In this and the next chapter we focus on the processing unit, which executes machine instructions and coordinates the activities of other units. This unit is often called the *Instruction Set Processor* (ISP), or simply the *processor*. We examine its internal structure and how it performs the tasks of fetching, decoding, and executing instructions of a program. The processing unit used to be called the *central processing unit* (CPU). The term "central" is less appropriate today because many modern computer systems include several processing units.

The organization of processors has evolved over the years, driven by developments in technology and the need to provide high performance. A common strategy in the development of high-performance processors is to make various functional units operate in parallel as much as possible. High-performance processors have a pipelined organization where the execution of one instruction is started before the execution of the preceding instruction is completed. In another approach, known as superscalar operation, several instructions are fetched and executed at the same time. Pipelining and superscalar architectures are discussed in Chapter 8. In this chapter, we concentrate on the basic ideas that are common to all processors.

A typical computing task consists of a series of steps specified by a sequence of machine instructions that constitute a program. An instruction is executed by carrying out a sequence of more rudimentary operations. These operations and the means by which they are controlled are the main topic of this chapter.

## 7.1 SOME FUNDAMENTAL CONCEPTS

To execute a program, the processor fetches one instruction at a time and performs the operations specified. Instructions are fetched from successive memory locations until a branch or a jump instruction is encountered. The processor keeps track of the address of the memory location containing the next instruction to be fetched using the program counter, PC. After fetching an instruction, the contents of the PC are updated to point to the next instruction in the sequence. A branch instruction may load a different value into the PC.

Another key register in the processor is the instruction register, IR. Suppose that each instruction comprises 4 bytes, and that it is stored in one memory word. To execute an instruction, the processor has to perform the following three steps:

1. Fetch the contents of the memory location pointed to by the PC. The contents of this location are interpreted as an instruction to be executed. Hence, they are loaded into the IR. Symbolically, this can be written as

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

2. Assuming that the memory is byte addressable, increment the contents of the PC by 4, that is,

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

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

In cases where an instruction occupies more than one word, steps 1 and 2 must be repeated as many times as necessary to fetch the complete instruction. These two steps are usually referred to as the *fetch phase;* step 3 constitutes the *execution phase.*

To study these operations in detail, we first need to examine the internal organization of the processor. The main building blocks of a processor were introduced in Figure 1.2. They can be organized and interconnected in a variety of ways. We will start with a very simple organization. Later in this chapter and in Chapter 8 we will present more complex structures that provide high performance. Figure 7.1 shows an organization

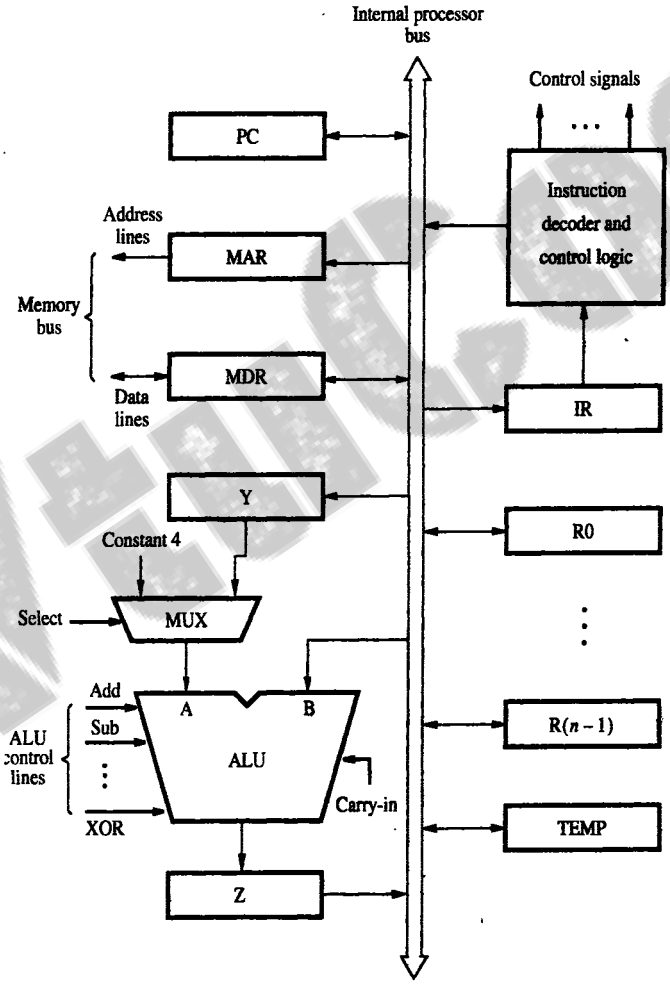



**Figure 7.1** Single-bus organization of the datapath inside a processor.

in which the arithmetic and logic unit (ALU) and all the registers are interconnected via a single common bus. This bus is internal to the processor and should not be confused with the external bus that connects the processor to the memory and I/O devices.

The data and address lines of the external memory bus are shown in Figure 7.1 connected to the internal processor bus via the memory data register, MDR, and the memory address register, MAR, respectively. Register MDR has two inputs and two outputs. Data may be loaded into MDR either from the memory bus or from the internal processor bus. The data stored in MDR may be placed on either bus. The input of MAR is connected to the internal bus, and its output is connected to the external bus. The control lines of the memory bus are connected to the instruction decoder and control logic block. This unit is responsible for issuing the signals that control the operation of all the units inside the processor and for interacting with the memory bus.

The number and use of the processor registers R0 through R($n - 1$) vary considerably from one processor to another. Registers may be provided for general-purpose use by the programmer. Some may be dedicated as special-purpose registers, such as index registers or stack pointers. Three registers, Y, Z, and TEMP in Figure 7.1, have not been mentioned before. These registers are transparent to the programmer, that is, the programmer need not be concerned with them because they are never referenced explicitly by any instruction. They are used by the processor for temporary storage during execution of some instructions. These registers are never used for storing data generated by one instruction for later use by another instruction.

