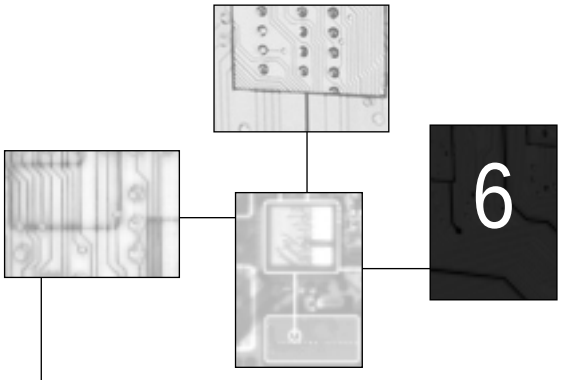


6 CPU Design

As we saw in Chapter 4, a CPU contains three main sections: the register section, the arithmetic/logic unit (ALU), and the control unit. These sections work together to perform the sequences of micro-operations needed to perform the fetch, decode, and execute cycles of every instruction in the CPU's instruction set. In this chapter we examine the process of designing a CPU in detail.

To demonstrate this design process, we present the designs of two CPUs, each implemented using **hardwired control**. (A different type of control, which uses a **microsequencer**, is examined in Chapter 7.) We start by analyzing the applications for the CPU. For instance, will it be used to control a microwave oven or a personal computer? Once we know its application, we can determine the types of programs it will run, and from there we can develop the instruction set architecture (ISA) for the CPU. Next, we determine the other registers we need to include within the CPU that are not a part of its ISA. We then design the state diagram for the CPU, along with the micro-operations needed to fetch, decode, and execute each instruction. Once this is done, we define the internal data paths and the necessary control signal. Finally, we design the control unit, the logic that generates the control signals and causes the operations to occur.

In this chapter we present the complete design of two simple CPUs, along with an analysis of their shortcomings. We also look at the internal architecture of the Intel 8085 microprocessor, whose instruction set architecture was introduced in Chapter 3.

6.1 Specifying a CPU

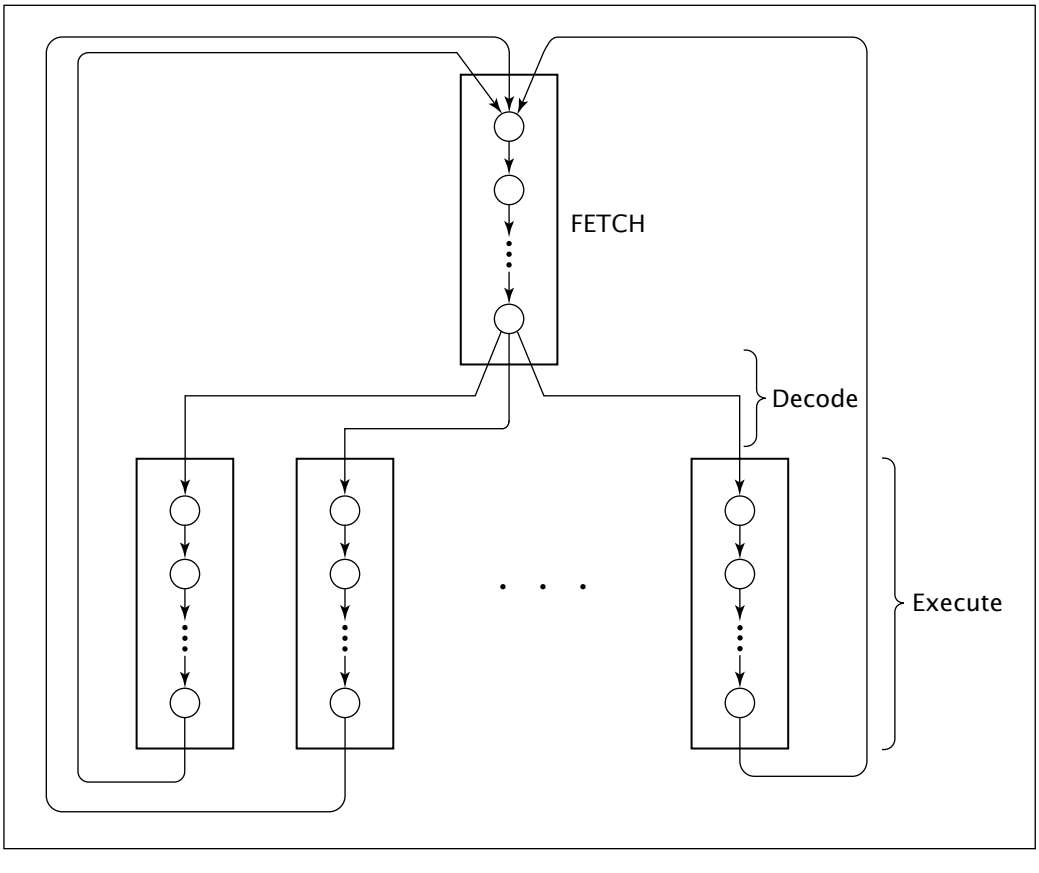
The first step in designing a CPU is to determine its applications. We don't need anything as complicated as an Itanium microprocessor to control a microwave oven; a simple 4-bit processor would

