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

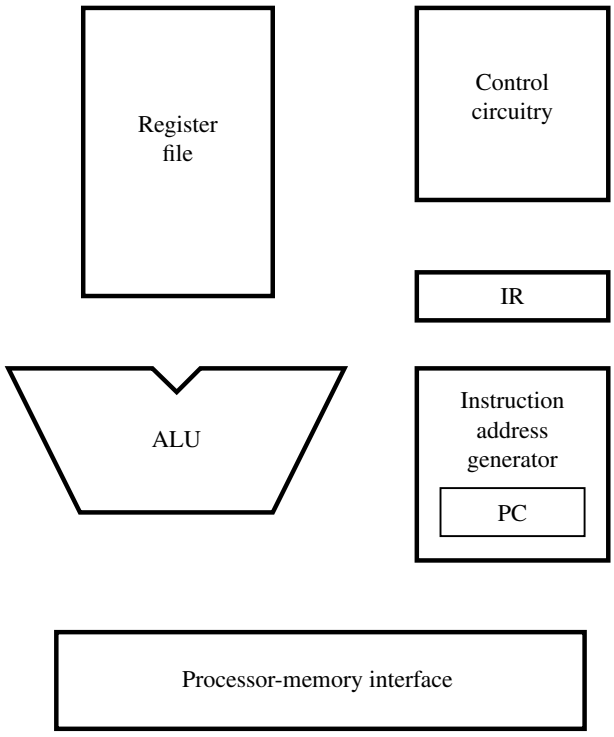


Figure 5.1 Main hardware components of a processor.

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.

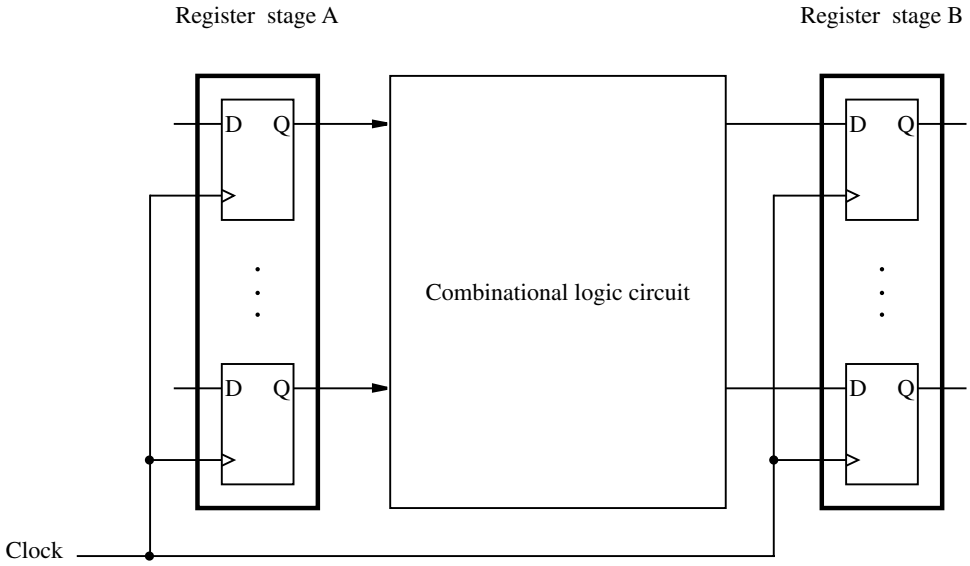


Figure 5.2 Basic structure for data processing.