The multiplexer MUX selects either the output of register Y or a constant value 4 to be provided as input A of the ALU. The constant 4 is used to increment the contents of the program counter. We will refer to the two possible values of the MUX control input Select as Select4 and SelectY for selecting the constant 4 or register Y, respectively.

As instruction execution progresses, data are transferred from one register to another, often passing through the ALU to perform some arithmetic or logic operation. The instruction decoder and control logic unit is responsible for implementing the actions specified by the instruction loaded in the IR register. The decoder generates the control signals needed to select the registers involved and direct the transfer of data. The registers, the ALU, and the interconnecting bus are collectively referred to as the *datapath*.

With few exceptions, an instruction can be executed by performing one or more of the following operations in some specified sequence:

- Transfer a word of data from one processor register to another or to the ALU
- Perform an arithmetic or a logic operation and store the result in a processor register
- Fetch the contents of a given memory location and load them into a processor register
- Store a word of data from a processor register into a given memory location

We now consider in detail how each of these operations is implemented, using the simple processor model in Figure 7.1.

### 7.1.1 REGISTER TRANSFERS

Instruction execution involves a sequence of steps in which data are transferred from one register to another. For each register, two control signals are used to place the contents of that register on the bus or to load the data on the bus into the register. This is represented symbolically in Figure 7.2. The input and output of register R*i* are connected to the bus via switches controlled by the signals $Ri_{in}$ and $Ri_{out}$, respectively. When $Ri_{in}$ is set to 1, the data on the bus are loaded into R*i*. Similarly, when $Ri_{out}$ is set to 1, the contents of register R*i* are placed on the bus. While $Ri_{out}$ is equal to 0, the bus can be used for transferring data from other registers.

Suppose that we wish to transfer the contents of register R1 to register R4. This can be accomplished as follows:

- Enable the output of register R1 by setting $R1_{out}$ to 1. This places the contents of R1 on the processor bus.
- Enable the input of register R4 by setting $R4_{in}$ to 1. This loads data from the processor bus into register R4.

All operations and data transfers within the processor take place within time periods defined by the *processor clock*. The control signals that govern a particular transfer are asserted at the start of the clock cycle. In our example, $R1_{out}$ and $R4_{in}$ are set to 1. The registers consist of edge-triggered flip-flops. Hence, at the next active edge of the clock, the flip-flops that constitute R4 will load the data present at their inputs. At the same time, the control signals $R1_{out}$ and $R4_{in}$ will return to 0. We will use this simple model of the timing of data transfers for the rest of this chapter. However, we should point out that other schemes are possible. For example, data transfers may use both the rising and falling edges of the clock. Also, when edge-triggered flip-flops are not used, two or more clock signals may be needed to guarantee proper transfer of data. This is known as *multiphase clocking*.

An implementation for one bit of register R*i* is shown in Figure 7.3 as an example. A two-input multiplexer is used to select the data applied to the input of an edge-triggered D flip-flop. When the control input $Ri_{in}$ is equal to 1, the multiplexer selects the data on the bus. This data will be loaded into the flip-flop at the rising edge of the clock. When $Ri_{in}$ is equal to 0, the multiplexer feeds back the value currently stored in the flip-flop.

The Q output of the flip-flop is connected to the bus via a tri-state gate. When $Ri_{out}$ is equal to 0, the gate's output is in the high-impedance (electrically disconnected) state. This corresponds to the open-circuit state of a switch. When $Ri_{out} = 1$, the gate drives the bus to 0 or 1, depending on the value of Q.

### 7.1.2 PERFORMING AN ARITHMETIC OR LOGIC OPERATION

The ALU is a combinational circuit that has no internal storage. It performs arithmetic and logic operations on the two operands applied to its A and B inputs. In Figures 7.1 and 7.2, one of the operands is the output of the multiplexer MUX and the other operand

