or Write request, to determine when it can proceed to the next step. When the requested data are found in the cache, the interface circuit asserts the MFC signal before the end of the same clock cycle in which the memory request is issued. Hence, instruction execution continues uninterrupted. If access to the main memory is required, the interface circuit delays asserting MFC until the operation is completed. In this case, the processor's control circuitry must extend the duration of the execution step for as many clock cycles as needed, until MFC is asserted. We will use the command Wait for MFC to indicate that a given execution step must be extended, if necessary, until a memory operation is completed. When MFC is received, the actions specified in the step are completed, and the processor proceeds to the next step in the execution sequence.

Step 1 of the execution sequence of any instruction involves fetching the instruction from the memory. Therefore, it must include a Wait for MFC command, as follows:

Memory address ← [PC], Read memory, Wait for MFC,
IR ← Memory data, PC ← [PC] + 4

The Wait for MFC command is also needed in step 4 of Load and Store instructions in Figures 5.13 and 5.14. Most of the time, the requested information is found in the cache, so the MFC signal is generated quickly, and the step is completed in one clock cycle. When an access involves the main memory, the MFC response is delayed, and the step is extended to several clock cycles.

5.5 CONTROL SIGNALS

The operation of the processor's hardware components is governed by *control signals*. These signals determine which multiplexer input is selected, what operation is performed by the ALU, and so on. In this section we discuss the signals needed to control the operation of the components in Figures 5.8 to 5.10.

It is instructive to begin by recalling how data flow through the four stages of the datapath, as described in Section 5.3.3. In each clock cycle, the results of the actions that take place in one stage are stored in inter-stage registers, to be available for use by the next stage in the next clock cycle. Since data are transferred from one stage to the next in every clock cycle, inter-stage registers are always enabled. This is the case for registers RA, RB, RZ, RY, RM, and PC-Temp. The contents of the other registers, namely, the PC, the IR, and the register file, must not be changed in every clock cycle. New data are loaded into these registers only when called for in a particular processing step. They must be enabled only at those times.

The role of the multiplexers is to select the data to be operated on in any given stage. For example, MuxB in stage 3 of Figure 5.8 selects the immediate field in the IR for instructions that use an immediate source operand. It also selects that field for instructions that use immediate data as an offset when computing the effective address of a memory operand. Otherwise, it selects register RB. The data selected by the multiplexer are used by the ALU. Examination of Figures 5.11, 5.13, and 5.14 shows that the ALU is used only in step 3, and hence the selection made by MuxB matters only during that step. To simplify the required control circuit, the same selection can be maintained in all execution steps. A similar observation can be made about MuxY. However, MuxMA in Figure 5.9 must change its selection in different execution steps. It selects the PC as the source of the memory address during step 1, when a new instruction is being fetched. During step 4 of Load and Store instructions, it selects register RZ, which contains the effective address of the memory operand.

Figures 5.18, 5.19, and 5.20 show the required control signals. The register file has three 5-bit address inputs, allowing access to 32 general-purpose registers. Two of these inputs, Address A and Address B, determine which registers are to be read. They are connected to fields IR_{31-27} and IR_{26-22} in the instruction register. The third address input, Address C, selects the destination register, into which the input data at port C are to be written. Multiplexer MuxC selects the source of that address. We have assumed that three-register instructions use bits IR_{21-17} and other instructions use IR_{26-22} to specify the destination register, as in Figure 5.12. The third input of the multiplexer is the address of the link register used in subroutine linkage instructions. New data are loaded into the selected register only when the control signal RF_write is asserted.

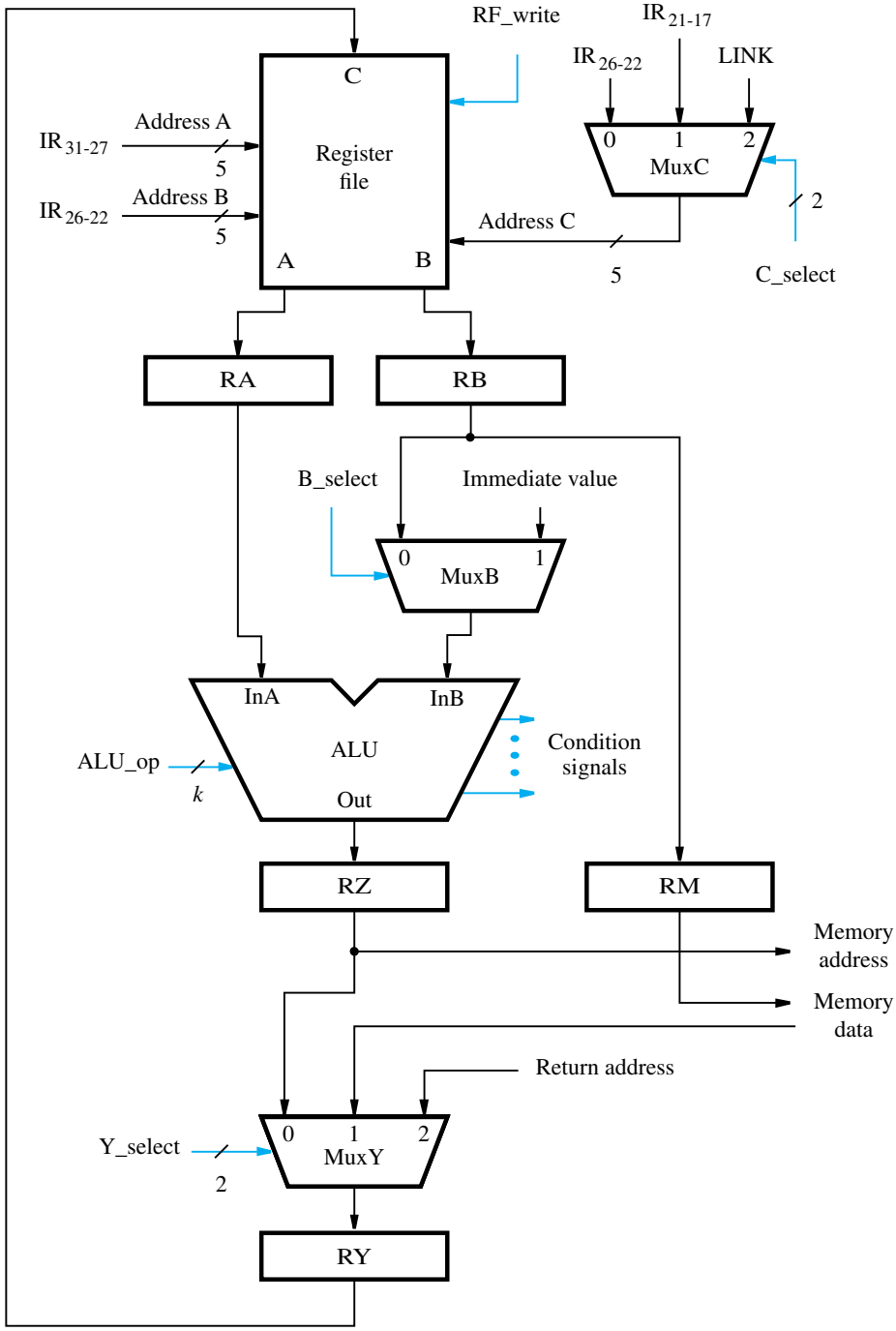


Figure 5.18 Control signals for the datapath.

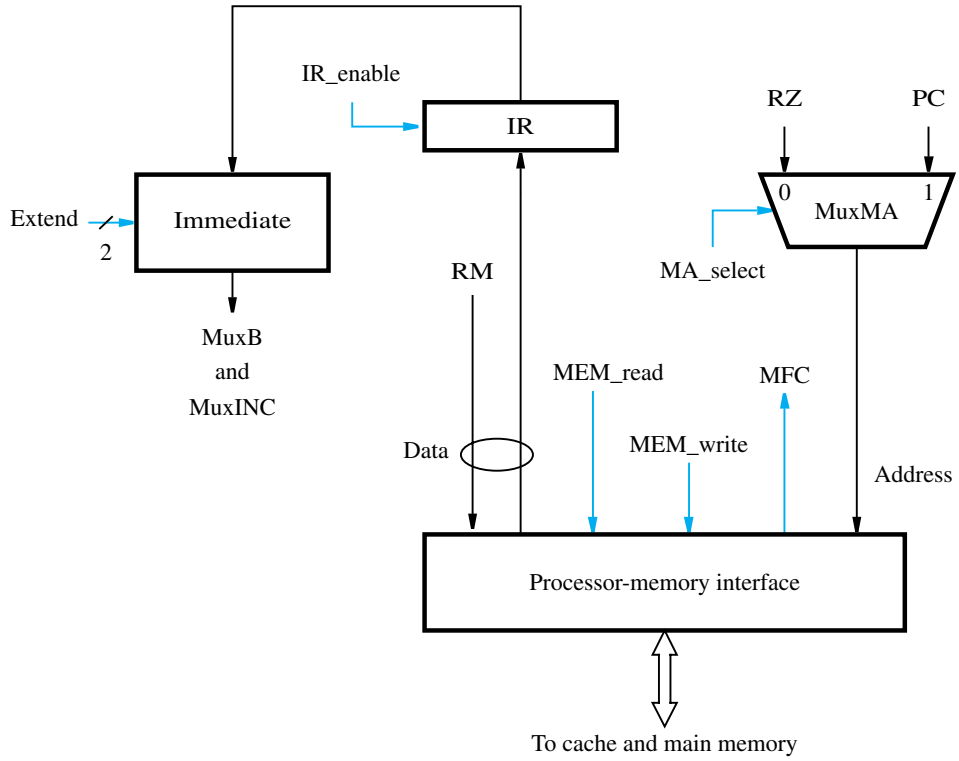


Figure 5.19 Processor-memory interface and IR control signals.

Multiplexers are controlled by signals that select which input data appear at the multiplexer's output. For example, when B_select is equal to 0, MuxB selects the contents of register RB to be available at input InB of the ALU. Note that two bits are needed to control MuxC and MuxY, because each multiplexer selects one of three inputs.

The operation performed by the ALU is determined by a k-bit control code, ALU_op, which can specify up to 2^k distinct operations, such as Add, Subtract, AND, OR, and XOR. When an instruction calls for two values to be compared, a comparator performs the comparison specified, as mentioned earlier. The comparator generates condition signals that indicate the result of the comparison. These signals are examined by the control circuitry during the execution of conditional branch instructions to determine whether the branch condition is true or false.

The interface between the processor and the memory and the control signals associated with the instruction register are presented in Figure 5.19. Two signals, MEM_read and MEM_write are used to initiate a memory Read or a memory Write operation. When the requested operation has been completed, the interface asserts the MFC signal. The instruction register has a control signal, IR_enable, which enables a new instruction to be loaded into the register. During a fetch step, it must be activated only after the MFC signal is asserted.

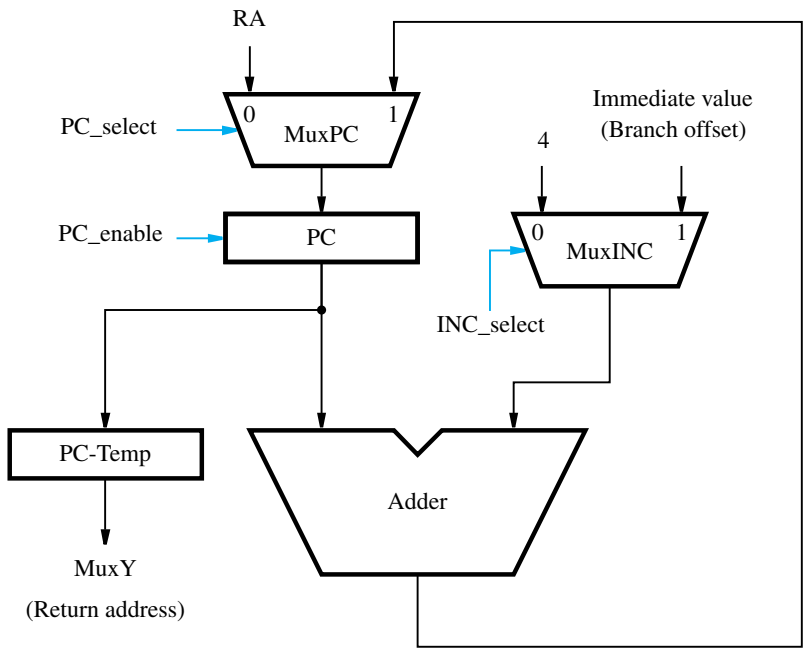


Figure 5.20 Control signals for the instruction address generator.

We have assumed that the Immediate block handles three possible formats for the immediate value: a sign-extended 16-bit value, a zero-extended 16-bit value, and a 26-bit value that is handled in a special way (see Problem 5.14). Hence, its control signal, Extend, comprises two bits.

The signals that control the operation of the instruction address generator are shown in Figure 5.20. The INC_select signal selects the value to be added to the PC, either the constant 4 or the branch offset specified in the instruction. The PC_select signal selects either the updated address or the contents of register RA to be loaded into the PC when the PC_enable control signal is activated.

5.6 HARDWIRED CONTROL

Previous sections described the actions needed to fetch and execute instructions. We now examine how the processor generates the control signals that cause these actions to take place in the correct sequence and at the right time. There are two basic approaches: hardwired control and microprogrammed control. Hardwired control is discussed in this section.

An instruction is executed in a sequence of steps, where each step requires one clock cycle. Hence, a step counter may be used to keep track of the progress of execution. Several

actions are performed in each step, depending on the instruction being executed. In some cases, such as for branch instructions, the actions taken depend on tests applied to the result of a computation or a comparison operation. External signals, such as interrupt requests, may also influence the actions to be performed. Thus, the setting of the control signals depends on:

- Contents of the step counter
- Contents of the instruction register
- The result of a computation or a comparison operation
- External input signals, such as interrupt requests

The circuitry that generates the control signals may be organized as shown in Figure 5.21. The instruction decoder interprets the OP-code and addressing mode information in the IR and sets to 1 the corresponding INS*i* output. During each clock cycle, one of the outputs T1 to T5 of the step counter is set to 1 to indicate which of the five steps involved in fetching and executing instructions is being carried out. Since all instructions are completed in five steps, a modulo-5 counter may be used. The control signal generator is a combinational circuit that produces the necessary control signals based on all its inputs. The required settings of the control signals can be determined from the action sequences that implement each of the instructions represented by the signals INS1 to INS*m*.

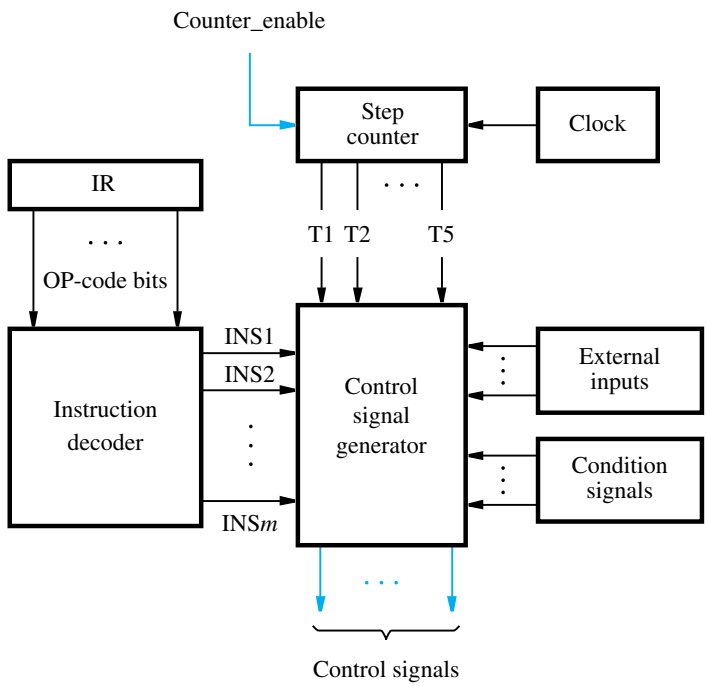


Figure 5.21 Generation of the control signals.