be powerful enough to handle this job. However, the same 4-bit processor would be woefully inadequate to power a personal computer. The key is to match the capabilities of the CPU to the tasks it will perform.

Once we have determined the tasks a CPU will perform, we must design an instruction set architecture capable of handling these tasks. We select the instructions a programmer could use to write the application programs and the registers these instructions will use.

After this is done, we design the state diagram for the CPU. We show the micro-operations performed during each state and the conditions that cause the CPU to go from one state to another. A CPU is just a complex finite state machine. By specifying the states and their micro-operations, we specify the steps the CPU must perform in order to fetch, decode, and execute every instruction in its instruction set.

Figure 6.1
Generic CPU state diagram



In general, a CPU performs the following sequence of operations:

- **Fetch cycle**: Fetch an instruction from memory, then go to the decode cycle.
- **Decode cycle**: Decode the instruction—that is, determine which instruction has been fetched—then go to the execute cycle for that instruction.
- **Execute cycle**: Execute the instruction, then go to the fetch cycle and fetch the next instruction.

A generic state diagram is shown in Figure 6.1 on page 215. Note that the decode cycle does not have any states. Rather, the decode cycle is actually the multiple branches from the end of the fetch routine to each individual execute routine.

6.2 Design and Implementation of a Very Simple CPU

In this section, we specify and design a Very Simple CPU, probably the simplest CPU you will ever encounter. This CPU isn't very practical, but it is not meant to be. The sole application of this CPU is to serve as an instructional aid, to illustrate the design process without burdening the reader with too many design details. In the next section, we design a more complex CPU, which builds on the design methods presented here.

6.2.1 Specifications for a Very Simple CPU

To illustrate the CPU design process, consider this small and somewhat impractical CPU. It can access 64 bytes of memory, each byte being 8 bits wide. The CPU does this by outputting a 6-bit address on its output pins A[5..0] and reading in the 8-bit value from memory on its inputs D[7..0].

This CPU will have only one programmer-accessible register, an 8-bit **accumulator** labeled *AC*. It has only four instructions in its instruction set, as shown in Table 6.1.

Table 6.1
Instruction set for the Very Simple CPU

Instruction	Instruction Code	Operation
ADD	00AAAAAA	AC←AC + M[AAAAAA]
AND	01AAAAAA	AC←AC ^ M[AAAAAA]
JMP	10AAAAAA	GOTO AAAAAA
INC	11XXXXXX	AC←AC + 1

As noted earlier, this is a fairly impractical CPU for several reasons. For example, although it can perform some computations, it cannot output the results.

In addition to *AC*, this CPU needs several additional registers to perform the internal operations necessary to fetch, decode, and execute instructions. The registers in this CPU are fairly standard and are found in many CPUs; their sizes vary depending on the CPU in which they are used. This CPU contains the following registers:

- A 6-bit **address register**, *AR*, which supplies an address to memory via A[5..0]
- A 6-bit **program counter**, *PC*, which contains the address of the next instruction to be executed
- An 8-bit **data register**, *DR*, which receives instructions and data from memory via D[7..0]
- A 2-bit **instruction register**, *IR*, which stores the opcode portion of the instruction code fetched from memory

A CPU is just a complex finite state machine, and that dictates the approach we take in designing this CPU. First, we design the state diagram for the CPU. Then we design both the necessary data paths and the control logic to realize the finite state machine, thus implementing the CPU.