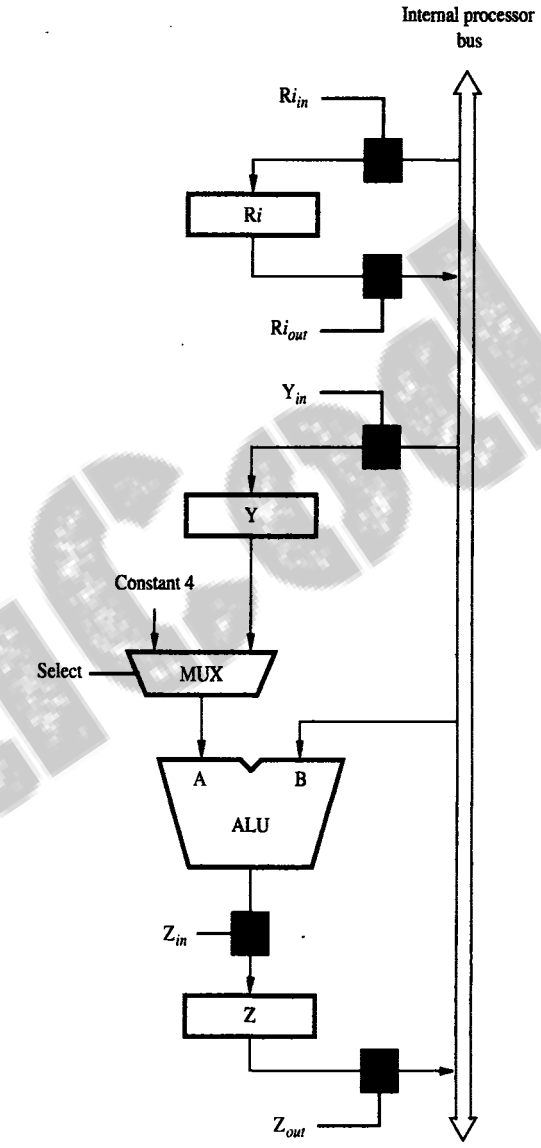



**Figure 7.2** Input and output gating for the registers in Figure 7.1.

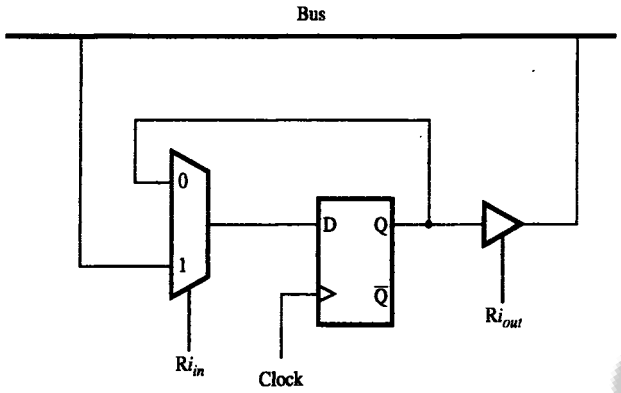



**Figure 7.3** Input and output gating for one register bit.

is obtained directly from the bus. The result produced by the ALU is stored temporarily in register Z. Therefore, a sequence of operations to add the contents of register R1 to those of register R2 and store the result in register R3 is

1. $R1_{out}$, $Y_{in}$
2. $R2_{out}$, SelectY, Add, $Z_{in}$
3. $Z_{out}$, $R3_{in}$

The signals whose names are given in any step are activated for the duration of the clock cycle corresponding to that step. All other signals are inactive. Hence, in step 1, the output of register R1 and the input of register Y are enabled, causing the contents of R1 to be transferred over the bus to Y. In step 2, the multiplexer's Select signal is set to SelectY, causing the multiplexer to gate the contents of register Y to input A of the ALU. At the same time, the contents of register R2 are gated onto the bus and, hence, to input B. The function performed by the ALU depends on the signals applied to its control lines. In this case, the Add line is set to 1, causing the output of the ALU to be the sum of the two numbers at inputs A and B. This sum is loaded into register Z because its input control signal is activated. In step 3, the contents of register Z are transferred to the destination register, R3. This last transfer cannot be carried out during step 2, because only one register output can be connected to the bus during any clock cycle.

In this introductory discussion, we assume that there is a dedicated signal for each function to be performed. For example, we assume that there are separate control signals to specify individual ALU operations, such as Add, Subtract, XOR, and so on. In reality, some degree of encoding is likely to be used. For example, if the ALU can perform eight different operations, three control signals would suffice to specify the required operation. We will discuss the limitations and pros and cons of control signal encoding in Section 7.5.1.

### 7.1.3 FETCHING A WORD FROM MEMORY

To fetch a word of information from memory, the processor has to specify the address of the memory location where this information is stored and request a Read operation. This applies whether the information to be fetched represents an instruction in a program or an operand specified by an instruction. The processor transfers the required address to the MAR, whose output is connected to the address lines of the memory bus. At the same time, the processor uses the control lines of the memory bus to indicate that a Read operation is needed. When the requested data are received from the memory they are stored in register MDR, from where they can be transferred to other registers in the processor.

The connections for register MDR are illustrated in Figure 7.4. It has four control signals: $MDR_{in}$ and $MDR_{out}$ control the connection to the internal bus, and $MDR_{inE}$ and $MDR_{outE}$ control the connection to the external bus. The circuit in Figure 7.3 is easily modified to provide the additional connections. A three-input multiplexer can be used, with the memory bus data line connected to the third input. This input is selected when $MDR_{inE} = 1$. A second tri-state gate, controlled by $MDR_{outE}$ can be used to connect the output of the flip-flop to the memory bus.

During memory Read and Write operations, the timing of internal processor operations must be coordinated with the response of the addressed device on the memory bus. The processor completes one internal data transfer in one clock cycle. The speed of operation of the addressed device, on the other hand, varies with the device. We saw in Chapter 5 that modern processors include a cache memory on the same chip as the processor. Typically, a cache will respond to a memory read request in one clock cycle. However, when a cache miss occurs, the request is forwarded to the main memory, which introduces a delay of several clock cycles. A read or write request may also be intended for a register in a memory-mapped I/O device.

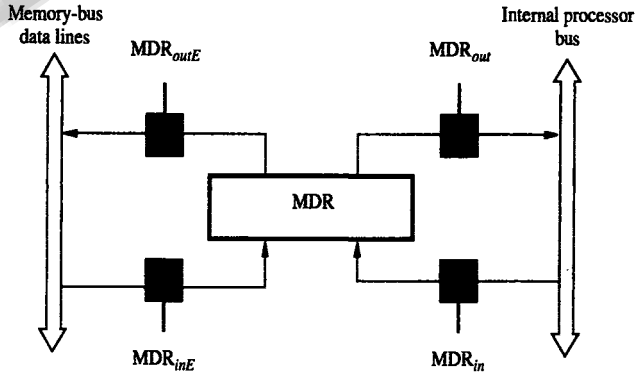



**Figure 7.4** Connection and control signals for register MDR.

Such I/O registers are not cached, so their accesses always take a number of clock cycles.

To accommodate the variability in response time, the processor waits until it receives an indication that the requested Read operation has been completed. We will assume that a control signal called Memory-Function-Completed (MFC) is used for this purpose. The addressed device sets this signal to 1 to indicate that the contents of the specified location have been read and are available on the data lines of the memory bus. (We encountered several examples of such a signal in conjunction with the buses discussed in Chapter 4, such as Slave-ready in Figure 4.25 and TRDY# in Figure 4.41.)

As an example of a read operation, consider the instruction Move (R1),R2. The actions needed to execute this instruction are:

1. MAR ← [R1]
2. Start a Read operation on the memory bus
3. Wait for the MFC response from the memory
4. Load MDR from the memory bus
5. R2 ← [MDR]

These actions may be carried out as separate steps, but some can be combined into a single step. Each action can be completed in one clock cycle, except action 3 which requires one or more clock cycles, depending on the speed of the addressed device.