As an example, consider step 1 in the instruction execution process. This is the step in which a new instruction is fetched from the memory. It is identified by signal T1 being asserted. During that clock period, the MA_select signal in Figure 5.19 is set to 1 to select the PC as the source of the memory address, and MEM_read is activated to initiate a memory Read operation. The data received from the memory are loaded into the IR by activating IR_enable when the memory's response signal, MFC, is asserted. At the same time, the PC is incremented by 4, by setting the INC_select signal in Figure 5.20 to 0 and PC_select to 1. The PC_enable signal is activated to cause the new value to be loaded into the PC at the positive edge of the clock marking the end of step T1.

5.6.1 DATAPATH CONTROL SIGNALS

Instructions that handle data include Load, Store, and all computational instructions. They perform various data movement and manipulation operations using the processor's datapath, whose control signals are shown in Figures 5.18 and 5.19. Once an instruction is loaded into the IR, the instruction decoder interprets its contents to determine the actions needed. At the same time, the source registers are read and their contents become available at the A and B outputs of the register file. As mentioned earlier, inter-stage registers RA, RB, RZ, RM, and RY are always enabled. This means that data flow automatically from one datapath stage to the next on every active edge of the clock signal.

The desired setting of various control signals can be determined by examining the actions taken in each execution step of every instruction. For example, the RF_write signal is set to 1 in step T5 during execution of an instruction that writes data into the register file. It may be generated by the logic expression

$$\text{RF\_write} = \text{T5} \cdot (\text{ALU} + \text{Load} + \text{Call})$$

where ALU stands for all instructions that perform arithmetic or logic operations, Load stands for all Load instructions, and Call stands for all subroutine-call and software-interrupt instructions. The RF_write signal is a function of both the instruction and the timing signals. But, as mentioned earlier, the setting of some of the multiplexers need not change from one timing step to another. In this case, the multiplexer's select signal can be implemented as a function of the instruction only. For example,

$$\text{B\_select} = \text{Immediate}$$

where Immediate stands for all instructions that use an immediate value in the IR. We encourage the reader to examine other control signals and derive the appropriate logic expressions for them, based on the execution steps of various instructions.

5.6.2 DEALING WITH MEMORY DELAY

The timing signals T1 to T5 are asserted in sequence as the step counter is advanced. Most of the time, the step counter is incremented at the end of every clock cycle. However, a step

in which a MEM_read or a MEM_write command is issued does not end until the MFC signal is asserted, indicating that the requested memory operation has been completed.

To extend the duration of an execution step to more than one clock cycle, we need to disable the step counter. Assume that the counter is incremented when enabled by a control signal called Counter_enable. Let the need to wait for a memory operation to be completed be indicated by a control signal called WMFC, which is activated during any execution step in which the Wait for MFC command is issued. Counter_enable should be set to 1 in any step in which WMFC is not asserted. Otherwise, it should be set to 1 when MFC is asserted. This means that

$$\text{Counter\_enable} = \overline{\text{WMFC}} + \text{MFC}$$

A new value is loaded into the PC at the end of any clock cycle in which the PC_enable signal in Figure 5.20 is activated. We must ensure that the PC is incremented only once when an execution step is extended for more than one clock cycle. Hence, when fetching an instruction, the PC should be enabled only when MFC is received. It is also enabled in step 3 of instructions that cause branching. Let BR denote all instructions in this group. Then, PC_enable may be realized as

$$\text{PC\_enable} = T1 \cdot \text{MFC} + T3 \cdot \text{BR}$$

5.7 CISC-STYLE PROCESSORS

We saw in the previous sections that a RISC-style instruction set is conducive to a multi-stage implementation of the processor. All instructions can be executed in a uniform manner using the same five-stage hardware. As a result, the hardware is simple and well suited to pipelined operation. Also, the control signals are easy to generate.

CISC-style instruction sets are more complex because they allow much greater flexibility in accessing instruction operands. Unlike RISC-style instruction sets, where only Load and Store instructions access data in the memory, CISC instructions can operate directly on memory operands. Also, they are not restricted to one word in length. An instruction may use several words to specify operand addresses and the actions to be performed, as explained in Section 2.10. Therefore, CISC-style instructions require a different organization of the processor hardware.

Figure 5.22 shows a possible processor organization. The main difference between this organization and the five-stage structure discussed earlier is that the Interconnect block, which provides interconnections among other blocks, does not prescribe any particular structure or pattern of data flow. It provides paths that make it possible to transfer data between any two components, as needed to implement instructions. The multi-stage structure of Figure 5.8 uses inter-stage registers, such as RZ and RY. These are not needed in the organization of Figure 5.22. Instead, some registers are needed to hold intermediate results during instruction execution. The temporary registers block in the figure is provided for this purpose. It includes two temporary registers, Temp1 and Temp2. The need for these registers will become apparent from the examples given later.

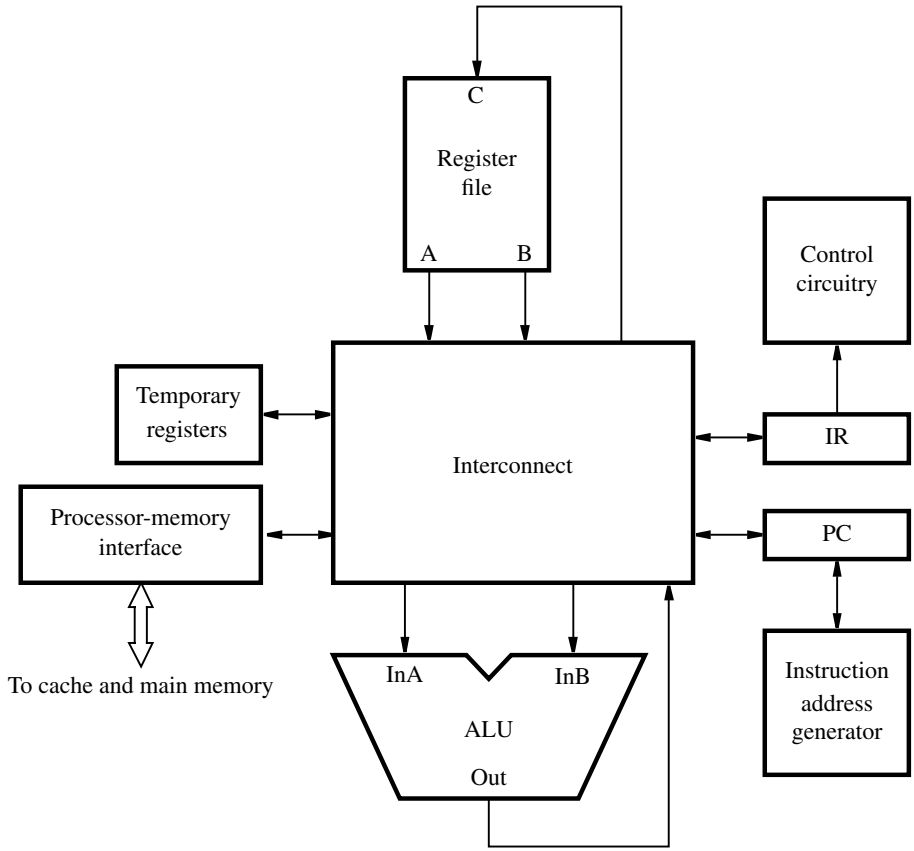


Figure 5.22 Organization of a CISC-style processor.

A traditional approach to the implementation of the Interconnect is to use buses. A *bus* consists of a set of lines to which several devices may be connected, enabling data to be transferred from any one device to any other. A logic gate that sends a signal over a bus line is called a *bus driver*. Since all devices connected to the bus have the ability to send data, we must ensure that only one of them is driving the bus at any given time. For this reason, the bus driver is a special type of logic gate called a *tri-state gate*. It has a control input that turns it on or off. When turned on, the gate places a logic signal of 0 or 1 on the bus, according to the value of its input. When turned off, the gate is electrically disconnected from the bus, as explained in Appendix A.

Figure 5.23 shows how a flip-flop that forms one bit of a data register can be connected to a bus line. There are two control signals, R_{in} and R_{out}. When R_{in} is equal to 1 the multiplexer selects the data on the bus line to be loaded into the flip-flop. Setting R_{in} to 0 causes the flip-flop to maintain its present value. The output of the flip-flop is connected to the bus line through a tri-state gate, which is turned on when R_{out} is asserted. At other times, the tri-state gate is turned off, allowing other components to drive the bus line.

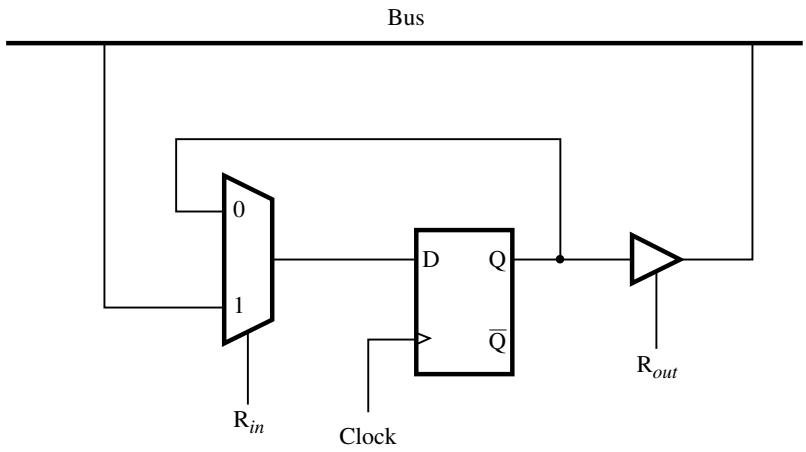


Figure 5.23 Input and output gating for one register bit.

5.7.1 AN INTERCONNECT USING BUSES

The Interconnect in Figure 5.22 may be implemented using one or more buses. Figure 5.24 shows a three-bus implementation. All registers are assumed to be edge-triggered. That is, when a register is enabled, data are loaded into it on the active edge of the clock at the end of the clock period. Addresses for the three ports of the register file are provided by the Control block. These connections are not shown to keep the figure simple. Also not shown is the Immediate block through which the IR is connected to bus B. This is the circuit that extends an immediate operand in the IR to 32 bits.

Consider the two-operand instruction

Add R5, R6

which performs the operation

$$R5 \leftarrow [R5] + [R6]$$

Fetching and executing this instruction using the hardware in Figure 5.24 can be performed in three steps, as shown in Figure 5.25. Each step, except for the step involving access to the memory, is completed in one clock cycle. In step 1, bus B is used to send the contents of the PC to the processor-memory interface, which sends them on the memory address lines and initiates a memory Read operation. The data received from the memory, which represent an instruction to be executed, are sent to the IR over bus C. The command Wait for MFC is included to accommodate the possibility that memory access may take more than one clock cycle, as explained in Section 5.4.2. The instruction is decoded in step 2 and the control circuitry begins reading the source registers, R5 and R6. However, the contents of the registers do not become available at the A and B outputs of the register file until step 3. They are sent to the ALU using buses A and B. The ALU performs the addition operation, and the sum is sent back to the ALU over bus C, to be written into register R5 at the end of the clock cycle.

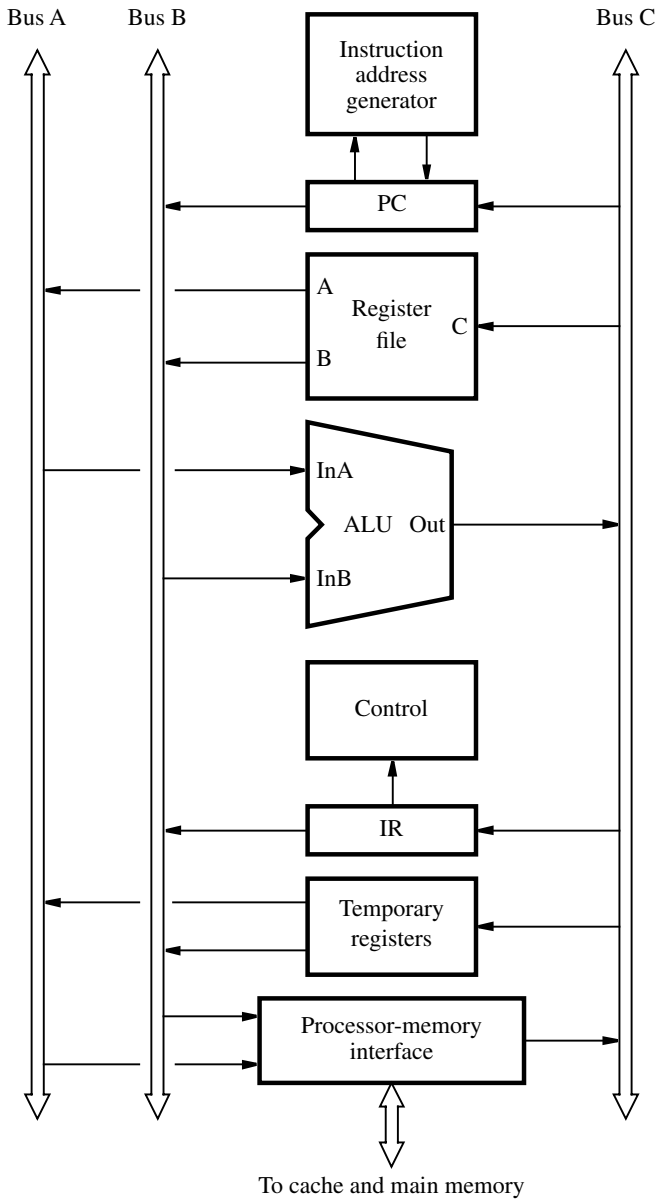


Figure 5.24 Three-bus CISC-style processor organization.

Note that reading the source registers is completed in step 2 in Figure 5.11. In that case, the action of reading the registers proceeds in parallel with the action of decoding the instruction, because the location of the bit fields containing register addresses in a RISC-style instruction is known. Since CISC-style instructions do not always use the same

Step	Action
1	Memory address ← [PC], Read memory, Wait for MFC, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction
3	R5 ← [R5] + [R6]

Figure 5.25 Sequence of actions needed to fetch and execute the instruction: Add R5, R6.

instruction fields to specify register addresses, the action of reading the source registers does not begin until the instruction has been at least partially decoded. Hence, it may not be possible to complete reading the source registers in step 2.

Next, consider the instruction

And X(R7), R9

which performs the logical AND operation on the contents of register R9 and memory location X + [R7] and stores the result back in the same memory location. Assume that the index offset X is a 32-bit value given as the second word of the instruction. To execute this instruction, it is necessary to access the memory four times. First, the OP-code word is fetched. Then, when the instruction decoding circuit recognizes the Index addressing mode, the index offset X is fetched. Next, the memory operand is fetched and the AND operation is performed. Finally, the result is stored back into the memory.

Figure 5.26 gives the steps needed to execute the instruction. After decoding the instruction in step 2, the second word of the instruction is read in step 3. The data received,

Step	Action
1	Memory address ← [PC], Read memory, Wait for MFC, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction
3	Memory address ← [PC], Read memory, Wait for MFC, Temp1 ← Memory data, PC ← [PC] + 4
4	Temp2 ← [Temp1] + [R7]
5	Memory address ← [Temp2], Read memory, Wait for MFC, Temp1 ← Memory data
6	Temp1 ← [Temp1] AND [R9]
7	Memory address ← [Temp2], Memory data ← [Temp1], Write memory, Wait for MFC

Figure 5.26 Sequence of actions needed to fetch and execute the instruction: And X(R7), R9.

which represent the offset X, are stored temporarily in register Temp1, to be used in the next step for computing the effective address of the memory operand. In step 4, the contents of registers Temp1 and R7 are sent to the ALU inputs over buses A and B. The effective address is computed and placed into register Temp2, then used to read the operand in step 5. Register Temp1 is used again during step 5, this time to hold the data operand received from the memory. The computation is performed in step 6, and the result is placed back in register Temp1. In the final step, the result is sent to be stored in the memory at the operand address, which is still available in register Temp2.

The two examples in Figures 5.25 and 5.26 illustrate the variability in the number of execution steps in CISC-style instructions. There is no uniform sequence of actions that can be followed for all instructions in the same way as was demonstrated for RISC instructions in Section 5.2.

5.7.2 MICROPROGRAMMED CONTROL

The control signals needed to control the operation of the components in Figures 5.22 and 5.24 can be generated using the hardwired approach described in Section 5.6. But, there is an interesting alternative that was popular in the past, which we describe next.