6.2.2 Fetching Instructions from Memory

Before the CPU can execute an instruction, it must fetch the instruction from memory. To do this, the CPU performs the following sequence of actions.

1. Send the address to memory by placing it on the address pins A[5..0].
2. After allowing memory enough time to perform its internal decoding and to retrieve the desired instruction, send a signal to memory so that it outputs the instruction on its output pins. These pins are connected to D[7..0] of the CPU. The CPU reads this data in from those pins.

The address of the instruction to be fetched is stored in the program counter. Since A[5..0] receive their values from the address register, the first step is accomplished by copying the contents of *PC* to *AR*. Thus the first state of the fetch cycle is

FETCH1: $AR \leftarrow PC$

Next, the CPU must read the instruction from memory. The CPU must assert a **READ** signal, which is output from the CPU to memory, to cause memory to output the data to D[7..0]. At the same time, the CPU must read the data in and store it in *DR*, since this is the only

register used to access memory. By waiting until one state after FETCH1, the CPU gives memory the time to access the requested data (which is an instruction in this case). The net result is, at first,

FETCH2: $DR \leftarrow M$

In fact, there is another operation that will be performed here. We must also increment the program counter, so FETCH2 should actually be as follows:

FETCH2: $DR \leftarrow M$, $PC \leftarrow PC + 1$

See Practical Perspective: Why a CPU Increments PC During the Fetch Cycle for the reasoning behind this.

Finally, there are two other things that the CPU will do as part of the fetch routine. First, it copies the two high-order bits of *DR* to *IR*. As shown in Table 6.1, these two bits indicate which instruction is to be executed. As we will see in the design of the control logic, it is necessary to save this value in a location other than *DR* so it will be available to the control unit. Also, the CPU copies the six low-order bits of *DR* to *AR* during the fetch routine. For the ADD and AND instructions, these bits contain the memory address of one of the operands for the instruction. Moving the address to *AR* here will result in one less state in the execute routines for these instructions. For the other two instructions, it will not cause a problem. They do not need to access memory again, so they just won't use the value loaded into *AR*. Once they return to the FETCH routine, FETCH1 will load *PC* into *AR*, over-

PRACTICAL PERSPECTIVE: Why a CPU Increments PC During the Fetch Cycle

To see why a CPU increments the program counter during FETCH2, consider what would happen if it did not increment *PC*. For example, assume that the CPU fetched an instruction from location 10. In FETCH1, it would perform the operation $AR \leftarrow PC$ (which has the value 10). In FETCH2, it would fetch the instruction from memory location 10 and store it in *DR*. Presumably the CPU would then decode the instruction and execute it, and then return to FETCH1 to fetch the next instruction. However, *PC* still contains the value 10, so the CPU would continuously fetch, decode, and execute the same instruction!

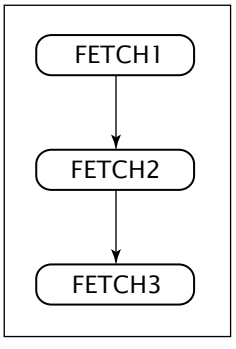
The next instruction to be executed is stored in the next location, 11. The CPU must increment the *PC* some time before it returns to the fetch routine. To make this happen, the designer has two options: have every instruction increment the *PC* as part of its execute routine, or increment the *PC* once during the fetch routine. The latter is much easier to implement, so CPUs take this approach.

writing the unused value. These two operations can be performed in one state as

FETCH3: $IR \leftarrow DR[7..6]$, $AR \leftarrow DR[5..0]$

The state diagram for the fetch cycle is shown in Figure 6.2.

Figure 6.2
Fetch cycle for the Very Simple CPU



6.2.3 Decoding Instructions

After the CPU has fetched an instruction from memory, it must determine which instruction it has fetched so that it may invoke the correct execute routine. The state diagram represents this as a series of branches from the end of the fetch routine to the individual execute routines. For this CPU, there are four instructions and thus four execute routines. The value in *IR*, 00, 01, 10, or 11, determines which execute routine is invoked. The state diagram for the fetch and decode cycles is shown in Figure 6.3 on page 220.