For simplicity, let us assume that the output of MAR is enabled all the time. Thus, the contents of MAR are always available on the address lines of the memory bus. This is the case when the processor is the bus master. When a new address is loaded into MAR, it will appear on the memory bus at the beginning of the next clock cycle, as shown in Figure 7.5. A Read control signal is activated at the same time MAR is loaded. This signal will cause the bus interface circuit to send a read command, MR, on the bus. With this arrangement, we have combined actions 1 and 2 above into a single control step. Actions 3 and 4 can also be combined by activating control signal $MDR_{inE}$ while waiting for a response from the memory. Thus, the data received from the memory are loaded into MDR at the end of the clock cycle in which the MFC signal is received. In the next clock cycle, $MDR_{out}$ is activated to transfer the data to register R2. This means that the memory read operation requires three steps, which can be described by the signals being activated as follows:

1. $R1_{out}$, $MAR_{in}$, Read
2. $MDR_{inE}$, WMFC
3. $MDR_{out}$, $R2_{in}$

where WMFC is the control signal that causes the processor's control circuitry to wait for the arrival of the MFC signal.

Figure 7.5 shows that $MDR_{inE}$ is set to 1 for exactly the same period as the read command, MR. Hence, in subsequent discussion, we will not specify the value of $MDR_{inE}$ explicitly, with the understanding that it is always equal to MR.

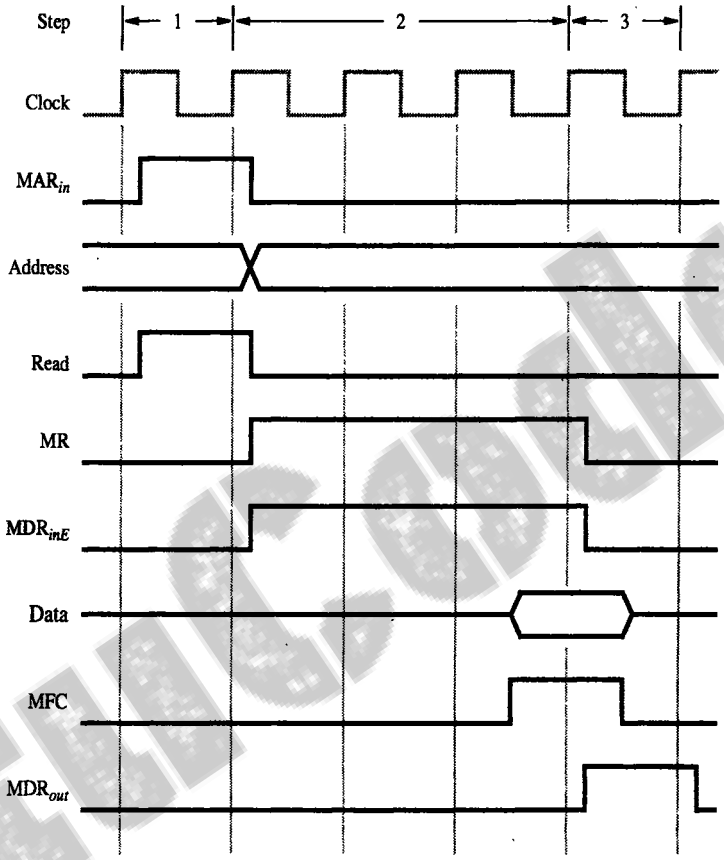



**Figure 7.5** Timing of a memory Read operation.

### 7.1.4 STORING A WORD IN MEMORY

Writing a word into a memory location follows a similar procedure. The desired address is loaded into MAR. Then, the data to be written are loaded into MDR, and a Write command is issued. Hence, executing the instruction Move R2,(R1) requires the following sequence:

1. R1$_{out}$, MAR$_{in}$
2. R2$_{out}$, MDR$_{in}$, Write
3. MDR$_{outE}$, WMFC

As in the case of the read operation, the Write control signal causes the memory bus interface hardware to issue a Write command on the memory bus. The processor remains in step 3 until the memory operation is completed and an MFC response is received.

## 7.2 EXECUTION OF A COMPLETE INSTRUCTION

Let us now put together the sequence of elementary operations required to execute one instruction. Consider the instruction

Add (R3),R1

which adds the contents of a memory location pointed to by R3 to register R1. Executing this instruction requires the following actions:

1. Fetch the instruction.
2. Fetch the first operand (the contents of the memory location pointed to by R3).
3. Perform the addition.
4. Load the result into R1.

Figure 7.6 gives the sequence of control steps required to perform these operations for the single-bus architecture of Figure 7.1. Instruction execution proceeds as follows. In step 1, the instruction fetch operation is initiated by loading the contents of the PC into the MAR and sending a Read request to the memory. The Select signal is set to Select4, which causes the multiplexer MUX to select the constant 4. This value is added to the operand at input B, which is the contents of the PC, and the result is stored in register Z. The updated value is moved from register Z back into the PC during step 2, while waiting for the memory to respond. In step 3, the word fetched from the memory is loaded into the IR.

Steps 1 through 3 constitute the instruction fetch phase, which is the same for all instructions. The instruction decoding circuit interprets the contents of the IR at the beginning of step 4. This enables the control circuitry to activate the control signals for steps 4 through 7, which constitute the execution phase. The contents of register R3 are transferred to the MAR in step 4, and a memory read operation is initiated. Then

| Step | Action |
|---|---|
| 1 | $PC_{out}$, $MAR_{in}$, Read, Select4, Add, $Z_{in}$ |
| 2 | $Z_{out}$, $PC_{in}$, $Y_{in}$, WMFC |
| 3 | $MDR_{out}$, $IR_{in}$ |
| 4 | $R3_{out}$, $MAR_{in}$, Read |
| 5 | $R1_{out}$, $Y_{in}$, WMFC |
| 6 | $MDR_{out}$, SelectY, Add, $Z_{in}$ |
| 7 | $Z_{out}$, $R1_{in}$, End |

**Figure 7.6** Control sequence for execution of the instruction Add (R3),R1.

the contents of R1 are transferred to register Y in step 5, to prepare for the addition operation. When the Read operation is completed, the memory operand is available in register MDR, and the addition operation is performed in step 6. The contents of MDR are gated to the bus, and thus also to the B input of the ALU, and register Y is selected as the second input to the ALU by choosing SelectY. The sum is stored in register Z, then transferred to R1 in step 7. The End signal causes a new instruction fetch cycle to begin by returning to step 1.