Control signals are generated for each execution step based on the instruction in the IR. In hardwired control, these signals are generated by circuits that interpret the contents of the IR as well as the timing signals derived from a step counter. Instead of employing such circuits, it is possible to use a "software" approach, in which the desired setting of the control signals in each step is determined by a program stored in a special memory. The control program is called a *microprogram* to distinguish it from the program being executed by the processor. The microprogram is stored on the processor chip in a small and fast memory called the *microprogram memory* or the *control store*.

Suppose that n control signals are needed. Let each control signal be represented by a bit in an n-bit word, which is often referred to as a *control word* or a *microinstruction*. Each bit in that word specifies the setting of the corresponding signal for a particular step in the execution flow. One control word is stored in the microprogram memory for each step in the execution sequence of an instruction. For example, the action of reading an instruction or a data operand from the memory requires use of the MEM_read and WMFC signals introduced in Sections 5.5 and 5.6.2, respectively. These signals are asserted by setting the corresponding bits in the control word to 1 for steps 1, 3, and 5 in Figure 5.26. When a microinstruction is read from the control store, each control signal takes on the value of its corresponding bit.

The sequence of microinstructions corresponding to a given machine instruction constitutes the *microroutine* that implements that instruction. The first two steps in Figures 5.25 and 5.26 specify the actions for fetching and decoding an instruction. They are common to all instructions. The microroutine that is specific to a given machine instruction starts with step 3.

Figure 5.27 depicts a typical organization of the hardware needed for microprogrammed control. It consists of a microinstruction address generator, which generates the address

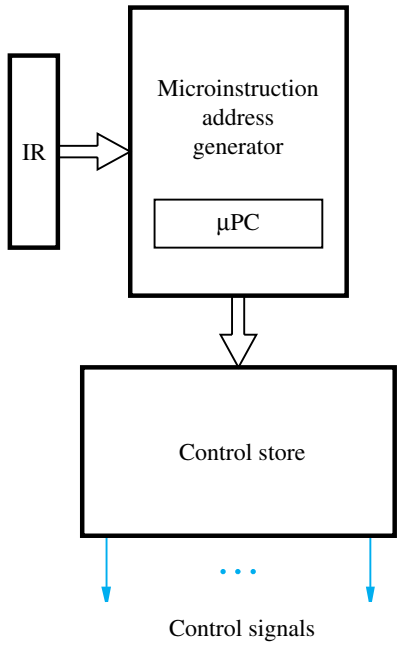


Figure 5.27 Microprogrammed control unit organization.

to be used for reading microinstructions from the control store. The address generator uses a *microprogram counter*, μPC, to keep track of control store addresses when reading microinstructions from successive locations. During step 2 in Figures 5.25 and 5.26, the microinstruction address generator decodes the instruction in the IR to obtain the starting address of the corresponding microroutine and loads that address into the μPC. This is the address that will be used in the following clock cycle to read the control word corresponding to step 3. As execution proceeds, the microinstruction address generator increments the μPC to read microinstructions from successive locations in the control store. One bit in the microinstruction, which we will call End, is used to mark the last microinstruction in a given microroutine. When End is equal to 1, as would be the case in step 3 in Figure 5.25 and step 7 in Figure 5.26, the address generator returns to the microinstruction corresponding to step 1, which causes a new machine instruction to be fetched.

Microprogrammed control can be viewed as having a control processor within the main processor. Microinstructions are fetched and executed much like machine instructions. Their function is to direct the actions of the main processor's hardware components, by indicating which control signals need to be active during each execution step.

Microprogrammed control is simple to implement and provides considerable flexibility in controlling the execution of machine instructions. But, it is slower than hardwired control. Also, the flexibility it provides is not needed in RISC-style processors. As the discussion in this chapter illustrates, the control signals needed to implement RISC-style instructions are

quite simple to generate. Since the cost of logic circuitry is no longer a significant factor, hardwired control has become the preferred choice.

5.8 CONCLUDING REMARKS

This chapter explained the basic structure of a processor and how it executes instructions. Modern processors have a multi-stage organization because this is a structure that is well-suited to pipelined operation. Each stage implements the actions needed in one of the execution steps of an instruction. A five-step sequence in which each step is completed in one clock cycle has been demonstrated. Such an approach is commonly used in processors that have a RISC-style instruction set.

The discussion in this chapter assumed that the execution of one instruction is completed before the next instruction is fetched. Only one of the five hardware stages is used at any given time, as execution moves from one stage to the next in each clock cycle. We will show in the next chapter that it is possible to overlap the execution steps of successive instructions, resulting in much better performance. This leads to a pipelined organization.

5.9 SOLVED PROBLEMS

This section presents some examples of the types of problems that a student may be asked to solve, and shows how such problems can be solved.

Example 5.1

Problem: Figure 5.11 shows an Add instruction being executed in five steps, but no processing actions take place in step 4. If it is desired to eliminate that step, what changes have to be made in the datapath in Figure 5.8 to make this possible?

Solution: Step 4 can be skipped by sending the output of the ALU in Figure 5.8 directly to register RY. This can be accomplished by adding one more input to multiplexer MuxY and connecting that input to the output of the ALU. Thus, the result of a computation at the output of the ALU is loaded into both registers RZ and RY at the end of step 3. For an Add instruction, or any other computational instruction, the register file control signal RF_write can be enabled in step 4 to load the contents of RY into the register file.

Example 5.2

Problem: Assume that all memory access operations are completed in one clock cycle in a processor that has a 1-GHz clock. What is the frequency of memory access operations if Load and Store instructions constitute 20 percent of the dynamic instruction count in a program? (The dynamic count is the number of instruction executions, including the effect of program loops, which may cause some instructions to be executed more than once.) Assume that all instructions are executed in 5 clock cycles.

Solution: There is one memory access to fetch each instruction. Then, 20 percent of the instructions have a second memory access to read or write a memory operand. On average, each instruction has 1.2 memory accesses in 5 clock cycles. Therefore, the frequency of memory accesses is $(1.2/5) \times 10^9$, or 240 million accesses per second.

Example 5.3 **Problem:** Derive the logic expressions for a circuit that compares two unsigned numbers: $X = x_2x_1x_0$ and $Y = y_2y_1y_0$ and generates three outputs: XGY, XEY, and XLY. One of these outputs is set to 1 to indicate that X is greater than, equal to, or less than Y, respectively.

Solution: To compare two unsigned numbers, we need to compare individual bit locations, starting with the most significant bit. If $x_2 = 1$ and $y_2 = 0$, then X is greater than Y. If $x_2 = y_2$, then we need to compare the next lower bit location, and so on. Thus, the logic expressions for the three outputs may be written as follows:

$$XGY = x_2\bar{y}_2 + \overline{(x_2 \oplus y_2)} \cdot (x_1\bar{y}_1 + \overline{(x_1 \oplus y_1)}\, x_0\bar{y}_0)$$

$$XEY = \overline{(x_2 \oplus y_2)} \cdot \overline{(x_1 \oplus y_1)} \cdot \overline{(x_0 \oplus y_0)}$$

$$XLY = \overline{XGY + XEY}$$

Example 5.4 **Problem:** Give the sequence of actions for a Return-from-subroutine instruction in a RISC processor. Assume that the address LINK of the general-purpose register in which the subroutine return address is stored is given in the instruction field connected to address A of the register file (IR_{31-27}).

Solution: Whenever an instruction is loaded into the IR, the contents of the general-purpose register whose address is given in bits IR_{31-27} are read and placed into register RA (see Figure 5.18). Hence, a Return-from-subroutine instruction will cause the contents of register LINK to be read and placed in register RA. Execution proceeds as follows:

1. Memory address ← [PC], Read memory, Wait for MFC, IR ← Memory data, PC ← [PC] + 4
2. Decode instruction, RA ← [LINK]
3. PC ← [RA]
4. No action
5. No action

Example 5.5 **Problem:** A processor has the following interrupt structure. When an interrupt is received, the interrupt return address is saved in a general-purpose register called IRA. The current contents of the processor status register, PS, are saved in a special register called IPS, which is not a general-purpose register. The interrupt-service routine starts at address ILOC.

Assume that the processor checks for interrupts in the last execution step of every instruction. If an interrupt request is present and interrupts are enabled, the request is accepted. Instead of fetching the next instruction, the processor saves the PC and the PS and branches to ILOC. Give a suitable sequence of steps for performing these actions. What additional hardware is needed in Figures 5.18 to 5.20 to support interrupt processing?

Solution: The first two steps of instruction execution, in which an instruction is fetched and decoded, are not needed in the case of an interrupt. They may be skipped, or they would take no action if it is desired to maintain a 5-step sequence. Saving the PC can be done in exactly the same manner as for a subroutine call instruction. Another input to MuxC in Figure 5.18 is needed to which the address of register IRA should be connected. To load the starting address of the interrupt-service routine into the PC, an additional input to MuxPC in Figure 5.20 is needed, to which the value ILOC should be connected. Registers PS and IPS should be connected directly to each other to enable data to be transferred between them. The execution steps required are:

3. PC-Temp ← [PC], PC ← ILOC, IPS ← [PS], Disable interrupts
4. RY ← [PC-Temp]
5. IRA ← [RY]

These actions are reversed by a Return-from-interrupt instruction. See Problem 5.8.

Example 5.6

Problem: Example 5.5 illustrates how the contents of the PC and the PS are saved when an interrupt request is accepted. In order to support interrupt nesting, it is necessary for the interrupt-service routine to save these registers on the processor stack, as described in Section 3.2. To do so, the contents of the PS, which are saved in register IPS at the time the interrupt is accepted, need to be moved to one of the general-purpose registers, from where they can be saved on the stack. Assume that two special instructions

MoveControl R*i*, IPS

and

MoveControl IPS, R*i*

are available to save and restore the contents of IPS, respectively. Suggest changes to the hardware in Figures 5.8 and 5.10 to implement these instructions.

Solution: A possible organization is shown in Figure 5.28. To save the contents of IPS, its output is connected to an additional input on MuxY. When restoring its contents, MuxIPS selects register RA.

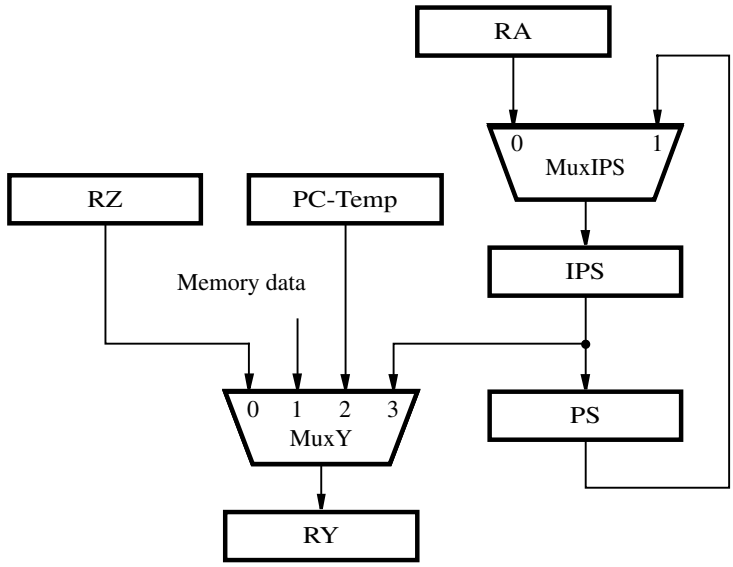


Figure 5.28 Connection of IPS for Example 5.6.

PROBLEMS

5.1 **[M]** The propagation delay through the combinational circuit in Figure 5.2 is 600 ps (picoseconds). The registers have a setup time requirement of 50 ps, and the maximum propagation delay from the clock input to the Q outputs is 70 ps.

(*a*) What is the minimum clock period required for correct operation of this circuit?

(*b*) Assume that the circuit is reorganized into three stages as in Figure 5.3, such that the combinational circuit in each stage has a delay of 200 ps. What is the minimum clock period in this case?

5.2 **[M]** At the time the instruction

```
Load    R6, 1000(R9)
```

is fetched, R6 and R9 contain the values 4200 and 85320, respectively. Memory location 86320 contains 75900. Show the contents of the interstage registers in Figure 5.8 during each of the 5 execution steps of this instruction.

5.3 **[E]** Figure 5.12 shows the bit fields assigned to register addresses for different groups of instructions. Why is it important to use the same field locations for all instructions?

5.4 **[M]** At some point in the execution of a program, registers R4, R6, and R7 contain the values 1000, 7500, and 2500, respectively. Show the contents of registers RA, RB, RZ,

RY, and R6 in Figure 5.8 during steps 3 to 5 as the instruction

Subtract R6, R4, R7

is fetched and executed, and also during step 1 of the instruction that is fetched next.

5.5 **[M]** The instruction

And R4, R4, R8

is stored in location 0x37C00 in the memory. At the time this instruction is fetched, registers R4 and R8 contain the values 0x1000 and 0xB2500, respectively. Give the values in registers PC, R4, RA, RM, RZ, and RY of Figures 5.8 and 5.10 in each clock cycle as this instruction is executed, and also in the first clock cycle of the next instruction.

5.6 **[D]** Modify the expressions given in Example 5.3 to compare two, 4-bit, signed numbers in 2's-complement representation.

5.7 **[E]** The subroutine-call instructions described in Chapter 2 always use the same general-purpose register, LINK, to store the return address. Hence, the return register address is not included in the instruction. However, the address LINK is included in bits IR_{31-27} of subroutine-return instructions (see Section 5.4.1 and Example 5.4). Why are the two instructions treated differently?

5.8 **[M]** Give the execution sequence for the Return-from-interrupt instruction for a processor that has the interrupt structure given in Example 5.5. Assume that the address of register IRA is given in bits IR_{31-27} of the instruction.

5.9 **[D]** Consider an instruction set in which instruction encoding is such that register addresses for different instructions are not always in the same bit locations. What effect would that have on the execution steps of the instructions? What would you do to maintain a five-step execution sequence in this case? Assume the same hardware structure as in Figure 5.8.

5.10 **[M]** Assume that immediate operands occupy bits IR_{21-6} of the instruction. The immediate value is sign-extended to 32 bits in arithmetic instructions, such as Add, and padded with zeros in logic instructions, such as Or. Design a suitable implementation for the Immediate block in Figure 5.9.

5.11 **[M]** A RISC processor that uses the five-step sequence in Figure 5.4 is driven by a 1-GHz clock. Instruction statistics in a large program are as follows:

Branch	20%
Load	20%
Store	10%
Computational instructions	50%

Estimate the rate of instruction execution in each of the following cases:

(*a*) Access to the memory is always completed in 1 clock cycle.

(*b*) 90% of instruction fetch operations are completed in one clock cycle and 10% are completed in 4 clock cycles. On average, access to the data operands of a Load or Store instruction is completed in 3 clock cycles.

5.12 **[E]** The execution of computational instructions follows the pattern given in Figure 5.11 for the Add instruction, in which no processing actions are performed in step 4. Consider a program that has the instruction statistics given in Problem 5.11. Estimate the increase in instruction execution rate if this step is eliminated, assuming that all execution steps are completed in one clock cycle.

5.13 **[D]** Figure 5.16 shows that step 3 of a conditional branch instruction may result in a new value being loaded into the PC. In pipelined processors, it is desirable to determine the outcome of a conditional branch as early as possible in the execution sequence. What hardware changes would be needed to make it possible to move the actions in step 3 to step 2? Examine all the actions involved in these two steps and show which actions can be carried out in parallel and which must be completed sequentially.

5.14 **[M]** The instructions of a computer are encoded as shown in Figure 5.12. When an immediate value is given in an instruction, it has to be extended to a 32-bit value. Assume that the immediate value is used in three different ways:

(*a*) A 16-bit value is sign-extended for use in arithmetic operations.

(*b*) A 16-bit value is padded with zeros to the left for use in logic operations.

(*c*) A 26-bit value is padded with 2 zeros to the right and the 4 high-order bits of the PC are appended to the left for use in subroutine-call instructions.

Show an implementation for the Immediate block in Figure 5.19 that would perform the required extensions.

5.15 **[E]** We have seen how all RISC-style instructions can be executed using the steps in Figure 5.4 on the multi-stage hardware of Figure 5.8. Autoincrement and Autodecrement addressing modes are not included in RISC-style instruction sets. Explain why the instruction

Load R3, (R5)+

cannot be executed on the hardware in Figure 5.8.