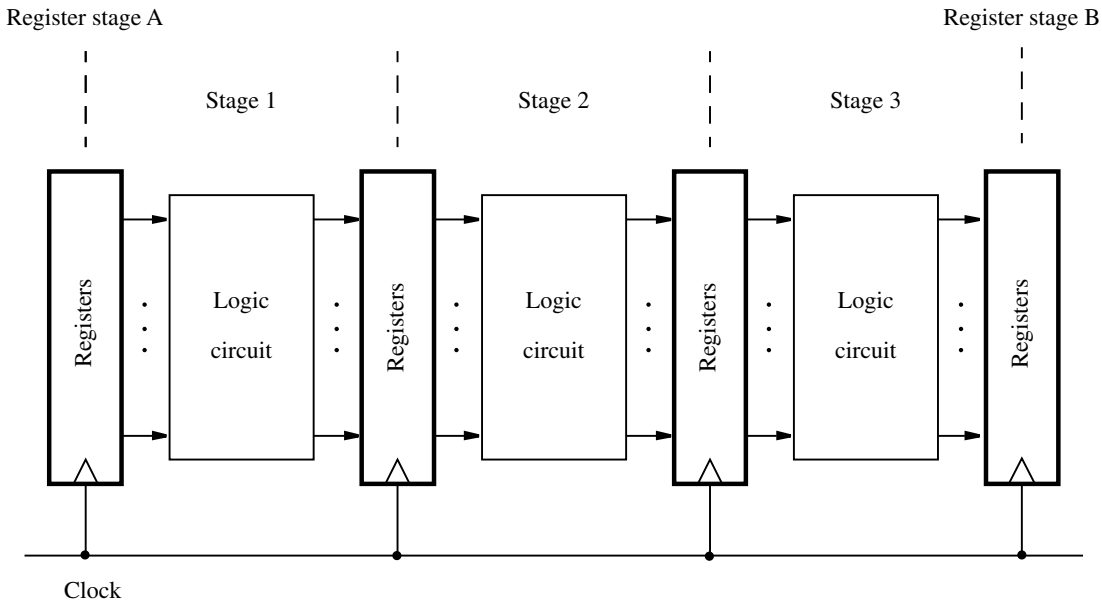


Figure 5.3 A hardware structure with multiple stages.

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.

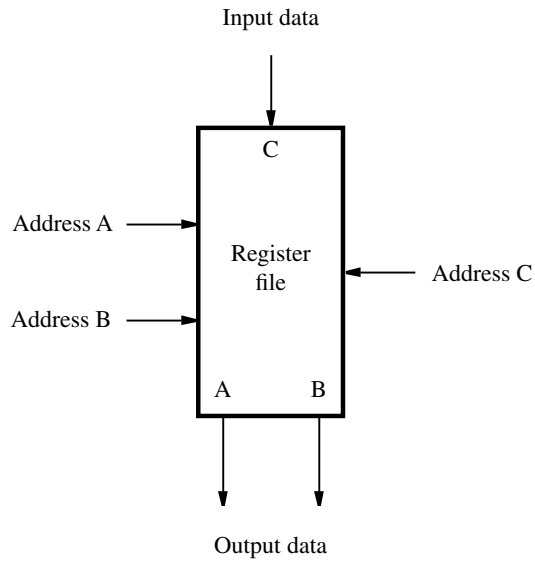
5.3 HARDWARE COMPONENTS

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.