This discussion accounts for all control signals in Figure 7.6 except $Y_{in}$ in step 2. There is no need to copy the updated contents of PC into register Y when executing the Add instruction. But, in Branch instructions the updated value of the PC is needed to compute the Branch target address. To speed up the execution of Branch instructions, this value is copied into register Y in step 2. Since step 2 is part of the fetch phase, the same action will be performed for all instructions. This does not cause any harm because register Y is not used for any other purpose at that time.

### 7.2.1 BRANCH INSTRUCTIONS

A branch instruction replaces the contents of the PC with the branch target address. This address is usually obtained by adding an offset X, which is given in the branch instruction, to the updated value of the PC. Figure 7.7 gives a control sequence that implements an unconditional branch instruction. Processing starts, as usual, with the fetch phase. This phase ends when the instruction is loaded into the IR in step 3. The offset value is extracted from the IR by the instruction decoding circuit, which will also perform sign extension if required. Since the value of the updated PC is already available in register Y, the offset X is gated onto the bus in step 4, and an addition operation is performed. The result, which is the branch target address, is loaded into the PC in step 5.

The offset X used in a branch instruction is usually the difference between the branch target address and the address immediately following the branch instruction.

| Step | Action |
|---|---|
| 1 | $PC_{out}$, $MAR_{in}$, Read, Select4, Add, $Z_{in}$ |
| 2 | $Z_{out}$, $PC_{in}$, $Y_{in}$, WMFC |
| 3 | $MDR_{out}$, $IR_{in}$ |
| 4 | Offset-field-of-$IR_{out}$, Add, $Z_{in}$ |
| 5 | $Z_{out}$, $PC_{in}$, End |

**Figure 7.7** Control sequence for an unconditional Branch instruction.

For example, if the branch instruction is at location 2000 and if the branch target address is 2050, the value of X must be 46. The reason for this can be readily appreciated from the control sequence in Figure 7.7. The PC is incremented during the fetch phase, before knowing the type of instruction being executed. Thus, when the branch address is computed in step 4, the PC value used is the updated value, which points to the instruction following the branch instruction in the memory.

Consider now a conditional branch. In this case, we need to check the status of the condition codes before loading a new value into the PC. For example, for a Branch-on-negative (Branch<0) instruction, step 4 in Figure 7.7 is replaced with

$$\text{Offset-field-of-IR}_{out}, \text{Add}, Z_{in}, \text{If N} = 0 \text{ then End}$$

Thus, if N=0 the processor returns to step 1 immediately after step 4. If N=1, step 5 is performed to load a new value into the PC, thus performing the branch operation.

## 7.3 MULTIPLE-BUS ORGANIZATION

We used the simple single-bus structure of Figure 7.1 to illustrate the basic ideas. The resulting control sequences in Figures 7.6 and 7.7 are quite long because only one data item can be transferred over the bus in a clock cycle. To reduce the number of steps needed, most commercial processors provide multiple internal paths that enable several transfers to take place in parallel.

Figure 7.8 depicts a three-bus structure used to connect the registers and the ALU of a processor. All general-purpose registers are combined into a single block called the *register file*. In VLSI technology, the most efficient way to implement a number of registers is in the form of an array of memory cells similar to those used in the implementation of random-access memories (RAMs) described in Chapter 5. The register file in Figure 7.8 is said to have three ports. There are two outputs, allowing the contents of two different registers to be accessed simultaneously and have their contents placed on buses A and B. The third port allows the data on bus C to be loaded into a third register during the same clock cycle.

Buses A and B are used to transfer the source operands to the A and B inputs of the ALU, where an arithmetic or logic operation may be performed. The result is transferred to the destination over bus C. If needed, the ALU may simply pass one of its two input operands unmodified to bus C. We will call the ALU control signals for such an operation R=A or R=B. The three-bus arrangement obviates the need for registers Y and Z in Figure 7.1.

A second feature in Figure 7.8 is the introduction of the Incrementer unit, which is used to increment the PC by 4. Using the Incrementer eliminates the need to add 4 to the PC using the main ALU, as was done in Figures 7.6 and 7.7. The source for the constant 4 at the ALU input multiplexer is still useful. It can be used to increment other addresses, such as the memory addresses in LoadMultiple and StoreMultiple instructions.

The basic building blocks of a computer are introduced in preceding chapters. In this chapter, we discuss in detail the concept of pipelining, which is used in modern computers to achieve high performance. We begin by explaining the basics of pipelining and how it can lead to improved performance. Then we examine machine instruction features that facilitate pipelined execution, and we show that the choice of instructions and instruction sequencing can have a significant effect on performance. Pipelined organization requires sophisticated compilation techniques, and *optimizing compilers* have been developed for this purpose. Among other things, such compilers rearrange the sequence of operations to maximize the benefits of pipelined execution.

## 8.1 BASIC CONCEPTS

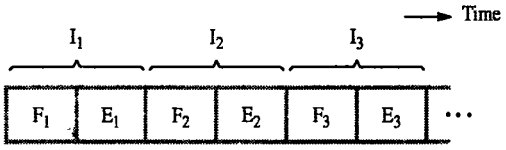
The speed of execution of programs is influenced by many factors. One way to improve performance is to use faster circuit technology to build the processor and the main memory. Another possibility is to arrange the hardware so that more than one operation can be performed at the same time. In this way,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

We have encountered concurrent activities several times before. Chapter 1 introduced the concept of multiprogramming and explained how it is possible for I/O transfers and computational activities to proceed simultaneously. DMA devices make this possible because they can perform I/O transfers independently once these transfers are initiated by the processor.