5.16 **[E]** Section 2.9 describes how the two instructions Or and OrHigh can be used to load a 32-bit value into a register. What additional functionality is needed in the processor's datapath to implement the OrHigh instruction? Give the sequence of actions needed to fetch and execute the instruction.

5.17 **[E]** During step 1 of instruction processing, a memory Read operation is started to fetch an instruction at location 0x46000. However, as the instruction is not found in the cache, the Read operation is delayed, and the MFC signal does not become active until the fourth clock cycle. Assume that the delay is handled as described in Section 5.6.2. Show the contents of the PC during each of the four clock cycles of step 1, and also during step 2.

5.18 **[M]** Give the sequence of steps needed to fetch and execute the two special instructions

MoveControl R*i*, IPS

and

MoveControl IPS, R*i*

used in Example 5.6.

5.19 **[D]** What are the essential differences between the hardware structures in Figures 5.8 and 5.22? Illustrate your answer by identifying the difficulties that would be encountered if one attempts to execute the instruction

Subtract LOC, R5

on the hardware in Figure 5.8. This instruction performs the operation

$$LOC \leftarrow [LOC] - [R5]$$

where LOC is a memory location whose address is given as the second word of a two-word instruction.

5.20 **[M]** Consider the actions needed to execute the instructions given in Section 5.4.1. Derive the logic expressions to generate the signals C_select, MA_select, and Y_select in Figures 5.18 and 5.19 for these instructions.

5.21 **[E]** Why is it necessary to include both WMFC and MFC in the logic expression for Counter_enable given in Section 5.6.2?

5.22 **[E]** Explain what would happen if the MFC variable is omitted from the expression for PC_enable given in Section 5.6.2.

5.23 **[M]** Derive the logic expressions to generate the signals PC_select and INC_select shown in Figure 5.20, taking into account the actions needed when executing the following instructions:

Branch: All branch instructions, with a 16-bit branch offset given in the instruction

Call_register: A subroutine-call instruction with the subroutine address given in a general-purpose register

Other: All other instructions that do not involve branching

5.24 **[M]** A microprogrammed processor has the following parameters. Generating the starting address of the microroutine of an instruction takes 2.1 ns, and reading a microinstruction from the control store takes 1.5 ns. Performing an operation in the ALU requires a maximum of 2.2 ns, and access to the cache memory requires 1.7 ns. Assume that all instructions and data are in the cache.

(a) Determine the minimum time needed for each of the steps in Figure 5.26.

(b) Ignoring all other delays, what is the minimum clock cycle that can be used for this processor?

5.25 **[M]** Give the sequence of steps needed to fetch and execute the instruction

Load R3, (R5)+

on the processor of Figure 5.24. Assume 32-bit operands.

5.26 **[M]** Consider a CISC-style processor that saves the return address of a subroutine on the processor stack instead of in the predefined register LINK. Give the sequence of actions needed to execute a Call_Register instruction on the processor of Figure 5.24.

This page intentionally left blank

chapter

6

PIPELINING

CHAPTER OBJECTIVES

In this chapter you will learn about:

- Pipelining as a means for improving performance by overlapping the execution of machine instructions
- Hazards that limit performance gains in pipelined processors and means for mitigating their effect
- Hardware and software implications of pipelining
- Influence of pipelining on instruction set design
- Superscalar processors

Chapter 5 introduced the organization of a processor for executing instructions one at a time. In this chapter, we discuss the concept of pipelining, which overlaps the execution of successive instructions to achieve high performance. We begin by explaining the basics of pipelining and how it can lead to improved performance. Then we examine hazards that cause performance degradation and techniques to alleviate their effect on performance. We discuss the role of *optimizing compilers*, which rearrange the sequence of instructions to maximize the benefits of pipelined execution. For further performance improvement, we also consider replicating hardware units in a *superscalar* processor so that multiple pipelines can operate concurrently.

6.1 BASIC CONCEPT—THE IDEAL CASE

The speed of execution of programs is influenced by many factors. One way to improve performance is to use faster circuit technology to implement the processor and the main memory. Another possibility is to arrange the hardware so that more than one operation can be performed at the same time. In this way, the number of operations performed per second is increased, even though the time needed to perform any one operation is not changed.

Pipelining is a particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly-line operation. Readers are undoubtedly familiar with the assembly line used in automobile manufacturing. The first station in an assembly line may prepare the automobile chassis, the next station adds the body, the next one installs the engine, and so on. While one group of workers is installing the engine on one automobile, another group is fitting a body on the chassis of a second automobile, and yet another group is preparing a new chassis for a third automobile. Although it may take hours or days to complete one automobile, the assembly-line operation makes it possible to have a new automobile rolling off the end of the assembly line every few minutes.

Consider how the idea of pipelining can be used in a computer. The five-stage processor organization in Figure 5.7 and the corresponding datapath in Figure 5.8 allow instructions to be fetched and executed one at a time. It takes five clock cycles to complete the execution of each instruction. Rather than wait until each instruction is completed, instructions can be fetched and executed in a pipelined manner, as shown in Figure 6.1. The five stages corresponding to those in Figure 5.7 are labeled as Fetch, Decode, Compute, Memory, and Write. Instruction I_j is fetched in the first cycle and moves through the remaining stages in the following cycles. In the second cycle, instruction I_{j+1} is fetched while instruction I_j is in the Decode stage where its operands are also read from the register file. In the third cycle, instruction I_{j+2} is fetched while instruction I_{j+1} is in the Decode stage and instruction I_j is in the Compute stage where an arithmetic or logic operation is performed on its operands. Ideally, this overlapping pattern of execution would be possible for all instructions. Although any one instruction takes five cycles to complete its execution, instructions are completed at the rate of one per cycle.

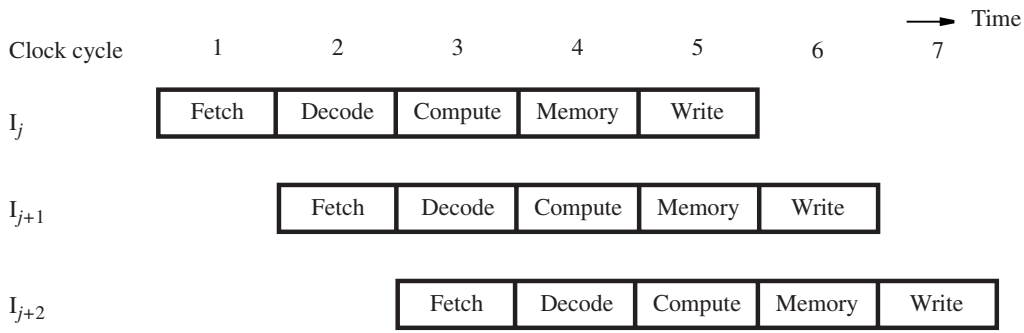


Figure 6.1 Pipelined execution—the ideal case.

6.2 PIPELINE ORGANIZATION

Figure 6.2 indicates how the five-stage organization in Figures 5.7 and 5.8 can be pipelined. In the first stage of the pipeline, the program counter (PC) is used to fetch a new instruction. As other instructions are fetched, execution proceeds through successive stages. At any given time, each stage of the pipeline is processing a different instruction. Information such as register addresses, immediate data, and the operations to be performed must be carried through the pipeline as each instruction proceeds from one stage to the next. This information is held in *interstage buffers*. These include registers RA, RB, RM, RY, and RZ in Figure 5.8, the IR and PC-Temp registers in Figures 5.9 and 5.10, and additional storage. The interstage buffers are used as follows:

- Interstage buffer B1 feeds the Decode stage with a newly-fetched instruction.
- Interstage buffer B2 feeds the Compute stage with the two operands read from the register file, the source/destination register identifiers, the immediate value derived from the instruction, the incremented PC value used as the return address for a subroutine call, and the settings of control signals determined by the instruction decoder. The settings for control signals move through the pipeline to determine the ALU operation, the memory operation, and a possible write into the register file.
- Interstage buffer B3 holds the result of the ALU operation, which may be data to be written into the register file or an address that feeds the Memory stage. In the case of a write access to memory, buffer B3 holds the data to be written. These data were read from the register file in the Decode stage. The buffer also holds the incremented PC value passed from the previous stage, in case it is needed as the return address for a subroutine-call instruction.
- Interstage buffer B4 feeds the Write stage with a value to be written into the register file. This value may be the ALU result from the Compute stage, the result of the Memory access stage, or the incremented PC value that is used as the return address for a subroutine-call instruction.

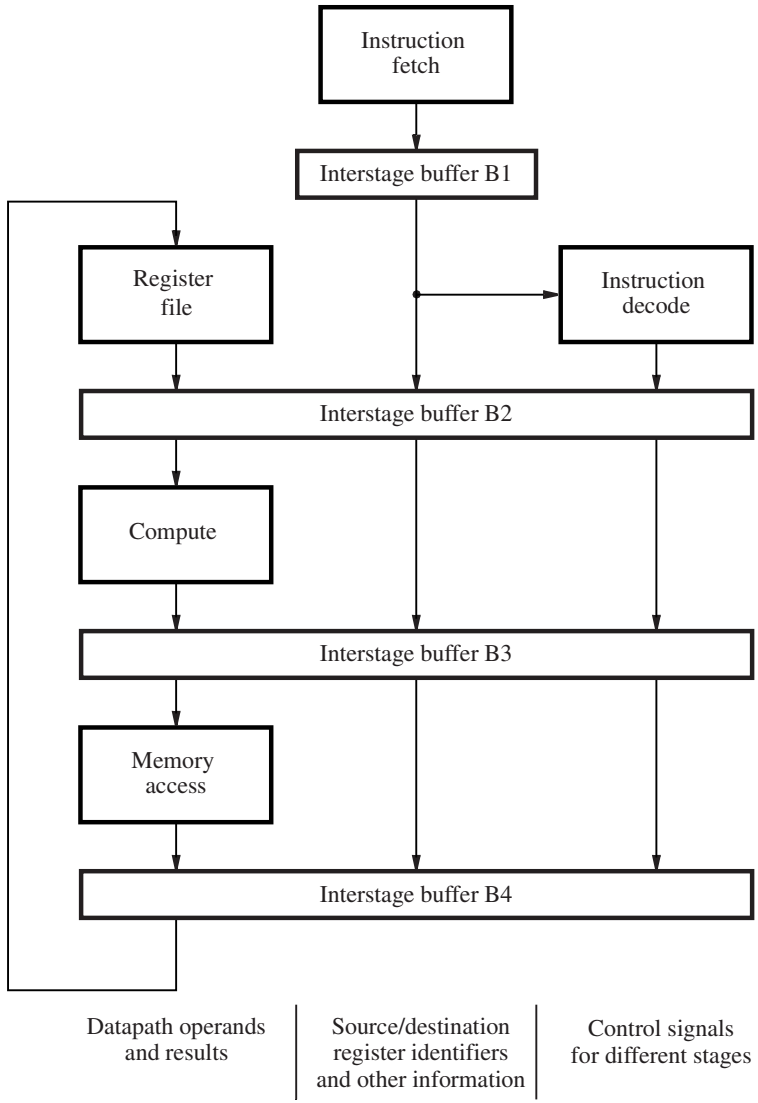


Figure 6.2 A five-stage pipeline.

6.3 PIPELINING ISSUES

Figure 6.1 depicts the ideal overlap of three successive instructions. But, there are times when it is not possible to have a new instruction enter the pipeline in every cycle. Consider the case of two instructions, I_j and I_{j+1}, where the destination register for instruction I_j is a source register for instruction I_{j+1}. The result of instruction I_j is not written into the

register file until cycle 5, but it is needed earlier in cycle 3 when the source operand is read for instruction I_{j+1}. If execution proceeds as shown in Figure 6.1, the result of instruction I_{j+1} would be incorrect because the arithmetic operation would be performed using the old value of the register in question. To obtain the correct result, it is necessary to wait until the new value is written into the register by instruction I_j. Hence, instruction I_{j+1} cannot read its operand until cycle 6, which means it must be *stalled* in the Decode stage for three cycles. While instruction I_{j+1} is stalled, instruction I_{j+2} and all subsequent instructions are similarly delayed. New instructions cannot enter the pipeline, and the total execution time is increased.

Any condition that causes the pipeline to stall is called a *hazard*. We have just described an example of a *data hazard*, where the value of a source operand of an instruction is not available when needed. Other hazards arise from memory delays, branch instructions, and resource limitations. The next several sections describe these hazards in more detail, along with techniques to mitigate their impact on performance.

6.4 DATA DEPENDENCIES

Consider the two instructions in Figure 6.3:

```
Add       R2, R3, #100
Subtract  R9, R2, #30
```

The destination register R2 for the Add instruction is a source register for the Subtract instruction. There is a *data dependency* between these two instructions, because register R2 carries data from the first instruction to the second. Pipelined execution of these two instructions is depicted in Figure 6.3. The Subtract instruction is stalled for three cycles to delay reading register R2 until cycle 6 when the new value becomes available.

We now explain the stall in more detail. The control circuit must first recognize the data dependency when it decodes the Subtract instruction in cycle 3 by comparing its source register identifier from interstage buffer B1 with the destination register identifier of the Add instruction that is held in interstage buffer B2. Then, the Subtract instruction must be held in interstage buffer B1 during cycles 3 to 5. Meanwhile, the Add instruction proceeds through the remaining pipeline stages. In cycles 3 to 5, as the Add instruction moves ahead, control

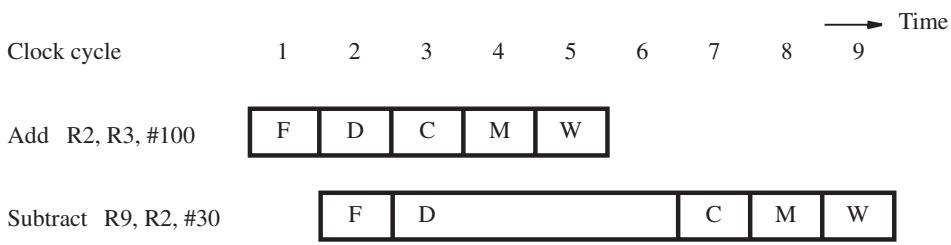


Figure 6.3 Pipeline stall due to data dependency.

signals can be set in interstage buffer B2 for an implicit NOP (No-operation) instruction that does not modify the memory or the register file. Each NOP creates one clock cycle of idle time, called a *bubble*, as it passes through the Compute, Memory, and Write stages to the end of the pipeline.

6.4.1 OPERAND FORWARDING

Pipeline stalls due to data dependencies can be alleviated through the use of *operand forwarding*. Consider the pair of instructions discussed above, where the pipeline is stalled for three cycles to enable the Subtract instruction to use the new value in register R2. The desired value is actually available at the end of cycle 3, when the ALU completes the operation for the Add instruction. This value is loaded into register RZ in Figure 5.8, which is a part of interstage buffer B3. Rather than stall the Subtract instruction, the hardware can *forward* the value from register RZ to where it is needed in cycle 4, which is the ALU input. Figure 6.4 shows pipelined execution when forwarding is implemented. The arrow shows that the ALU result from cycle 3 is used as an input to the ALU in cycle 4.

Figure 6.5 shows the modification needed in the datapath of Figure 5.8 to make this forwarding possible. A new multiplexer, MuxA, is inserted before input InA of the ALU, and the existing multiplexer MuxB is expanded with another input. The multiplexers select either a value read from the register file in the normal manner, or the value available in register RZ.

Forwarding can also be extended to a result in register RY in Figure 5.8. This would handle a data dependency such as the one involving register R2 in the following sequence of instructions:

```
Add       R2, R3, #100
Or        R4, R5, R6
Subtract  R9, R2, #30
```

When the Subtract instruction is in the Compute stage of the pipeline, the Or instruction is in the Memory stage (where no operation is performed), and the Add instruction is in the Write stage. The new value of register R2 generated by the Add instruction is now in register RY. Forwarding this value from register RY to ALU input InA makes it possible

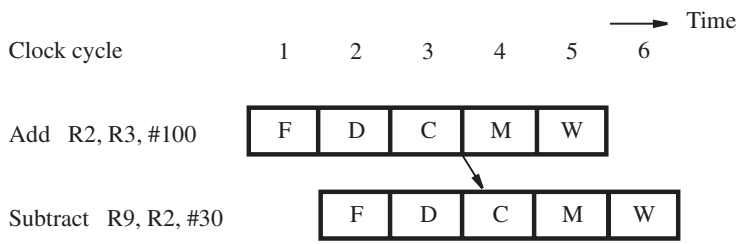


Figure 6.4 Avoiding a stall by using operand forwarding.