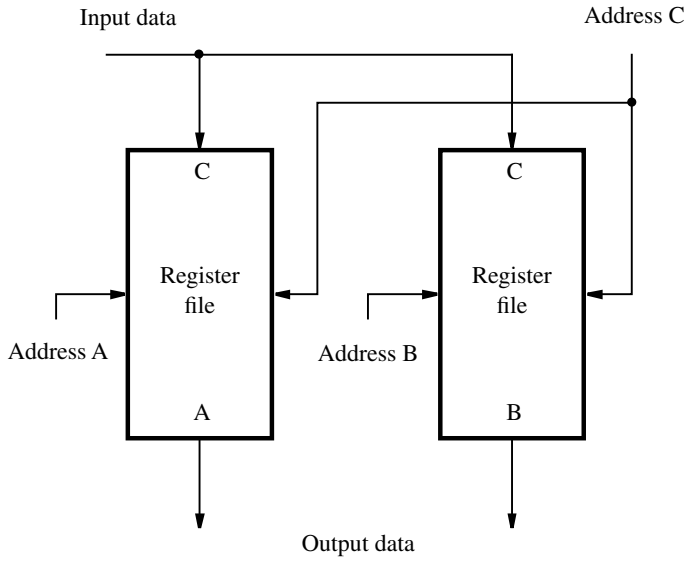
5.3.1 REGISTER FILE

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

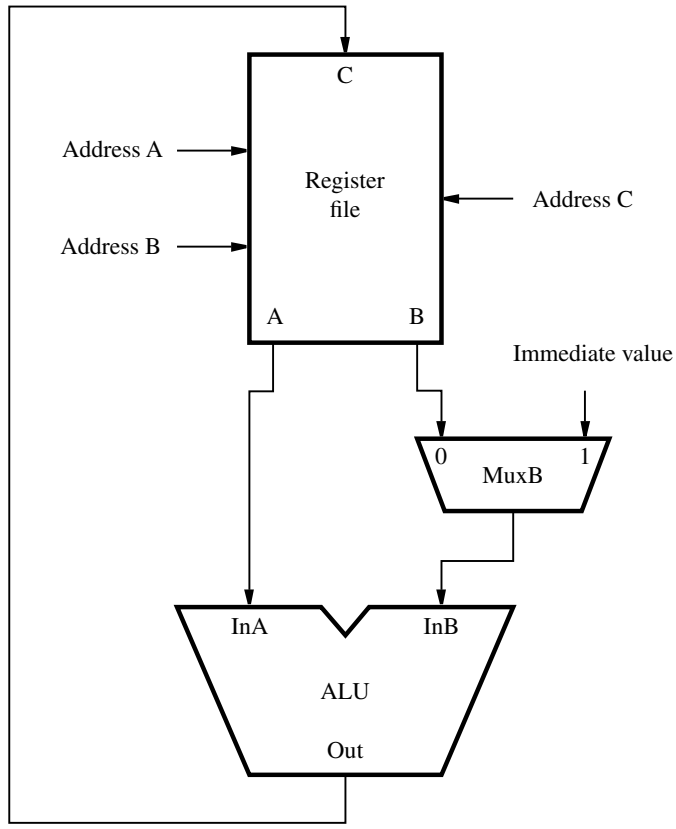


Figure 5.6 Conceptual view of the hardware needed for computation.

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.

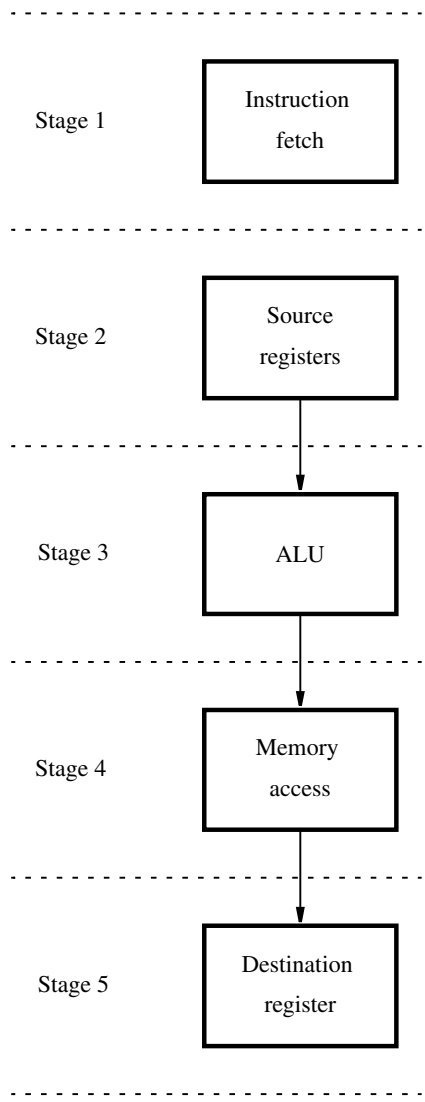


Figure 5.7 A five-stage organization.

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.