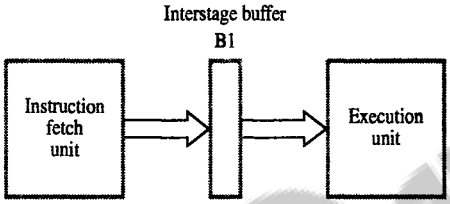
Pipelining is a particularly effective way of organizing concurrent activity in a computer system. The basic idea is very simple. It is frequently encountered in manufacturing plants, where pipelining is commonly known as an assembly-line operation. Readers are undoubtedly familiar with the assembly line used in car manufacturing. The first station in an assembly line may prepare the chassis of a car, 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 car, another group is fitting a car body on the chassis of another car, and yet another group is preparing a new chassis for a third car. It may take days to complete work on a given car, but it is possible to have a new car rolling off the end of the assembly line every few minutes.

Consider how the idea of pipelining can be used in a computer. The processor executes a program by fetching and executing instructions, one after the other. Let $F_i$ and $E_i$ refer to the fetch and execute steps for instruction $I_i$. Execution of a program consists of a sequence of fetch and execute steps, as shown in Figure 8.1*a*.

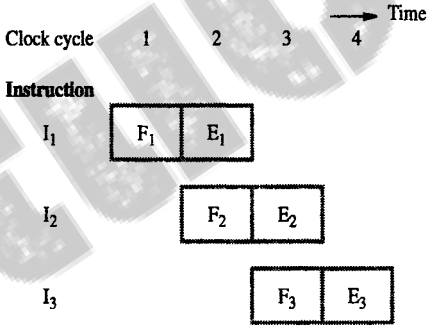
Now consider a computer that has two separate hardware units, one for fetching instructions and another for executing them, as shown in Figure 8.1*b*. The instruction fetched by the fetch unit is deposited in an intermediate storage buffer, B1. This buffer is needed to enable the execution unit to execute the instruction while the fetch unit is fetching the next instruction. The results of execution are deposited in the destination location specified by the instruction. For the purposes of this discussion, we assume that both the source and the destination of the data operated on by the instructions are inside the block labeled "Execution unit."




(a) Sequential execution




(b) Hardware organization




(c) Pipelined execution

**Figure 8.1** Basic idea of instruction pipelining.

The computer is controlled by a clock whose period is such that the fetch and execute steps of any instruction can each be completed in one clock cycle. Operation of the computer proceeds as in Figure 8.1*c*. In the first clock cycle, the fetch unit fetches an instruction $I_1$ (step $F_1$) and stores it in buffer B1 at the end of the clock cycle. In the second clock cycle, the instruction fetch unit proceeds with the fetch operation for instruction $I_2$ (step $F_2$). Meanwhile, the execution unit performs the operation specified by instruction $I_1$, which is available to it in buffer B1 (step $E_1$). By the end of the

second clock cycle, the execution of instruction $I_1$ is completed and instruction $I_2$ is available. Instruction $I_2$ is stored in B1, replacing $I_1$, which is no longer needed. Step $E_2$ is performed by the execution unit during the third clock cycle, while instruction $I_3$ is being fetched by the fetch unit. In this manner, both the fetch and execute units are kept busy all the time. If the pattern in Figure 8.1*c* can be sustained for a long time, the completion rate of instruction execution will be twice that achievable by the sequential operation depicted in Figure 8.1*a*.

In summary, the fetch and execute units in Figure 8.1*b* constitute a two-stage pipeline in which each stage performs one step in processing an instruction. An inter-stage storage buffer, B1, is needed to hold the information being passed from one stage to the next. New information is loaded into this buffer at the end of each clock cycle.

The processing of an instruction need not be divided into only two steps. For example, a pipelined processor may process each instruction in four steps, as follows:

F Fetch: read the instruction from the memory.

D Decode: decode the instruction and fetch the source operand(s).

E Execute: perform the operation specified by the instruction.

W Write: store the result in the destination location.

The sequence of events for this case is shown in Figure 8.2*a*. Four instructions are in progress at any given time. This means that four distinct hardware units are needed, as shown in Figure 8.2*b*. These units must be capable of performing their tasks simultaneously and without interfering with one another. Information is passed from one unit to the next through a storage buffer. As an instruction progresses through the pipeline, all the information needed by the stages downstream must be passed along. For example, during clock cycle 4, the information in the buffers is as follows:

- Buffer B1 holds instruction $I_3$, which was fetched in cycle 3 and is being decoded by the instruction-decoding unit.
- Buffer B2 holds both the source operands for instruction $I_2$ and the specification of the operation to be performed. This is the information produced by the decoding hardware in cycle 3. The buffer also holds the information needed for the write step of instruction $I_2$ (step $W_2$). Even though it is not needed by stage E, this information must be passed on to stage W in the following clock cycle to enable that stage to perform the required Write operation.
- Buffer B3 holds the results produced by the execution unit and the destination information for instruction $I_1$.

### 8.1.1 Role of Cache Memory

Each stage in a pipeline is expected to complete its operation in one clock cycle. Hence, the clock period should be sufficiently long to complete the task being performed in any stage. If different units require different amounts of time, the clock period must allow the longest task to be completed. A unit that completes its task early is idle for the remainder of the clock period. Hence, pipelining is most effective in improving