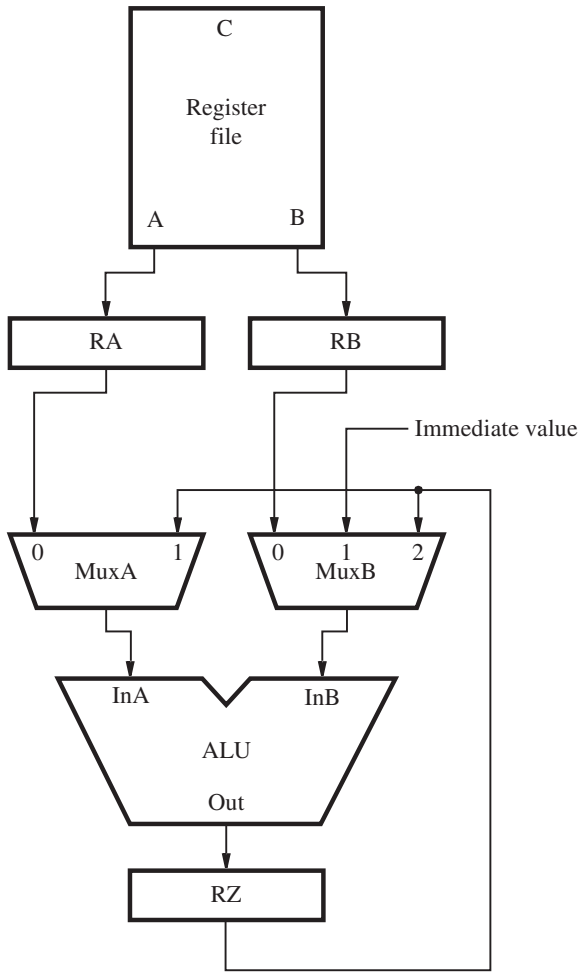


Figure 6.5 Modification of the datapath of Figure 5.8 to support data forwarding from register RZ to the ALU inputs.

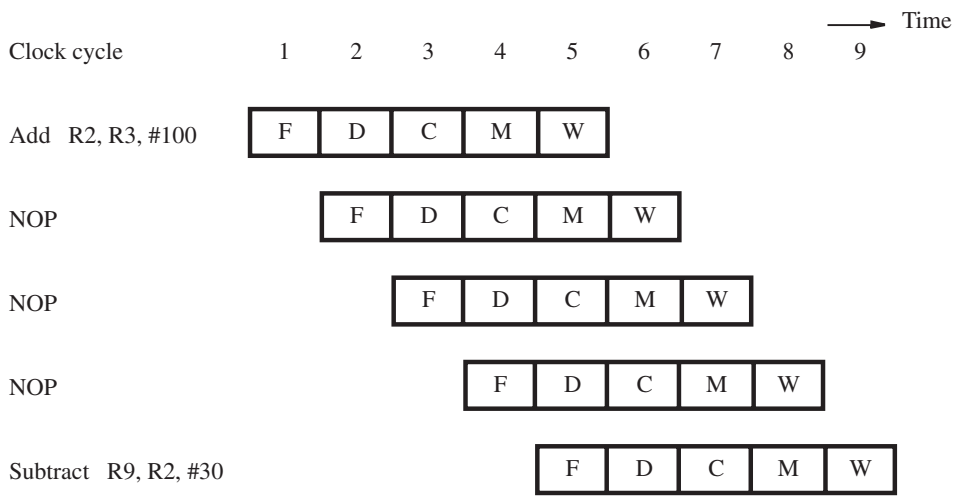
to avoid stalling the pipeline. MuxA requires another input for the value of RY. Similarly, MuxB is extended with another input.

6.4.2 HANDLING DATA DEPENDENCIES IN SOFTWARE

Figures 6.3 and 6.4 show how data dependencies may be handled by the processor hardware, either by stalling the pipeline or by forwarding data. An alternative approach is to leave the task of detecting data dependencies and dealing with them to the compiler. When the

```
Add       R2, R3, #100
NOP
NOP
NOP
Subtract  R9, R2, #30
```

(a) Insertion of NOP instructions for a data dependency



(b) Pipelined execution of instructions

Figure 6.6 Using NOP instructions to handle a data dependency in software.

compiler identifies a data dependency between two successive instructions I_j and I_{j+1}, it can insert three explicit NOP (No-operation) instructions between them. The NOPs introduce the necessary delay to enable instruction I_{j+1} to read the new value from the register file after it is written. For the instructions in Figure 6.4, the compiler would generate the instruction sequence in Figure 6.6*a*. Figure 6.6*b* shows that the three NOP instructions have the same effect on execution time as the stall in Figure 6.3.

Requiring the compiler to identify dependencies and insert NOP instructions simplifies the hardware implementation of the pipeline. However, the code size increases, and the execution time is not reduced as it would be with operand forwarding. The compiler can attempt to *optimize* the code to improve performance and reduce the code size by reordering instructions to move useful instructions into the NOP slots. In doing so, the compiler must consider data dependencies between instructions, which constrain the extent to which the NOP slots can be usefully filled.

6.5 MEMORY DELAYS

Delays arising from memory accesses are another cause of pipeline stalls. For example, a Load instruction may require more than one clock cycle to obtain its operand from memory. This may occur because the requested instruction or data are not found in the cache, resulting in a *cache miss*. Figure 6.7 shows the effect of a delay in accessing data in the memory on pipelined execution. A memory access may take ten or more cycles. For simplicity, the figure shows only three cycles. A cache miss causes all subsequent instructions to be delayed. A similar delay can be caused by a cache miss when fetching an instruction.

There is an additional type of memory-related stall that occurs when there is a data dependency involving a Load instruction. Consider the instructions:

```
Load      R2, (R3)
Subtract  R9, R2, #30
```

Assume that the data for the Load instruction is found in the cache, requiring only one cycle to access the operand. The destination register R2 for the Load instruction is a source register for the Subtract instruction. Operand forwarding cannot be done in the same manner as Figure 6.4, because the data read from memory (the cache, in this case) are not available until they are loaded into register RY at the beginning of cycle 5. Therefore, the Subtract instruction must be stalled for one cycle, as shown in Figure 6.8, to delay the ALU operation. The memory operand, which is now in register RY, can be forwarded to the ALU input in cycle 5.

The compiler can eliminate the one-cycle stall for this type of data dependency by reordering instructions to insert a useful instruction between the Load instruction and the instruction that depends on the data read from the memory. The inserted instruction fills the bubble that would otherwise be created. If a useful instruction cannot be found by the compiler, then the hardware introduces the one-cycle stall automatically. If the processor hardware does not deal with dependencies, then the compiler must insert an explicit NOP instruction.

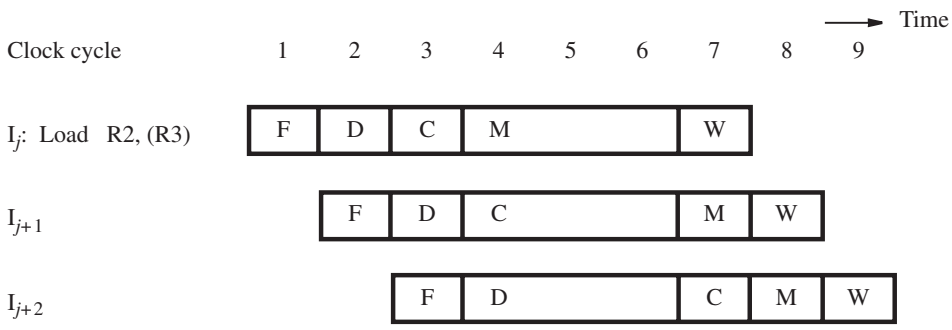


Figure 6.7 Stall caused by a memory access delay for a Load instruction.

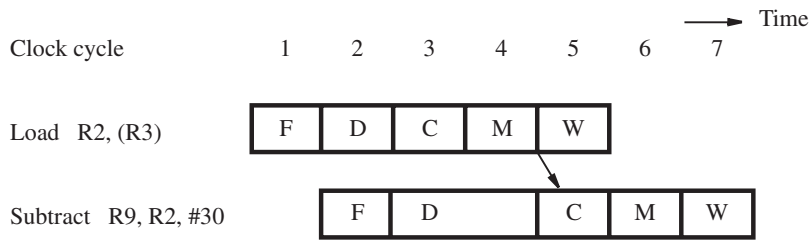


Figure 6.8 Stall needed to enable forwarding for an instruction that follows a Load instruction.

6.6 BRANCH DELAYS

In ideal pipelined execution a new instruction is fetched every cycle, while the preceding instruction is still being decoded. Branch instructions can alter the sequence of execution, but they must first be executed to determine whether and where to branch. We now examine the effect of branch instructions and the techniques that can be used for mitigating their impact on pipelined execution.

6.6.1 UNCONDITIONAL BRANCHES

Figure 6.9 shows the pipelined execution of a sequence of instructions, beginning with an unconditional branch instruction, I_j. The next two instructions, I_{j+1} and I_{j+2}, are stored in successive memory addresses following I_j. The target of the branch is instruction I_k. According to Figure 5.15, the branch instruction is fetched in cycle 1 and decoded in cycle 2, and the target address is computed in cycle 3. Hence, instruction I_k is fetched in cycle 4, after the program counter has been updated with the target address. In pipelined execution, instructions I_{j+1} and I_{j+2} are fetched in cycles 2 and 3, respectively, before the branch instruction is decoded and its target address is known. They must be discarded. The resulting two-cycle delay constitutes a *branch penalty*.

Branch instructions occur frequently. In fact, they represent about 20 percent of the dynamic instruction count of most programs. (The dynamic count is the number of instruction executions, taking into account the fact that some instructions in a program are executed many times, because of loops.) With a two-cycle branch penalty, the relatively high frequency of branch instructions could increase the execution time for a program by as much as 40 percent. Therefore, it is important to find ways to mitigate this impact on performance.

Reducing the branch penalty requires the branch target address to be computed earlier in the pipeline. Rather than wait until the Compute stage, it is possible to determine the target address and update the program counter in the Decode stage. Thus, instruction I_k can be fetched one clock cycle earlier, reducing the branch penalty to one cycle, as shown in Figure 6.10. This time, only one instruction, I_{j+1}, is fetched incorrectly, because the target address is determined in the Decode stage.

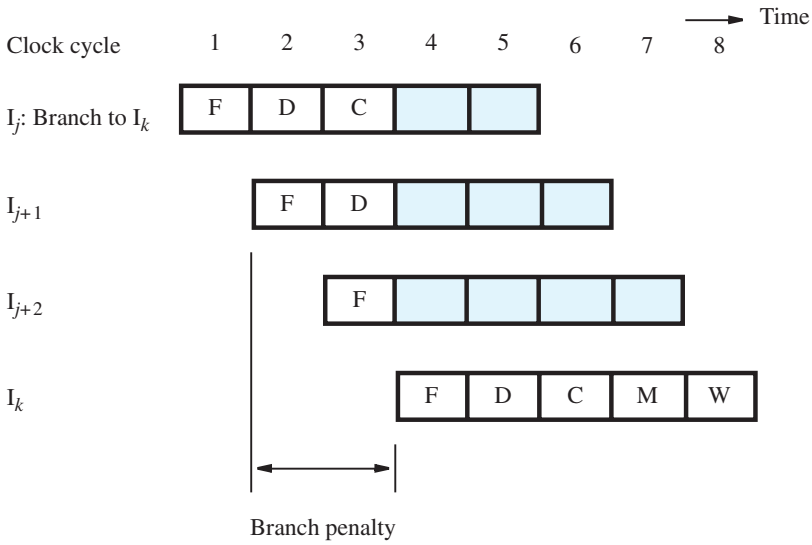


Figure 6.9 Branch penalty when the target address is determined in the Compute stage of the pipeline.

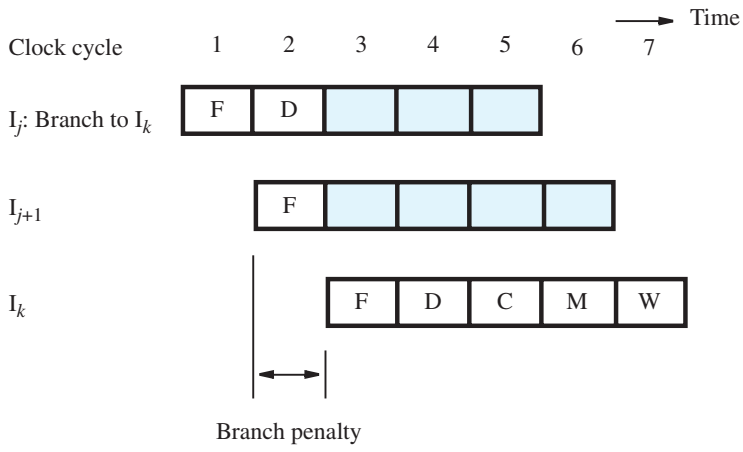


Figure 6.10 Branch penalty when the target address is determined in the Decode stage of the pipeline.

The hardware in Figure 5.10 must be modified to implement this change. The adder in the figure is needed to increment the PC in every cycle. A second adder is needed in the Decode stage to compute a branch target address for every instruction. When the instruction decoder determines that the instruction is indeed a branch instruction, the computed target address will be available before the end of the cycle. It can then be used to fetch the target instruction in the next cycle.

6.6.2 CONDITIONAL BRANCHES

Consider a conditional branch instruction such as

Branch_if_[R5]=[R6] LOOP

The execution steps for this instruction are shown in Figure 5.16. The result of the comparison in the third step determines whether the branch is taken.

For pipelining, the branch condition must be tested as early as possible to limit the branch penalty. We have just described how the target address for an unconditional branch instruction can be determined in the Decode stage. Similarly, the comparator that tests the branch condition can also be moved to the Decode stage, enabling the conditional branch decision to be made at the same time that the target address is determined. In this case, the comparator uses the values from outputs A and B of the register file directly.

Moving the branch decision to the Decode stage ensures a common branch penalty of only one cycle for all branch instructions. In the next two sections, we discuss additional techniques that can be used to further mitigate the effect of branches on execution time.

6.6.3 THE BRANCH DELAY SLOT

Consider the program fragment shown in Figure 6.11*a*. Assume that the branch target address and the branch decision are determined in the Decode stage, at the same time that instruction I_{j+1} is fetched. The branch instruction may cause instruction I_{j+1} to be discarded, after the branch condition is evaluated. If the condition is true, then there is a branch penalty of one cycle before the correct target instruction I_k is fetched. If the condition is false, then instruction I_{j+1} is executed, and there is no penalty. In both of these cases, the instruction immediately following the branch instruction is always fetched. Based on this observation, we describe a technique to reduce the penalty for branch instructions.

The location that follows a branch instruction is called the *branch delay slot*. Rather than conditionally discard the instruction in the delay slot, we can arrange to have the pipeline always execute this instruction, whether or not the branch is taken. The instruction in the delay slot cannot be I_{j+1}, the one that may be discarded depending on the branch condition. Instead, the compiler attempts to find a suitable instruction to occupy the delay slot, one that needs to be executed even when the branch is taken. It can do so by moving one of the instructions preceding the branch instruction to the delay slot. Of course, this can only be done if any data dependencies involving the instruction being moved are preserved. If a useful instruction is found, then there will be no branch penalty. If no useful instruction can be placed in the delay slot because of constraints arising from data dependencies, a NOP must be placed there instead. In this case, there will be a penalty of one cycle whether or not the branch is taken.

For the instructions in Figure 6.11*a*, the Add instruction can safely be moved into the branch delay slot, as shown in Figure 6.11*b*. The Add instruction is always fetched and executed, even if the branch is taken. Instruction I_{j+1} is fetched only if the branch is not taken. Logically, execution proceeds as though the branch instruction were placed after the

```
                Add             R7, R8, R9
                Branch_if_[R3]=0 TARGET
                I_{j+1}
                ⋮
TARGET:         I_k
```

(a) Original sequence of instructions containing a conditional branch instruction

```
                Branch_if_[R3]=0 TARGET
                Add             R7, R8, R9
                I_{j+1}
                ⋮
TARGET:         I_k
```

(b) Placing the Add instruction in the branch delay slot where it is always executed

Figure 6.11 Filling the branch delay slot with a useful instruction.