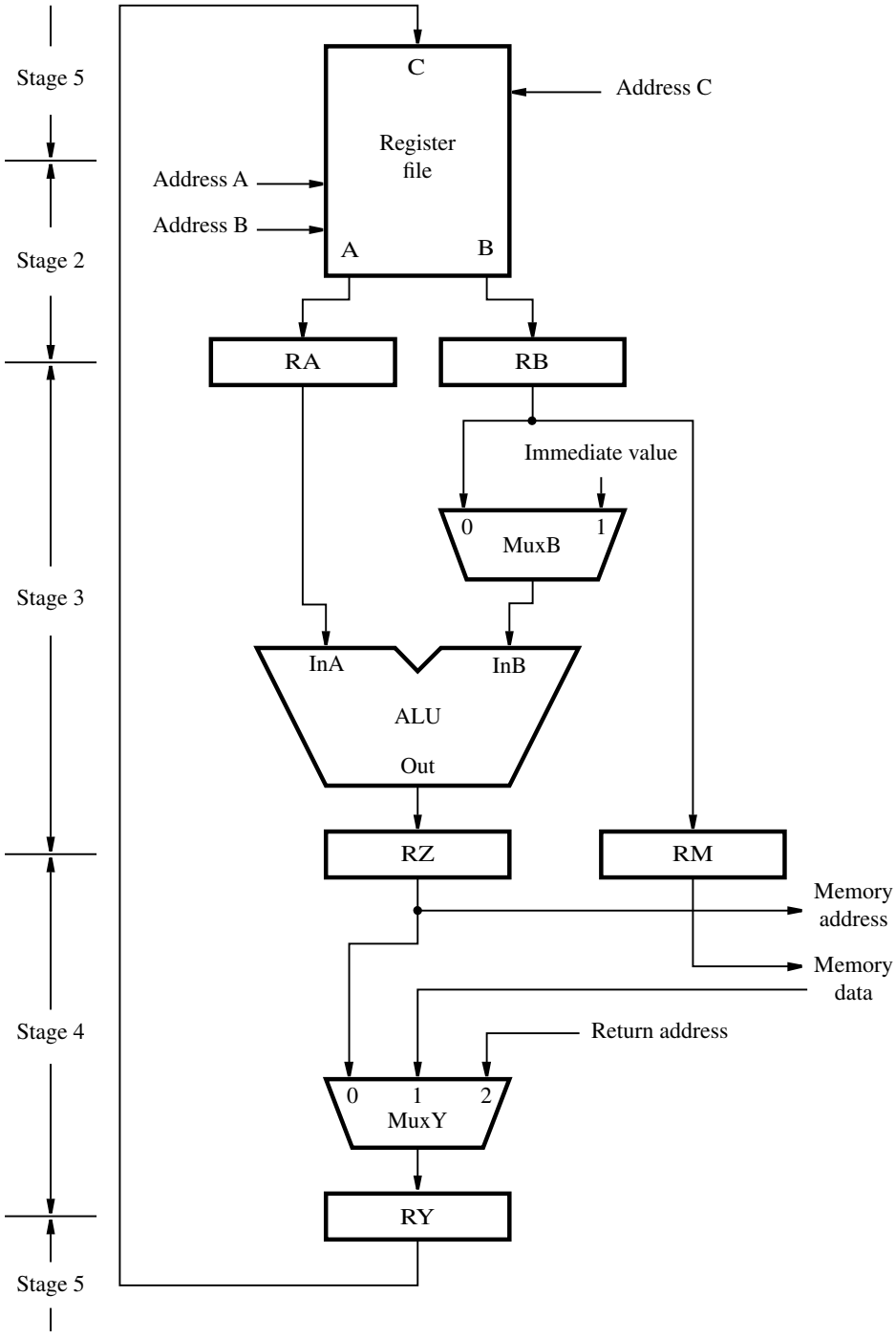


Figure 5.8 Datapath in a processor.