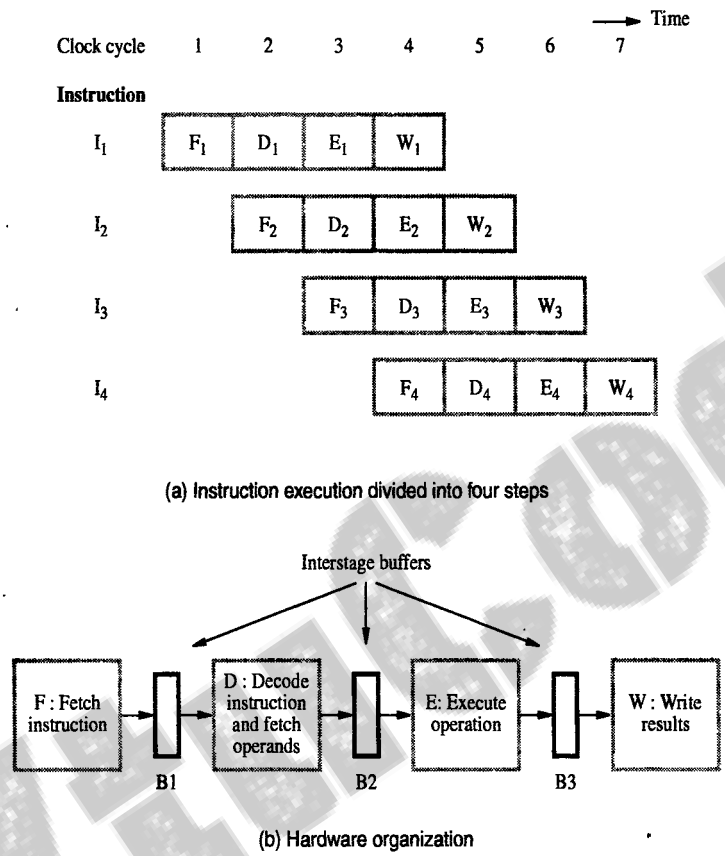



**Figure 8.2** A 4-stage pipeline.

performance if the tasks being performed in different stages require about the same amount of time.

This consideration is particularly important for the instruction fetch step, which is assigned one clock period in Figure 8.2*a*. The clock cycle has to be equal to or greater than the time needed to complete a fetch operation. However, the access time of the main memory may be as much as ten times greater than the time needed to perform basic pipeline stage operations inside the processor, such as adding two numbers. Thus, if each instruction fetch required access to the main memory, pipelining would be of little value.

The use of cache memories solves the memory access problem. In particular, when a cache is included on the same chip as the processor, access time to the cache is usually the same as the time needed to perform other basic operations inside the processor. This

makes it possible to divide instruction fetching and processing into steps that are more or less equal in duration. Each of these steps is performed by a different pipeline stage, and the clock period is chosen to correspond to the longest one.

### 8.1.2 PIPELINE PERFORMANCE

The pipelined processor in Figure 8.2 completes the processing of one instruction in each clock cycle, which means that the rate of instruction processing is four times that of sequential operation. The potential increase in performance resulting from pipelining is proportional to the number of pipeline stages. However, this increase would be achieved only if pipelined operation as depicted in Figure 8.2*a* could be sustained without interruption throughout program execution. Unfortunately, this is not the case.

For a variety of reasons, one of the pipeline stages may not be able to complete its processing task for a given instruction in the time allotted. For example, stage E in the four-stage pipeline of Figure 8.2*b* is responsible for arithmetic and logic operations, and one clock cycle is assigned for this task. Although this may be sufficient for most operations, some operations, such as divide, may require more time to complete. Figure 8.3 shows an example in which the operation specified in instruction $I_2$ requires three cycles to complete, from cycle 4 through cycle 6. Thus, in cycles 5 and 6, the Write stage must be told to do nothing, because it has no data to work with. Meanwhile, the information in buffer B2 must remain intact until the Execute stage has completed its operation. This means that stage 2 and, in turn, stage 1 are blocked from accepting new instructions because the information in B1 cannot be overwritten. Thus, steps $D_4$ and $F_5$ must be postponed as shown.

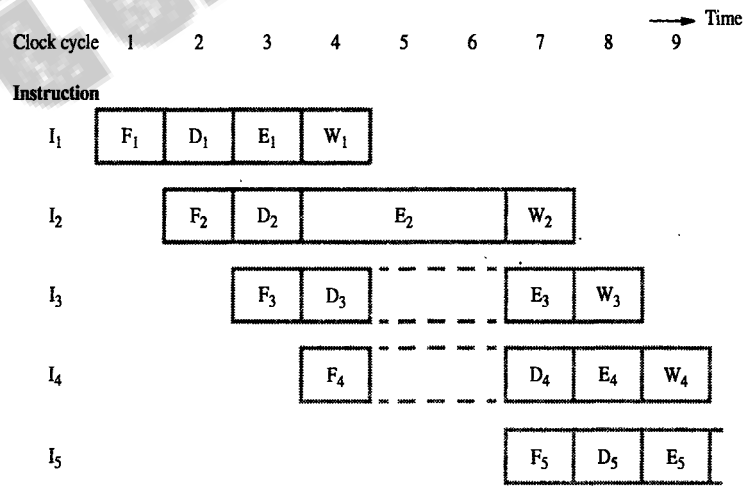
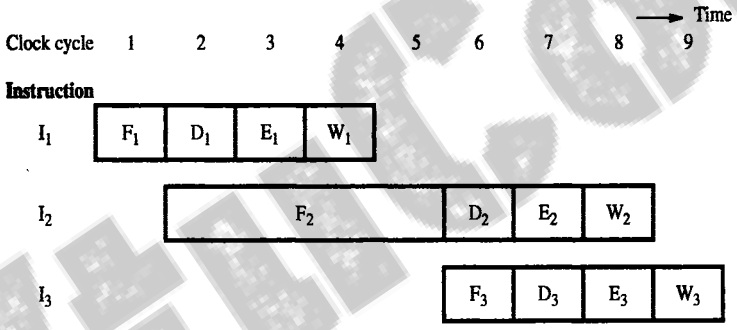



**Figure 8.3** Effect of an execution operation taking more than one clock cycle.

Pipelined operation in Figure 8.3 is said to have been *stalled* for two clock cycles. Normal pipelined operation resumes in cycle 7. Any condition that causes the pipeline to stall is called a *hazard*. We have just seen an example of a *data hazard*. A data hazard is any condition in which either the source or the destination operands of an instruction are not available at the time expected in the pipeline. As a result some operation has to be delayed, and the pipeline stalls.