Add instruction. That is, branching takes place one instruction later than where the branch instruction appears in the instruction sequence. This technique is called *delayed branching*.

The effectiveness of delayed branching depends on how often the compiler can reorder instructions to usefully fill the delay slot. Experimental data collected from many programs indicate that the compiler can fill a branch delay slot in 70 percent or more of the cases.

6.6.4 BRANCH PREDICTION

The discussion above shows that making the branch decision in cycle 2 of the execution of a branch instruction reduces the branch penalty. But, even then, the instruction immediately following the branch instruction is still fetched in cycle 2 and may have to be discarded. The decision to fetch this instruction is actually made in cycle 1, when the PC is incremented while the branch instruction itself is being fetched. Thus, to reduce the branch penalty further, the processor needs to anticipate that an instruction being fetched is a branch instruction and *predict* its outcome to determine which instruction should be fetched in cycle 2. In this section, we first describe different methods for branch prediction. Then, we discuss how the prediction is made in cycle 1 while a branch instruction is being fetched.

Static Branch Prediction

The simplest form of branch prediction is to assume that the branch will not be taken and to fetch the next instruction in sequential address order. If the prediction is correct, the fetched instruction is allowed to complete and there is no penalty. However, if it is determined that the branch is to be taken, the instruction that has been fetched is discarded and the correct branch target instruction is fetched. Misprediction incurs the full branch penalty. This simple approach is a form of *static branch prediction*. The same choice (assume not-taken) is used every time a conditional branch is encountered.

If branch outcomes were random, then half of all conditional branches would be taken. In this case, always assuming that branches will not be taken results in a prediction accuracy of 50 percent. However, a backward branch at the end of a loop is taken most of the time. For such a branch, better accuracy can be achieved by predicting that the branch is likely to be taken. Thus, instructions are fetched using the branch target address as soon as it is known. Similarly, for a forward branch at the beginning of a loop, the not-taken prediction leads to good prediction accuracy. The processor can determine the static prediction of taken or not-taken by checking the sign of the branch offset. Alternatively, the machine encoding of a branch instruction may include one bit that indicates whether the branch should be predicted as taken or nor taken. The setting of this bit can be specified by the compiler.

Dynamic Branch Prediction

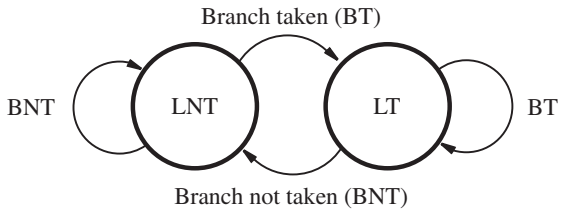
To improve prediction accuracy further, we can use actual branch behavior to influence the prediction, resulting in *dynamic branch prediction*. The processor hardware assesses the likelihood of a given branch being taken by keeping track of branch decisions every time that a branch instruction is executed.

In its simplest form, a dynamic prediction algorithm can use the result of the most recent execution of a branch instruction. The processor assumes that the next time the instruction is executed, the branch decision is likely to be the same as the last time. Hence, the algorithm may be described by the two-state machine in Figure 6.12*a*. The two states are:

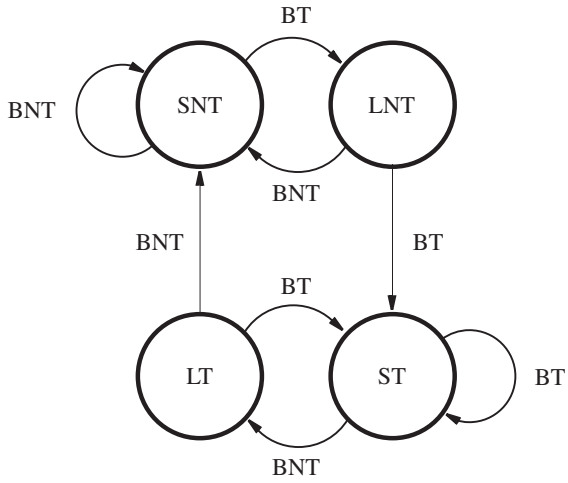
LT - Branch is likely to be taken
LNT - Branch is likely not to be taken

Suppose that the algorithm is started in state LNT. When the branch instruction is executed and the branch is taken, the machine moves to state LT. Otherwise, it remains in state LNT. The next time the same instruction is encountered, the branch is predicted as taken if the state machine is in state LT. Otherwise it is predicted as not taken.

This simple scheme, which requires only a single bit to represent the history of execution for a branch instruction, works well inside program loops. Once a loop is entered, the decision for the branch instruction that controls looping will always be the same except for the last pass through the loop. Hence, each prediction for the branch instruction will be correct except in the last pass. The prediction in the last pass will be incorrect, and the branch history state machine will be changed to the opposite state. Unfortunately, this means that the next time this same loop is entered—and assuming that there will be more than one pass through the loop—the state machine will lead to the wrong prediction for the



(a) A 2-state algorithm



(b) A 4-state algorithm

Figure 6.12 State-machine representation of branch prediction algorithms.

first pass. Thus, repeated execution of the same loop results in mispredictions in the first pass and the last pass.

Better prediction accuracy can be achieved by keeping more information about execution history. An algorithm that uses four states is shown in Figure 6.12*b*. The four states are:

ST	-	Strongly likely to be taken
LT	-	Likely to be taken
LNT	-	Likely not to be taken
SNT	-	Strongly likely not to be taken

Again assume that the state of the algorithm is initially set to LNT. After the branch instruction is executed, and if the branch is actually taken, the state is changed to ST; otherwise, it is changed to SNT. As program execution progresses and the same branch instruction is encountered multiple times, the state of the prediction algorithm changes as shown. The branch is predicted as taken if the state is either ST or LT. Otherwise, the branch is predicted as not taken.

Let us reconsider what happens when executing a program loop. Assume that the branch instruction is at the end of the loop and that the processor sets the initial state of the algorithm to LNT. In the first pass, the prediction (not taken) will be wrong, and hence the state will be changed to ST. In all subsequent passes, the prediction will be correct, except for the last pass. At that time, the state will change to LT. When the loop is entered a second time, the prediction in the first pass will be to take the branch, which will be correct if there is more than one iteration. Thus, repeated execution of the same loop now results in only one misprediction in the last pass.

Branch Target Buffer for Dynamic Prediction

In earlier discussion, we pointed out that the branch target address and the branch decision can both be determined in the Decode stage of the pipeline, which is cycle 2 of instruction execution. The instruction being fetched in the same cycle may or may not be the one that has to be executed after the branch instruction. It may have to be discarded, in which case the correct instruction will be fetched in cycle 3. How can branch prediction be used to obtain better performance?

The key to improving performance is to increase the likelihood that the instruction fetched in cycle 2 is the correct one. This can be achieved only if branch prediction takes place in cycle 1, at the same time that the branch instruction is being fetched. To make this possible, the processor needs to keep more information about the history of execution. The required information is usually stored in a small, fast memory called the *branch target buffer*.

The branch target buffer identifies branch instructions by their addresses. As each branch instruction is executed, the processor records the address of the instruction and the outcome of the branch decision in the buffer. The information is organized in the form of a lookup table, in which each entry includes:

- the address of the branch instruction
- one or two state bits for the branch prediction algorithm
- the branch target address

With this information, the processor is able to identify branch instructions and obtain the corresponding branch prediction state bits based on the address of the instruction being fetched.

Every time the processor fetches a new instruction, it checks the branch target buffer for an entry containing the same instruction address. If an entry with that address is found, this means that the instruction being fetched is a branch instruction. The processor is then able to use the state bits to predict whether that branch is likely to be taken. At the same time, the target address is also obtained. The processor is able to obtain this information as the branch instruction is being fetched in cycle 1. In cycle 2, the processor uses the predicted

outcome of the branch to fetch the next instruction. Of course, it must also determine the actual branch decision and target address to determine whether the predicted values were correct. If they are, execution continues without penalty. Otherwise, the instruction that has just been fetched is discarded, and the correct instruction is fetched in cycle 3. The main value of the branch target buffer is that the state information needed for branch prediction and the target address of a branch instruction are both obtained at the same time the branch instruction is being fetched.

Large programs have many branch instructions. A branch target buffer with enough storage to accommodate information for all of them would be large, and searching it quickly would be difficult. For this reason, the table has a limited size, containing information for only the most recently executed branch instructions. Entries in the table are replaced as other branch instructions are executed. Typically, the table contains on the order of 1024 entries.

6.7 RESOURCE LIMITATIONS

Pipelining enables overlapped execution of instructions, but the pipeline stalls when there are insufficient hardware resources to permit all actions to proceed concurrently. If two instructions need to access the same resource in the same clock cycle, one instruction must be stalled to allow the other instruction to use the resource. This can be prevented by providing additional hardware.

Such stalls can occur in a computer that has a single cache that supports only one access per cycle. If both the Fetch and Memory stages of the pipeline are connected to the cache, then it is not possible for activity in both stages to proceed simultaneously. Normally, the Fetch stage accesses the cache in every cycle. However, this activity must be stalled for one cycle when there is a Load or Store instruction in the Memory stage also needing to access the cache. If 25 percent of all instructions executed are Load or Store instructions, these stalls increase the execution time by 25 percent. Using separate caches for instructions and data allows the Fetch and Memory stages to proceed simultaneously without stalling.

6.8 PERFORMANCE EVALUATION

For a non-pipelined processor, the execution time, T, of a program that has a dynamic instruction count of N is given by

$$T = \frac{N \times S}{R}$$

where S is the average number of clock cycles it takes to fetch and execute one instruction, and R is the clock rate in cycles per second. This is often referred to as the *basic performance equation*. A useful performance indicator is the *instruction throughput*, which is the number of instructions executed per second. For non-pipelined execution, the throughput, P_{np}, is given by

$$P_{np} = \frac{R}{S}$$

The processor presented in Chapter 5 uses five cycles to execute all instructions. Thus, if there are no cache misses, S is equal to 5.

Pipelining improves performance by overlapping the execution of successive instructions, which increases instruction throughput even though an individual instruction is still executed in the same number of cycles. For the five-stage pipeline described in this chapter, each instruction is executed in five cycles, but a new instruction can ideally enter the pipeline every cycle. Thus, in the absence of stalls, S is equal to 1, and the ideal throughput with pipelining is

$$P_p = R$$

A five-stage pipeline can potentially increase the throughput by a factor of five. In general, an n-stage pipeline has the potential to increase throughput n times. Thus, it would appear that the higher the value of n, the larger the performance gain. This leads to two questions:

- How much of this potential increase in instruction throughput can actually be realized in practice?
- What is a good value for n?

Any time a pipeline is stalled or instructions are discarded, the instruction throughput is reduced below its ideal value. Hence, the performance of a pipeline is highly influenced by factors such as stalls due to data dependencies between instructions and penalties due to branches. Cache misses increase the execution time even further. We discuss these issues first, and then we return to the question of how many pipeline stages should be used.

6.8.1 EFFECTS OF STALLS AND PENALTIES

The effects of stalls and penalties have been examined qualitatively in the previous sections. We now consider these effects in quantitative terms.

The five-stage pipeline involves memory-access operations in the Fetch and Memory stages, and ALU operations in the Compute stage. The operations with the longest delay dictate the cycle time, and hence the clock rate R. For a processor that has on-chip caches, memory-access operations have a small delay when the desired instructions or data are found in the cache. The delay through the ALU is likely to be the critical parameter. If this delay is 2 ns, then $R = 500$ MHz, and the ideal pipelined instruction throughput is $P_p = 500$ MIPS (million instructions per second).

Consider a processor with operand forwarding in hardware, as explained in Section 6.4.1. This means that there are no penalties due to data dependencies, except in the case of Load instructions. To evaluate the effect of stalls not related to cache misses, we can consider how often a Load instruction is immediately followed by another instruction that uses the result of the memory access. Section 6.5 explained that a one-cycle stall is necessary in such cases. While ideal pipelined execution has $S = 1$, stalls due to such Load instructions have the effect of increasing S by an amount δ_{stall}. For example, assume that Load instructions constitute 25 percent of the dynamic instruction count, and assume that 40 percent of these Load instructions are followed by a dependent instruction. A one-cycle

stall is needed in such cases. Hence, the increase over the ideal case of $S = 1$ is

$$\delta_{\text{stall}} = 0.25 \times 0.40 \times 1 = 0.10$$

That is, the execution time T is increased by 10 percent, and throughput is reduced to

$$P_p = \frac{R}{1 + \delta_{\text{stall}}} = \frac{R}{1.1} = 0.91R$$

The compiler can improve performance by reducing the number of times that a Load instruction is immediately followed by a dependent instruction. A stall is eliminated each time the compiler can safely move a nearby instruction to a position between the Load instruction and the dependent instruction.

Now, consider the penalties due to mispredicting branches during program execution. When both the branch decision and the branch target address are determined in the Decode stage of the pipeline, the branch penalty is one cycle. Assume that branches constitute 20 percent of the dynamic instruction count of a program, and that the average prediction accuracy for branch instructions is 90 percent. In other words, 10 percent of all branch instructions that are executed incur a one-cycle penalty due to misprediction. The increase in the average number of cycles per instruction due to branch penalties is

$$\delta_{\text{branch\_penalty}} = 0.20 \times 0.10 \times 1 = 0.02$$

High prediction accuracy is beneficial in limiting the adverse impact of this penalty on performance.

The stalls related to Load instructions and the penalties from branch misprediction are independent. Hence, their effect on performance is additive. The sum of δ_{stall} and $\delta_{\text{branch\_penalty}}$ determines the increase in the number of cycles, S, the increase in the execution time, T, and the reduction in the throughput, P_p.

The effect of cache misses on performance can be assessed by considering the frequency of their occurrence. The time to access the slower main memory is a penalty that stalls the pipeline for p_m cycles every time there is a cache miss. A fraction m_i of all instructions that are fetched incur a cache miss. A fraction d of all instructions are Load or Store instructions, and a fraction m_d of these instructions incur a cache miss. The increase over the ideal case of $S = 1$ due to cache misses is

$$\delta_{\text{miss}} = (m_i + d \times m_d) \times p_m$$

Suppose that 5 percent of all fetched instructions incur a cache miss, 30 percent of all instructions executed are Load or Store instructions, and 10 percent of their data-operand accesses incur a cache miss. Assume that the penalty to access the main memory for a cache miss is 10 cycles. The increase over the ideal case of $S = 1$ due to cache misses in this case is given by

$$\delta_{\text{miss}} = (0.05 + 0.30 \times 0.10) \times 10 = 0.8$$

Compared to δ_{stall} for data dependencies and $\delta_{\text{branch\_penalty}}$ for mispredicted branches, the effect of a slow main memory for cache misses is more significant in this example. When all factors are combined, S is increased from the ideal value of 1 to $1 + \delta_{\text{stall}} + \delta_{\text{branch\_penalty}} + \delta_{\text{miss}}$. The contribution of cache misses is often the dominant one.

6.8.2 NUMBER OF PIPELINE STAGES

The fact that an *n*-stage pipeline may increase instruction throughput by a factor of *n* suggests that we should use a large number of stages. However, as the number of pipeline stages increases, there are more instructions being executed concurrently. Consequently, there are more potential dependencies between instructions that may lead to pipeline stalls. Furthermore, the branch penalty may be larger than one cycle if a longer pipeline moves the branch decision to a later stage. For these reasons, the gain in throughput from increasing the value of *n* begins to diminish, and the cost of a deeper pipeline may not be justified.

Another important factor is the inherent delay in the basic operations performed by the processor. The most important among these is the ALU delay. In many processors, the cycle time of the processor clock is chosen such that one ALU operation can be completed in one cycle. Other operations, including accesses to a cache memory, are typically divided into steps that each take about the same time as an ALU operation. Further reductions in the clock cycle time are possible if a pipelined ALU is used. Some recent processor implementations have used twenty or more pipeline stages to aggressively reduce the cycle time. Implementing such long pipelines using modern technology allows for clock rates of several GHz.