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

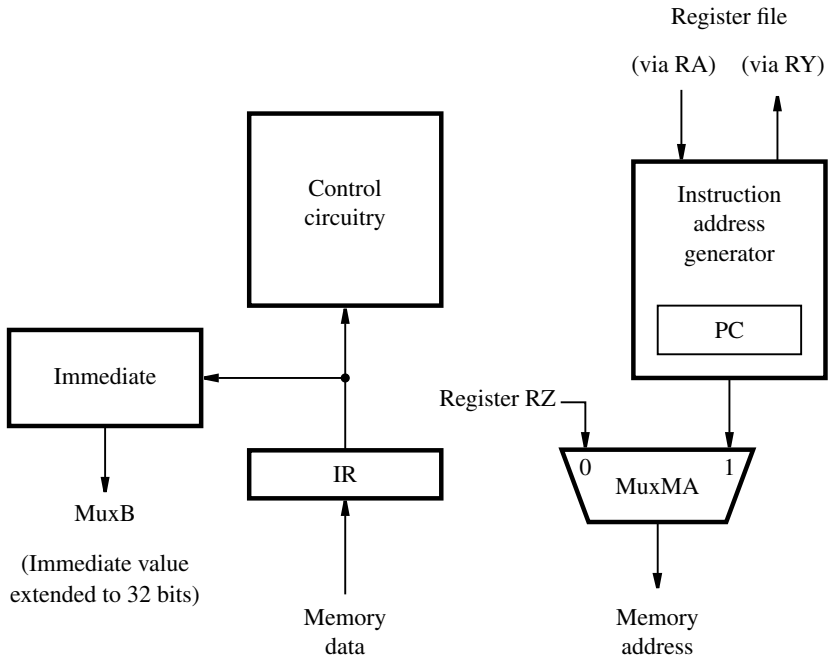


Figure 5.9 Instruction fetch section of Figure 5.7.

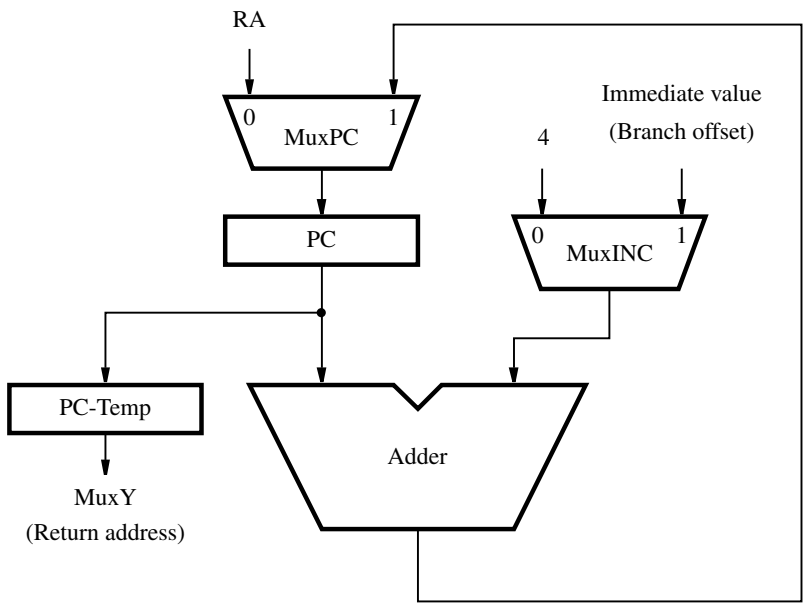


Figure 5.10 Instruction address generator.

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

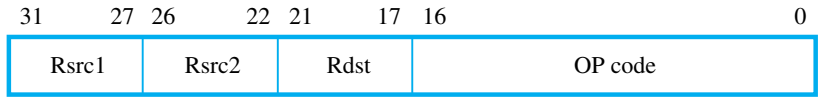
We now examine the process of fetching and executing instructions in more detail, using the datapath in Figure 5.8. Consider again the instruction

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

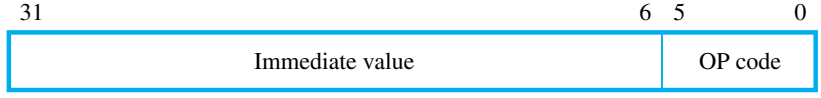
Step	Action
1	Memory address ← [PC], Read memory, IR ← Memory data, PC ← [PC] + 4
2	Decode instruction, RA ← [R4], RB ← [R5]
3	RZ ← [RA] + [RB]
4	RY ← [RZ]
5	R3 ← [RY]

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.