The pipeline may also be stalled because of a delay in the availability of an instruction. For example, this may be a result of a miss in the cache, requiring the instruction to be fetched from the main memory. Such hazards are often called *control hazards* or *instruction hazards*. The effect of a cache miss on pipelined operation is illustrated in Figure 8.4. Instruction $I_1$ is fetched from the cache in cycle 1, and its execution proceeds normally. However, the fetch operation for instruction $I_2$, which is started in cycle 2, results in a cache miss. The instruction fetch unit must now suspend any further fetch requests and wait for $I_2$ to arrive. We assume that instruction $I_2$ is received and loaded into buffer B1 at the end of cycle 5. The pipeline resumes its normal operation at that point.




(a) Instruction execution steps in successive clock cycles

| Clock cycle | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| **Stage** | | | | | | | | | |
| F: Fetch | $F_1$ | $F_2$ | $F_2$ | $F_2$ | $F_2$ | $F_3$ | | | |
| D: Decode | | $D_1$ | idle | idle | idle | $D_2$ | $D_3$ | | |
| E: Execute | | | $E_1$ | idle | idle | idle | $E_2$ | $E_3$ | |
| W: Write | | | | $W_1$ | idle | idle | idle | $W_2$ | $W_3$ |

(b) Function performed by each processor stage in successive clock cycles

**Figure 8.4** Pipeline stall caused by a cache miss in F2.

An alternative representation of the operation of a pipeline in the case of a cache miss is shown in Figure 8.4*b*. This figure gives the function performed by each pipeline stage in each clock cycle. Note that the Decode unit is idle in cycles 3 through 5, the Execute unit is idle in cycles 4 through 6, and the Write unit is idle in cycles 5 through 7. Such idle periods are called *stalls*. They are also often referred to as *bubbles* in the pipeline. Once created as a result of a delay in one of the pipeline stages, a bubble moves downstream until it reaches the last unit.

A third type of hazard that may be encountered in pipelined operation is known as a *structural hazard*. This is the situation when two instructions require the use of a given hardware resource at the same time. The most common case in which this hazard may arise is in access to memory. One instruction may need to access memory as part of the Execute or Write stage while another instruction is being fetched. If instructions and data reside in the same cache unit, only one instruction can proceed and the other instruction is delayed. Many processors use separate instruction and data caches to avoid this delay.

An example of a structural hazard is shown in Figure 8.5. This figure shows how the load instruction

Load X(R1),R2

can be accommodated in our example 4-stage pipeline. The memory address, X+[R1], is computed in step $E_2$ in cycle 4, then memory access takes place in cycle 5. The operand read from memory is written into register R2 in cycle 6. This means that the execution step of this instruction takes two clock cycles (cycles 4 and 5). It causes the pipeline to stall for one cycle, because both instructions $I_2$ and $I_3$ require access to the register file in cycle 6. Even though the instructions and their data are all available, the pipeline is

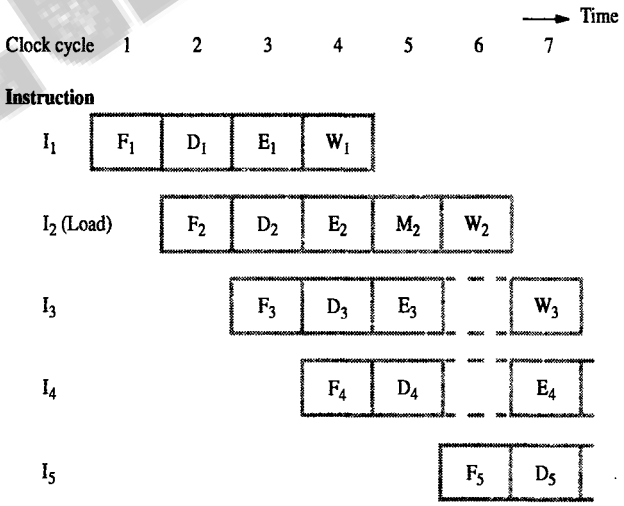



**Figure 8.5** Effect of a Load instruction on pipeline timing.

stalled because one hardware resource, the register file, cannot handle two operations at once. If the register file had two input ports, that is, if it allowed two simultaneous write operations, the pipeline would not be stalled. In general, structural hazards are avoided by providing sufficient hardware resources on the processor chip.

It is important to understand that pipelining does not result in individual instructions being executed faster; rather, it is the throughput that increases, where throughput is measured by the rate at which instruction execution is completed. Any time one of the stages in the pipeline cannot complete its operation in one clock cycle, the pipeline stalls, and some degradation in performance occurs. Thus, the performance level of one instruction completion in each clock cycle is actually the upper limit for the throughput achievable in a pipelined processor organized as in Figure 8.2*b*.

An important goal in designing processors is to identify all hazards that may cause the pipeline to stall and to find ways to minimize their impact. In the following sections we discuss various hazards, starting with data hazards, followed by control hazards. In each case we present some of the techniques used to mitigate their negative effect on performance. We return to the issue of performance assessment in Section 8.8.

## 8.2 DATA HAZARDS

A data hazard is a situation in which the pipeline is stalled because the data to be operated on are delayed for some reason, as illustrated in Figure 8.3. We will now examine the issue of availability of data in some detail.

Consider a program that contains two instructions, $I_1$ followed by $I_2$. When this program is executed in a pipeline, the execution of $I_2$ can begin before the execution of $I_1$ is completed. This means that the results generated by $I_1$ may not be available for use by $I_2$. We must ensure that the results obtained when instructions are executed in a pipelined processor are identical to those obtained when the same instructions are executed sequentially. The potential for obtaining incorrect results when operations are performed concurrently can be demonstrated by a simple example. Assume that $A = 5$, and consider the following two operations:

$$A \leftarrow 3 + A$$

$$B \leftarrow 4 \times A$$

When these operations are performed in the order given, the result is $B = 32$. But if they are performed concurrently, the value of A used in computing B would be the original value, 5, leading to an incorrect result. If these two operations are performed by instructions in a program, then the instructions must be executed one after the other, because the data used in the second instruction depend on the result of the first instruction. On the other hand, the two operations

$$A \leftarrow 5 \times C$$

$$B \leftarrow 20 + C$$

can be performed concurrently, because these operations are independent.