6.9 SUPERSCALAR OPERATION

The maximum throughput of a pipelined processor is one instruction per clock cycle. A more aggressive approach is to equip the processor with multiple execution units, each of which may be pipelined, to increase the processor's ability to handle several instructions in parallel. With this arrangement, several instructions start execution in the same clock cycle, but in different execution units, and the processor is said to use *multiple-issue*. Such processors can achieve an instruction execution throughput of more than one instruction per cycle. They are known as *superscalar* processors. Many modern high-performance processors use this approach.

To enable multiple-issue execution, a superscalar processor has a more elaborate *fetch unit* that fetches two or more instructions per cycle before they are needed and places them in an instruction queue. A separate unit, called the *dispatch unit*, takes two or more instructions from the front of the queue, decodes them, and sends them to the appropriate execution units. At the end of the pipeline, another unit is responsible for writing results into the register file. Figure 6.13 shows a superscalar processor with this organization. It incorporates two execution units, one for arithmetic instructions and another for Load and Store instructions. Arithmetic operations normally require only one cycle, hence the first execution unit is simple. Because Load and Store instructions involve an address calculation for the Index mode before each memory access, the Load/Store unit has a two-stage pipeline.

The organization in Figure 6.13 raises some important implications for the register file. An arithmetic instruction and a Load or Store instruction must obtain all their operands from the register file when they are dispatched in the same cycle to the two execution units. The register file must now have four output ports instead of the two output ports needed in

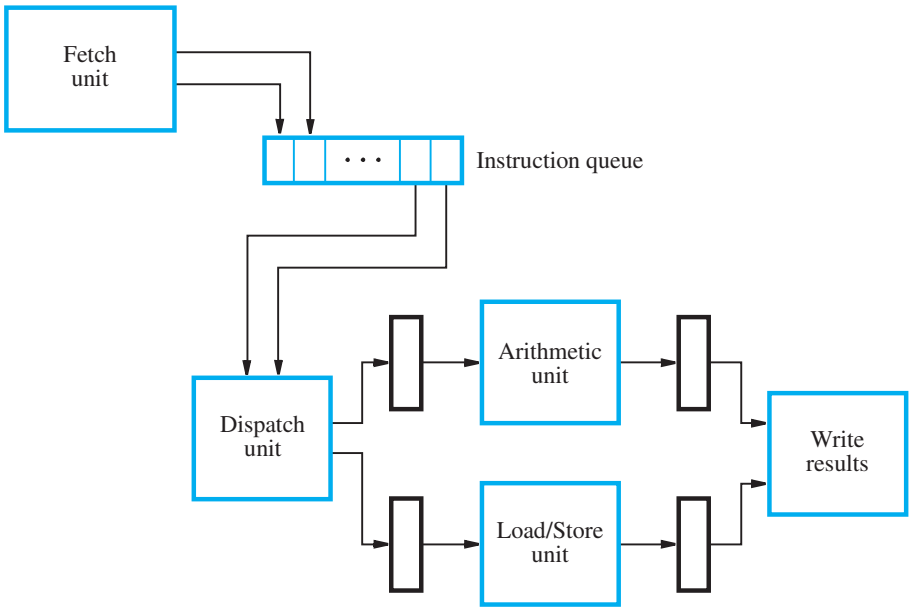


Figure 6.13 A superscalar processor with two execution units.

the simple pipeline. Similarly, an arithmetic instruction and a Load instruction must write their results into the register file when they complete in the same cycle. Thus, the register file must now have two input ports instead of the single input port for the simple pipeline. There is also the potential complication of two instructions completing at the same time with the same destination register for their results. This complication is avoided, if possible, by dispatching the instructions in a manner that prevents its occurrence. Otherwise, one instruction is stalled to ensure that results are written into the destination register in the same order as in the original instruction sequence of the program.

To illustrate superscalar execution in the processor in Figure 6.13, consider the following sequence of instructions:

```
Add       R2, R3, #100
Load      R5, 16(R6)
Subtract  R7, R8, R9
Store     R10, 24(R11)
```

Figure 6.14 shows how these instructions would be executed. The fetch unit fetches two instructions every cycle. The instructions are decoded and their source registers are read in the next cycle. Then, they are dispatched to the arithmetic and Load/Store units. Arithmetic operations can be initiated every cycle. A Load or Store instruction can also be initiated every cycle, because the two-stage pipeline overlaps the address calculation for one Load or Store instruction with the memory access for the preceding Load or Store instruction.

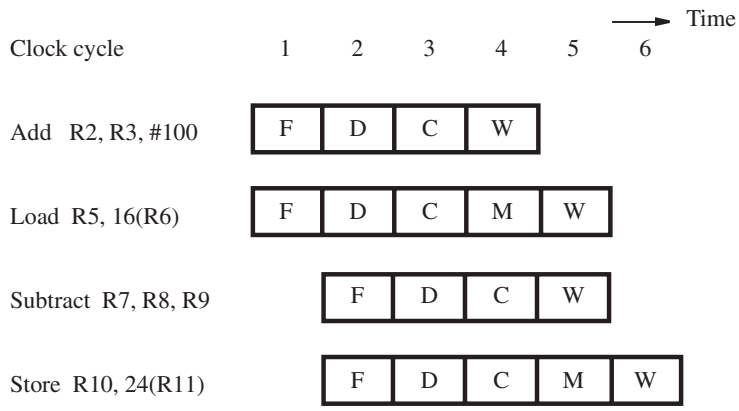


Figure 6.14 An example of instruction flow in the processor of Figure 6.13.

As instructions complete execution in each unit, the register file allows two results to be written in the same cycle because the destination registers are different.

6.9.1 BRANCHES AND DATA DEPENDENCIES

In the absence of any branch instructions and any data dependencies between instructions, throughput is maximized by interleaving instructions that can be dispatched simultaneously to different execution units. However, programs contain branch instructions that change the execution flow, and data dependencies between instructions that impose sequential ordering constraints. A superscalar processor must ensure that instructions are executed in the proper sequence. Furthermore, memory delays due to cache misses may occasionally stall the fetching and dispatching of instructions. As a result, actual throughput is typically below the maximum that is possible. The challenges presented by branch instructions and data dependencies can be addressed with additional hardware. We first consider branch instructions and then consider the issues stemming from data dependencies.

The fetch unit handles branch instructions as it determines which instructions to place in the queue for dispatching. It must determine both the branch decision and the target for each branch instruction. The branch decision may depend on the result of an earlier instruction that is either still queued or newly dispatched. Stalling the fetch unit until the result is available can significantly reduce the throughput and is therefore not a desirable approach. Instead, it is better to employ branch prediction. Since the aim is to achieve high throughput, prediction is also combined with a technique called *speculative execution*. In this technique, subsequent instructions based on an unconfirmed prediction are fetched, dispatched, and possibly executed, but are labeled as being speculative so that they and their results may be discarded if the prediction is incorrect. Additional hardware is required to maintain information about speculatively executed instructions and to ensure that registers or memory locations are not modified until the validity of the prediction is confirmed.

Additional hardware is also needed to ensure that the correct instructions are fetched and dispatched in the event of misprediction.

Data dependencies between instructions impose ordering constraints. A simple approach is to dispatch dependent instructions in sequence to the same execution unit, where their order would be preserved. However, dependent instructions may be dispatched to different execution units. For example, the result of a Load instruction dispatched to the Load/Store unit in Figure 6.13 may be needed by an Add instruction dispatched to the arithmetic unit. Because the units operate independently and because other instructions may have already been dispatched to them, there is no guarantee as to when the result needed by the Add instruction is generated by the Load instruction. A mechanism is needed to ensure that a dependent instruction waits for its operands to become available. When an instruction is dispatched to an execution unit, it is buffered until all necessary results from other instructions have been generated. Such buffers are called *reservation stations*, and they are used to hold information and operands relevant to each dispatched instruction. Results from each execution unit are broadcast to all reservation stations with each result tagged with a register identifier. This enables the reservation stations to recognize a result on which a buffered instruction depends. When there is a matching tag, the hardware copies the result into the reservation station containing the instruction. The control circuit begins the execution of a buffered instruction only when it has all of its operands.

In a superscalar processor using multiple-issue, the detrimental effect of stalls becomes even more pronounced than in a single-issue pipelined processor. The compiler can avoid many stalls through judicious selection and ordering of instructions. For example, for the processor in Figure 6.13, the compiler should strive to interleave arithmetic and memory instructions. This enables the dispatch unit to keep both units busy most of the time.

6.9.2 OUT-OF-ORDER EXECUTION

The instructions in Figure 6.14 are dispatched in the same order as they appear in the program. However, their execution may be completed out of order. For example, the Subtract instruction writes to register R7 in the same cycle as the Load instruction that was fetched earlier writes to register R5. If the memory access for the Load instruction requires more than one cycle to complete, execution of the Subtract instruction would be completed before the Load instruction. Does this type of situation lead to problems?

We have already discussed the issues arising from dependencies among instructions. For example, if an instruction I_{j+1} depends on the result of instruction I_j, the execution of I_{j+1} will be delayed if the result is not available when it is needed. As long as such dependencies are handled correctly, there is no reason to delay the execution of an unrelated instruction. If there is no dependency between a pair of instructions, the order in which execution is completed does not matter.

However, a new complication arises when we consider the possibility of an instruction causing an exception. For example, the Load instruction in Figure 6.14 may attempt an illegal unaligned memory access for a data operand. By the time this illegal operation is recognized, the Subtract instruction that is fetched after the Load instruction may have already modified its destination register. Program execution is now in an inconsistent

state. The instruction that caused the exception in the original sequence is identified, but a succeeding instruction in that sequence has been executed to completion. If such a situation is permitted, the processor is said to have *imprecise exceptions*.

The alternative of *precise exceptions* requires additional hardware. To guarantee a consistent state when exceptions occur, the results of the execution of instructions must be written into the destination locations strictly in program order. This means that we must delay writing into register R7 for the Subtract instruction in Figure 6.14 until after register R5 for the Load instruction has been updated. Either the arithmetic unit in Figure 6.13 must retain the result of the Subtract instruction, or the result must be buffered in a temporary register until preceding instructions have written their results. If an exception occurs during the execution of an instruction, all subsequent instructions and their buffered results are discarded.

It is easier to provide precise exceptions in the case of external interrupts. When an external interrupt is received, the dispatch unit stops reading new instructions from the instruction queue, and the instructions remaining in the queue are discarded. All instructions whose execution is pending continue to completion. At this point, the processor and all its registers are in a consistent state, and interrupt processing can begin.

6.9.3 EXECUTION COMPLETION

To improve performance, an execution unit should be allowed to execute any instructions whose operands are ready in its reservation station. This may lead to out-of-order execution of instructions. However, instructions must be completed in program order to allow precise exceptions. These seemingly conflicting requirements can be resolved if execution is allowed to proceed out of order, but the results are written into temporary registers. The contents of these registers are later transferred to the permanent registers in correct program order. This last step is often called the *commitment* step, because the effect of an instruction cannot be reversed after that point. If an instruction causes an exception, the results of any subsequent instructions that have been executed would still be in temporary registers and can be safely discarded. Results that would normally be written to memory would also be buffered temporarily, and they can be safely discarded as well.

A temporary register that is assigned for the result of an instruction assumes the role of the permanent register whose data it is holding. Its contents are forwarded to any subsequent instruction that refers to the original permanent register during that period. This technique is called *register renaming*. There may be as many temporary registers as there are permanent registers, or there may be fewer temporary registers that are allocated as needed for association with different permanent registers.

When out-of-order execution is allowed, a special control unit is needed to guarantee in-order commitment. This is called the *commitment unit*. It uses a separate queue called the *reorder buffer* to determine which instruction(s) should be committed next. Instructions are entered in the queue strictly in program order as they are dispatched for execution. When an instruction reaches the head of this queue and the execution of that instruction has been completed, the corresponding results are transferred from the temporary registers to the permanent registers and the instruction is removed from the queue. All resources that were assigned to the instruction, including the temporary registers, are released. The instruction

is said to have been *retired* at this point. Because an instruction is retired only when it is at the head of the queue, all instructions that were dispatched before it must also have been retired. Hence, instructions may complete execution out of order, but they are retired in program order.

6.9.4 DISPATCH OPERATION

We now return to the dispatch operation. When dispatching decisions are made, the dispatch unit must ensure that all the resources needed for the execution of an instruction are available. For example, since the results of an instruction may have to be written in a temporary register, there should be one available, and it is reserved for use by that instruction as a part of the dispatch operation. There must be space available in the reservation station of an appropriate execution unit. Finally, a location in the reorder buffer for later commitment of results must also be available for the instruction. When all the resources needed are assigned, the instruction is dispatched.

Should instructions be dispatched out of order? For example, the dispatch of the Load instruction in Figure 6.14 may be delayed because there is no space in the reservation station of the Load/Store unit as a result of a cache miss in a previously dispatched instruction. Should the Subtract instruction be dispatched instead? In principle this is possible, provided that all the resources needed by the Load instruction, including a place in the reorder buffer, are reserved for it. This is essential to ensure that all instructions are ultimately retired in the correct order and that no deadlocks occur.

A *deadlock* is a situation that can arise when two units, A and B, use a shared resource. Suppose that unit B cannot complete its operation until unit A completes its operation. At the same time, unit B has been assigned a resource that unit A needs. If this happens, neither unit can complete its operation. Unit A is waiting for the resource it needs, which is being held by unit B. At the same time, unit B is waiting for unit A to finish before it can complete its operation and release that resource.

As an example of a deadlock when dispatching instructions out of order, consider a superscalar processor that has only one temporary register. When the Subtract instruction in Figure 6.14 is dispatched before the Load instruction, the temporary register is reserved for it. The Load instruction cannot be dispatched because it is waiting for the same temporary register, which, in turn, will not become free until the Subtract instruction is retired. Since the Subtract instruction cannot be retired before the Load instruction, we have a deadlock.

To prevent deadlocks, the dispatch unit must take many factors into account. Hence, issuing instructions out of order is likely to increase the complexity of the dispatch unit significantly. It may also mean that more time is required to make dispatching decisions. Dispatching instructions in order avoids this complexity. In this case, the program order of instructions is enforced at the time instructions are dispatched and again at the time they are retired. Between these two events, the execution of several instructions across multiple execution units can proceed out of order, subject only to interdependencies among them.

A final comment on superscalar processors concerns the number of execution units. The processor in Figure 6.13 has one arithmetic unit and one Load/Store unit. For higher performance, modern superscalar processors often have two arithmetic units for integer operations, as well as a separate arithmetic unit for floating-point operations. The floating-

point unit has its own register file. Many processors also include a vector unit for integer or floating-point arithmetic, which typically performs two to eight operations in parallel. Such a unit may also have a dedicated register file. A single Load/Store unit typically supports all memory accesses to or from the register files for integer, floating-point, or vector units. To keep many execution units busy, modern processors may fetch four or more instructions at the same time to place at the tail of the instruction queue, and similarly four or more instructions may be dispatched to the execution units from the head of the instruction queue.

6.10 PIPELINING IN CISC PROCESSORS

The instruction set of a RISC processor makes pipelining relatively easy to implement. All instructions are one word in size, and operand information is typically located in the same position within a word for different instructions. No instruction requires more than one memory operand. Only Load and Store instructions access memory operands, typically using only indexed addressing. All other instructions operate on register operands. The five-stage pipeline described in this chapter is tailored for these characteristics of RISC-style instructions.

For pipelining in CISC processors, complications arise due to instructions that are variable in size, have multiple memory operands and complex addressing modes, and use condition codes. Instructions that occupy more than one word may take several cycles to fetch. Furthermore, variability in instruction size and format complicates both decoding and operand access, as well as management of the dispatch queue in a superscalar processor.

The availability of more complex addressing modes such as Autoincrement or Autodecrement introduces *side effects* when executing instructions. A side effect occurs when a location other than that of the destination operand is also affected. For example, the instruction

```
Move    R5, (R8)+
```

has a side effect. Not only is the destination register R5 affected, but source register R8 is also affected by the autoincrement operation. Should a later instruction depend on the value in register R8, this dependency must be handled with additional hardware in the same manner as a dependency involving the destination register, R5. It may require stalling the pipeline or forwarding the new value. In a superscalar processor, such a dependency requires the use of temporary registers and register renaming as discussed in Section 6.9.3.