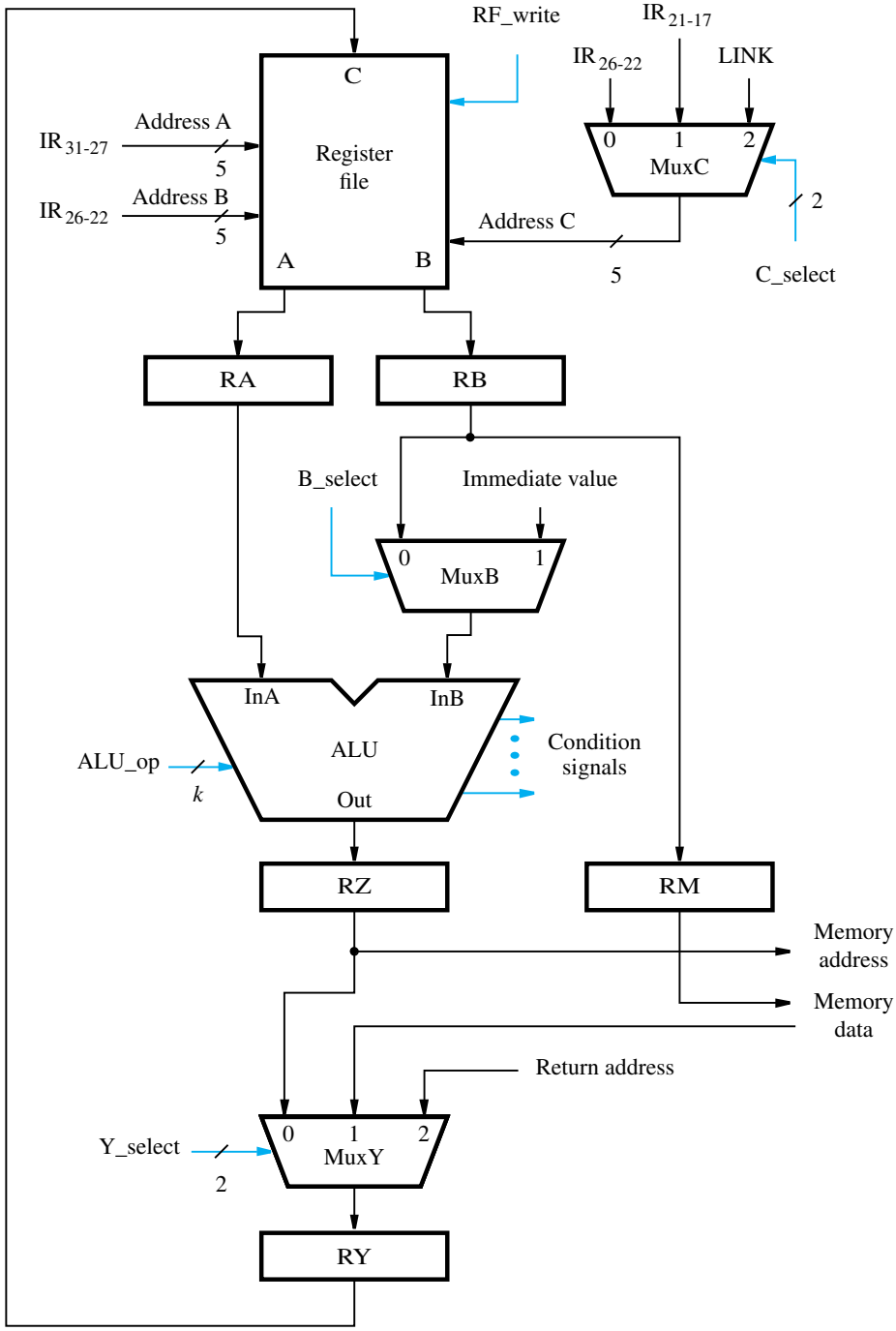


Figure 5.18 Control signals for the datapath.

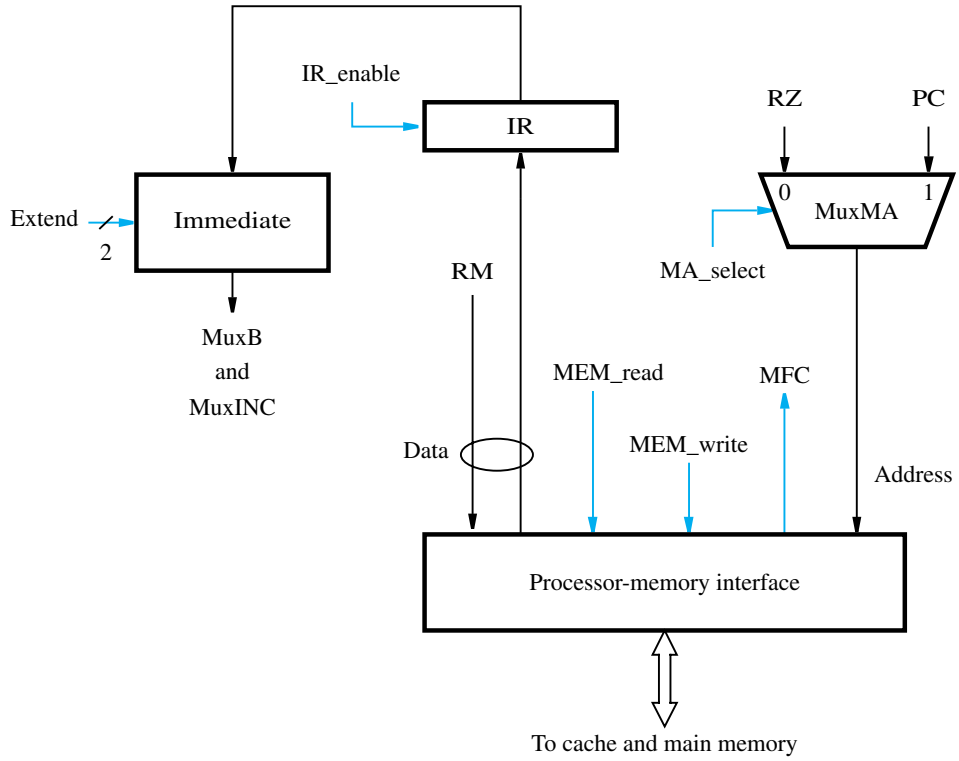


Figure 5.19 Processor-memory interface and IR control signals.

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.

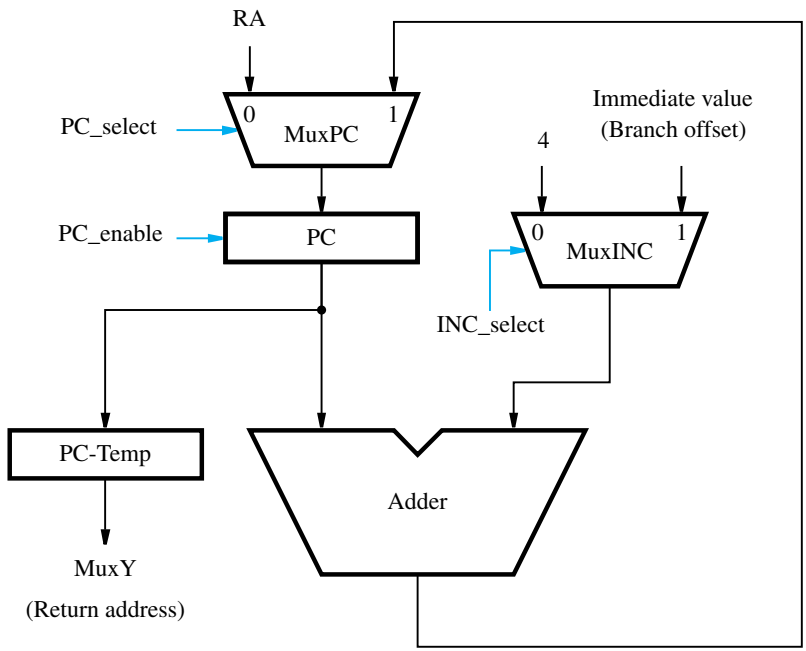


Figure 5.20 Control signals for the instruction address generator.

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