Condition codes also introduce side effects. For example, in the sequence of instructions

```
Compare     R7, R8
Branch>0    TARGET
```

the result of the Compare instruction affects the condition code flags as a side effect. The Branch instruction, in turn, implicitly depends on this side effect. A condition code register can be included with relative ease in a simple pipeline such as the one shown in Figure 6.2, because only one ALU operation is performed in any cycle. However, in a superscalar processor with multiple execution units, many instructions may be in various

stages of execution, and two or more ALU operations may be performed in each cycle. Dependencies arising from side effects related to the condition codes require the use of additional temporary registers and register renaming.

Finally, consider the following sequence of CISC-style instructions:

```
Move    (R2), (R3)
Move    (R4), R5
```

The first Move instruction requires two operand accesses to the memory, while the second Move instruction requires only one. Executing these instructions in a pipeline such as the one in Figure 6.2 requires additional hardware to stall the second Move instruction so that the first Move instruction can complete its two operand accesses to the memory. In a superscalar processor such as the one in Figure 6.13, the Load/Store unit must similarly stall its internal pipeline.

CISC-style instructions complicate pipelining. This was one of the main reasons for developing the RISC approach. Nonetheless, pipelined processors have been implemented for CISC-style instruction sets, which were initially introduced before the widespread use of pipelining. Examples include processors based on the ColdFire and Intel instruction sets discussed in Appendices C and E. ColdFire processors are primarily intended for embedded applications, while Intel processors serve general-purpose needs. Consequently, the extent to which pipelining is used in ColdFire processors is less than that in Intel processors.

6.10.1 PIPELINING IN COLDFIRE PROCESSORS

ColdFire processor implementations labeled as versions V1 and V2 have two pipelines in series with a first-in first-out (FIFO) buffer between them. A two-stage instruction fetch pipeline prefetches instructions into the buffer. This buffer then feeds a two-stage pipeline that executes instructions. Instructions that involve register-only or register-to-memory operations pass once through the two execution stages. Instructions that involve memory-to-register or memory-to-memory operations must make two passes through the execution stages.

Later versions of ColdFire processor implementations use a similar buffer arrangement between two pipelines, but they incorporate various enhancements for higher performance. For example, the instruction fetch pipeline in version V4 is extended to four stages and includes branch prediction. The execution pipeline is extended to five stages. The early stages are used for address calculation, and the later stages are used for arithmetic/logic operations. This separation of functions enables a limited form of superscalar processing. In certain cases, a Move instruction and another instruction can be issued to the execution pipeline in the same cycle. Version V5 implementations have two distinct execution pipelines based on the V4 organization. They provide true superscalar processing.

6.10.2 PIPELINING IN INTEL PROCESSORS

Intel processors achieve high performance with superscalar execution and deep pipelines. For example, the Core 2 and Core i7 architectures have a multiple-issue width of four

instructions and a 14-stage pipeline. Branch prediction, register renaming, out-of-order execution, and other techniques are used.

To reduce internal complexity, CISC-style instructions are dynamically converted by the hardware into simpler RISC-style micro-operations. These micro-operations are then issued to the execution units to complete the tasks specified by the original CISC-style instructions. This approach preserves code compatibility while making it possible to use the aggressive performance enhancement techniques that have been developed for RISC-style instruction sets. In some cases, micro-operations are fused back together into macro-operations for more efficient handling. For example, in a program containing original CISC-style instructions, a comparison instruction that affects condition codes is often followed by a branch instruction. The hardware may initially convert the comparison and branch instructions into separate micro-operations, but would then fuse them into a combined compare-and-branch operation, whose function reflects what is typically found in a RISC-style instruction set.

6.11 Concluding Remarks

Two important features for performance enhancement have been introduced in this chapter, pipelining and multiple-issue. Pipelining enables processors to have instruction throughput approaching one instruction per clock cycle. Multiple-issue combined with pipelining makes possible superscalar operation, with instruction throughput of several instructions per clock cycle.

The potential gain in performance can only be realized by careful attention to three aspects:

- The instruction set of the processor
- The design of the pipeline hardware
- The design of the associated compiler

It is important to appreciate that there are strong interactions among all three aspects. High performance is critically dependent on the extent to which these interactions are taken into account in the design of a processor. Instruction sets that are particularly well-suited for pipelined execution are key features of modern processors.

There are many sources that provide additional details on the topics presented in this chapter. Reference [1] covers pipelining and Reference [2] covers superscalar processors.

6.12 Examples of Solved Problems

This section presents some examples of the types of problems that a student may be asked to solve, and shows how such problems can be solved.

Example 6.1

Problem: Consider the pipelined execution of the following sequence of instructions:

```
Add       R4, R3, R2
Or        R7, R6, R5
Subtract  R8, R7, R4
```

Initially, registers R2 and R3 contain 4 and 8, respectively. Registers R5 and R6 contain 128 and 2, respectively. Assume that the pipeline provides forwarding paths to the ALU from registers RY and RZ in Figure 5.8. The first instruction is fetched in cycle 1, and the remaining instructions are fetched in successive cycles.

Draw a diagram similar to Figure 6.1 to show the pipelined execution of these instructions assuming that the processor uses operand forwarding. Then, with reference to Figure 5.8, describe the contents of registers RY and RZ during cycles 4 to 7.

Solution: There are data dependencies involving registers R4 and R7. The Subtract instruction needs the new values for these registers before they are written to the register file. Hence, those values need to be forwarded to the ALU inputs when the Subtract instruction is in the Compute stage of the pipeline. Figure 6.15 shows the execution with forwarding. One arrow represents the new value of register R7 being forwarded from register RZ, and the other arrow represents the new value of register R4 being forwarded from register RY.

As for the contents of registers RY and RZ during cycles 4 to 7, the following description provides the answer.

- Using the initial values for registers R2 and R3, the Add instruction generates the result of 12 in cycle 3. That result is available in register RZ during cycle 4. The value in register RY during cycle 4 is the result for the unspecified instruction preceding the Add instruction.
- In cycle 4, the Or instruction generates the result of 130. That result is placed in register RZ to be available during cycle 5. The result of 12 for the Add instruction is in register RY during cycle 5.
- In cycle 5, the Subtract instruction is in the Compute stage. To generate a correct result, forwarding is used to provide the value of 130 in register RY and the value of

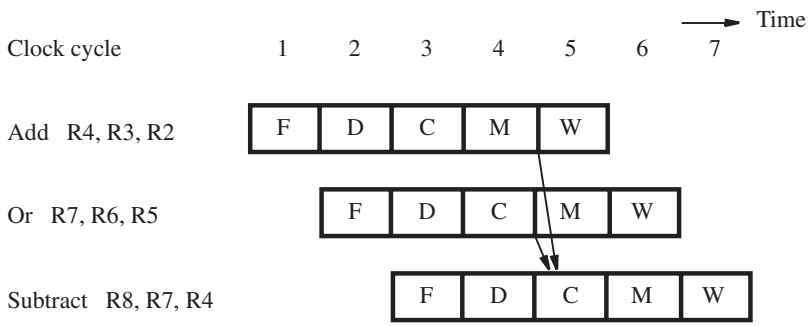


Figure 6.15 Pipelined execution of instructions for Example 6.1.

12 in register RZ. The result from the ALU is $130 - 12 = 118$. This result is available in register RZ during cycle 6. The result of the Or instruction, 130, is in register RY during in cycle 6.

- In cycle 6, the Subtract instruction is in the Memory stage. The unspecified instruction following the Subtract instruction is generating a result in the Compute stage. In cycle 7, the result of the unspecified instruction is in register RZ, and the result of the Subtract instruction is in register RY.

Example 6.2

Problem: Assume that 20 percent of the dynamic count of the instructions executed for a program are branch instructions. There are no pipeline stalls due to data dependencies. Static branch prediction is used with a not-taken assumption.

(*a*) Determine the execution times for two cases: when 30 percent of the branches are taken, and when 70 percent of the branches are taken.

(*b*) Determine the speedup for one case relative to the other. Express the speedup as a percentage relative to 1.

Solution: Section 6.8.1 describes the calculation of $\delta_{\text{branch\_penalty}}$ to consider the effect of branch penalties.

(*a*) The value of $\delta_{\text{branch\_penalty}}$ is $0.20 \times 0.30 = 0.06$ for the first case and $0.20 \times 0.70 = 0.14$ for the second case. Using $S = 1 + \delta_{\text{branch\_penalty}}$, the execution time for the first case is $(1.06 \times N)/R$ and $(1.14 \times N)/R$ for the second case.

(*b*) Because the execution time for the first case is smaller, the performance improvement as a speedup percentage is

$$\left(\frac{1.14}{1.06} - 1\right) \times 100 = 7.5 \text{ percent}$$

PROBLEMS

6.1 **[M]** Consider the following instructions at the given addresses in the memory:

1000	Add	R3, R2, #20
1004	Subtract	R5, R4, #3
1008	And	R6, R4, #0x3A
1012	Add	R7, R2, R4

Initially, registers R2 and R4 contain 2000 and 50, respectively. These instructions are executed in a computer that has a five-stage pipeline as shown in Figure 6.2. The first instruction is fetched in clock cycle 1, and the remaining instructions are fetched in successive cycles.

(*a*) Draw a diagram similar to Figure 6.1 that represents the flow of the instructions through the pipeline. Describe the operation being performed by each pipeline stage during clock cycles 1 through 8.

(*b*) With reference to Figures 5.8 and 5.9, describe the contents of registers IR, PC, RA, RB, RY, and RZ in the pipeline during cycles 2 to 8.

6.2 **[M]** Repeat Problem 6.1 for the following program:

```
1000    Add         R3, R2, #20
1004    Subtract    R5, R4, #3
1008    And         R6, R3, #0x3A
1012    Add         R7, R2, R4
```

Assume that the pipeline provides forwarding paths to the ALU from registers RY and RZ in Figure 5.8 and that the processor uses forwarding of operands.

6.3 **[M]** Consider the loop in the program of Figure 2.8. Assume it is executed in a five-stage pipeline with forwarding paths to the ALU from registers RY and RZ in Figure 5.8. Assume that the pipeline uses static branch prediction with a not-taken assumption. Draw a diagram similar to Figure 6.1 for the execution of two successive iterations of the loop.

6.4 **[D]** Repeat Problem 6.3, but first reorder the instructions to optimize performance as the compiler would do.

6.5 **[D]** Repeat Problem 6.3 for a pipeline that uses delayed branching with one delay slot. Reorder the instructions as needed to improve performance.

6.6 **[M]** The forwarding path in Figure 6.5 allows the contents of register RZ to be used directly in an ALU operation. The result of that operation is stored in register RZ, replacing its previous contents. This problem involves tracing the contents of register RZ over multiple cycles. Consider the two instructions

```
I1:   Add       R3, R2, R1
I2:   LShiftL   R3, R3, #1
```

While instruction I_1 is being fetched in cycle 1, a previously fetched instruction is performing an ALU operation that gives a result of 17. Then, while instruction I_1 is being decoded in cycle 2, another previously fetched instruction is performing an ALU operation that gives a result of 198. Also during cycle 2, registers R1, R2, and R3 contain the values 30, 100, and 45, respectively. Using this information, draw a timing diagram that shows the contents of register RZ during cycles 2 to 5.

6.7 **[M]** Assume that 20 percent of the dynamic count of the instructions executed for a program are branch instructions. Delayed branching is used, with one delay slot. Assume that there are no stalls caused by other factors. First, derive an expression for the execution time in

cycles if all delay slots are filled with NOP instructions. Then, derive another expression that reflects the execution time with 70 percent of delay slots filled with useful instructions by the optimizing compiler. From these expressions, determine the compiler's contribution to the increase in performance, expressed as a speedup percentage.

6.8 **[D]** Repeat Problem 6.7, but this time for a pipelined processor with two branch delay slots. The output from the optimizing compiler is such that the first delay slot is filled with a useful instruction 70 percent of the time, but the second slot is filled with a useful instruction only 10 percent of the time.

Compare the compiler-optimized execution time for this case with the compiler-optimized execution time for Problem 6.7. Assume that the two processors have the same clock rate. Indicate which processor/compiler combination is faster, and determine the speedup percentage by which it is faster.

6.9 **[D]** Assume that 20 percent of the dynamic count of the instructions executed for a program are branch instructions. Assume further that 75 percent of branches are actually taken. The program is executed in two different processors that have the same clock rate. One uses static branch prediction with the assume-not-taken approach. The other uses dynamic branch prediction based on the states in Figure 6.12*a*. The branch target buffer is used in the manner described in Section 6.6.4.

(*a*) With no pipeline stalls due to other causes, what must be the minimum prediction accuracy for the processor using dynamic branch prediction to perform at least as well as the processor using static branch prediction?

(*b*) If the dynamic prediction accuracy is actually 90 percent, what is the speedup relative to using static prediction?

6.10 **[M]** Additional control logic is required in the pipeline to forward the value of register RZ as shown in Figure 6.5. What specific conditions must this additional logic check to determine the settings of the multiplexers feeding the ALU inputs in the Compute stage of the pipeline?

6.11 **[M]** Repeat Problem 6.10 for the specific conditions related to forwarding of the contents of register RY in Figure 5.8 to the multiplexers feeding the inputs of the ALU.

6.12 **[D]** As a continuation of Problems 6.10 and 6.11, consider the following sequence of instructions:

```
Add       R3, R2, R1
Subtract  R3, R5, R4
Or        R8, R3, #1
```

Describe the manner in which forwarding must be handled for this situation. How should the conditions developed in Problems 6.10 and 6.11 be modified?

6.13 **[M]** Consider a program that consists of four memory-access instructions and four arithmetic instructions. Assume that there are no data dependencies between the instructions. Two versions of this program are executed on the superscalar processor shown in Figure 6.13. The first version has the four memory-access instructions in sequence, followed by the four arithmetic instructions. The second version has the memory-access instructions

interleaved with the arithmetic instructions. Draw two diagrams similar to Figure 6.14 to compare the execution of these two versions of the program.

6.14 **[E]** Assume that a program contains no branch instructions. It is executed on the superscalar processor shown in Figure 6.13. What is the best execution time in cycles that can be expected if the mix of instructions consists of 75 percent arithmetic instructions and 25 percent memory-access instructions? How does this time compare to the best execution time on the simpler processor in Figure 6.2 using the same clock?

6.15 **[M]** Repeat Problem 6.14 to find the best possible execution times for the processors in Figures 6.2 and 6.13, assuming that the mix of instructions consists of 15 percent branch instructions that are never taken, 65 percent arithmetic instructions, and 20 percent memory-access instructions. Assume a prediction accuracy of 100 percent for all branch instructions.

6.16 **[E]** Consider a processor that uses the branch prediction scheme represented in Figure 6.12*b*. The instruction set for the processor is enhanced with a feature that enables the compiler to specify the initial prediction state as either LT or LNT for each branch instruction. This initial state is used by the processor at execution time when information about the branch instruction is not found in the branch target buffer. Discuss how the compiler should use this feature when generating code for the following cases:

(*a*) A loop with a conditional branch instruction at the end to branch to the start of the loop

(*b*) A loop with a conditional branch at the beginning of the loop to exit the loop, and an unconditional branch at the end of the loop to branch to the start

6.17 **[M]** Assume that a processor has the feature described in Problem 6.16 for specifying the initial prediction state for branch instructions. Consider a statement of the form

IF A>B THEN A = A + 1 ELSE B = B + 1

(*a*) Generate assembly-language code for the statement above.

(*b*) In the absence of any other information, discuss how the compiler should specify the initial prediction state for the branch instructions in the assembly code.

(*c*) A study of the execution behavior of the program containing the above statement reveals that the value of variable A is often larger than the value of variable B. If this information is made available to the compiler, discuss how it would influence the initial prediction state for the branch instructions.

6.18 **[M]** Consider a statement of the form

IF A>B THEN A = A + 1 ELSE B = B + 1

(*a*) Consider a processor that has the pipelined organization shown in Figure 6.2, with static branch prediction that uses a not-taken assumption. Write assembly-language code for the statement above. Draw diagrams similar to Figure 6.1 to show the pipelined execution of the instructions for different branch decisions and determine the execution times in cycles.

(*b*) Now assume that delayed branching is used. Write assembly-language code for the statement above. Draw diagrams to show the pipelined execution of the instructions for different branch decisions and compare the execution times in cycles with the times for the previous case.

REFERENCES

1. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 4th edition, Morgan Kaufmann, Burlington, Massachusetts, 2009.
2. J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, McGraw-Hill, New York, 2005.