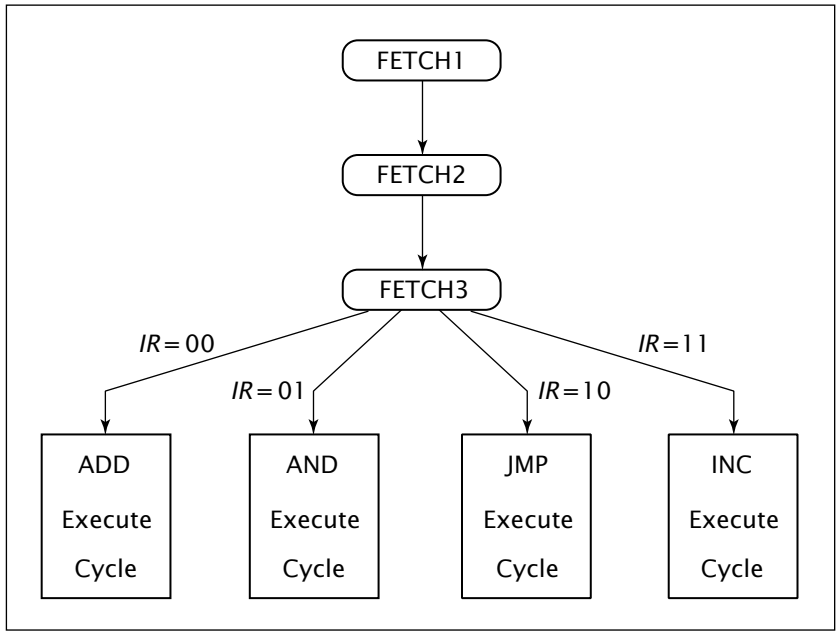
6.2.4 Executing Instructions

To complete the state diagram for this CPU, we must develop the state diagram for each execute routine. Now we design the portion of the state diagram for each execute routine and the overall design for the CPU. The state diagrams for the individual execute routines are fairly simple, so they are only included in the diagram of the finite state machine for the entire CPU.

6.2.4.1 ADD Instruction

In order to perform the ADD instruction, the CPU must do two things. First, it must fetch one operand from memory. Then it must add this operand to the current contents of the accumulator and store the result back into the accumulator.

Figure 6.3
Fetch and decode cycles for the Very Simple CPU



To fetch the operand from memory, the CPU must first make its address available via A[5..0], just as it did to fetch the instruction from memory. This is done by moving the address into *AR*. However, this was already done in FETCH3, so the CPU can simply read the value in immediately. (This is the time savings mentioned earlier.) Thus,

$$\text{ADD1:} \quad DR \leftarrow M$$

Now that both operands are within the CPU, it can perform the actual addition in one state.

$$\text{ADD2:} \quad AC \leftarrow AC + DR$$

These two operations comprise the entire execute cycle for the ADD instruction. At this point, the ADD execute cycle would branch back to the fetch cycle to begin fetching the next instruction.

6.2.4.2 AND Instruction

The execute cycle for the AND instruction is virtually the same as that for the ADD instruction. It must fetch an operand from memory, making use of the address copied to *AR* during FETCH3. However, instead of adding the two values, it must logically AND the two values. The states that comprise this execute cycle are

AND1: $DR \leftarrow M$
AND2: $AC \leftarrow AC \wedge DR$

6.2.4.3 JMP Instruction

Any JMP instruction is implemented in basically the same way. The address to which the CPU must jump is copied into the program counter. Then, when the CPU fetches the next instruction, it uses this new address, thus realizing the JMP.

The execute cycle for the JMP instruction for this CPU is quite trivial. Since the address is already stored in *DR*[5..0], we simply copy that value into *PC* and go to the fetch routine. The single state which comprises this execute cycle is

JMP1: $PC \leftarrow DR[5..0]$

In this case, we actually had a second choice. Since this value was copied into *AR* during FETCH3, we could have performed the operation $PC \leftarrow AR$ instead. Either is acceptable.

6.2.4.4 INC Instruction

The INC instruction can also be executed using a single state. The CPU simply adds 1 to the contents of *AC* and goes to the fetch routine. The state for this execute cycle is

INC1: $AC \leftarrow AC + 1$

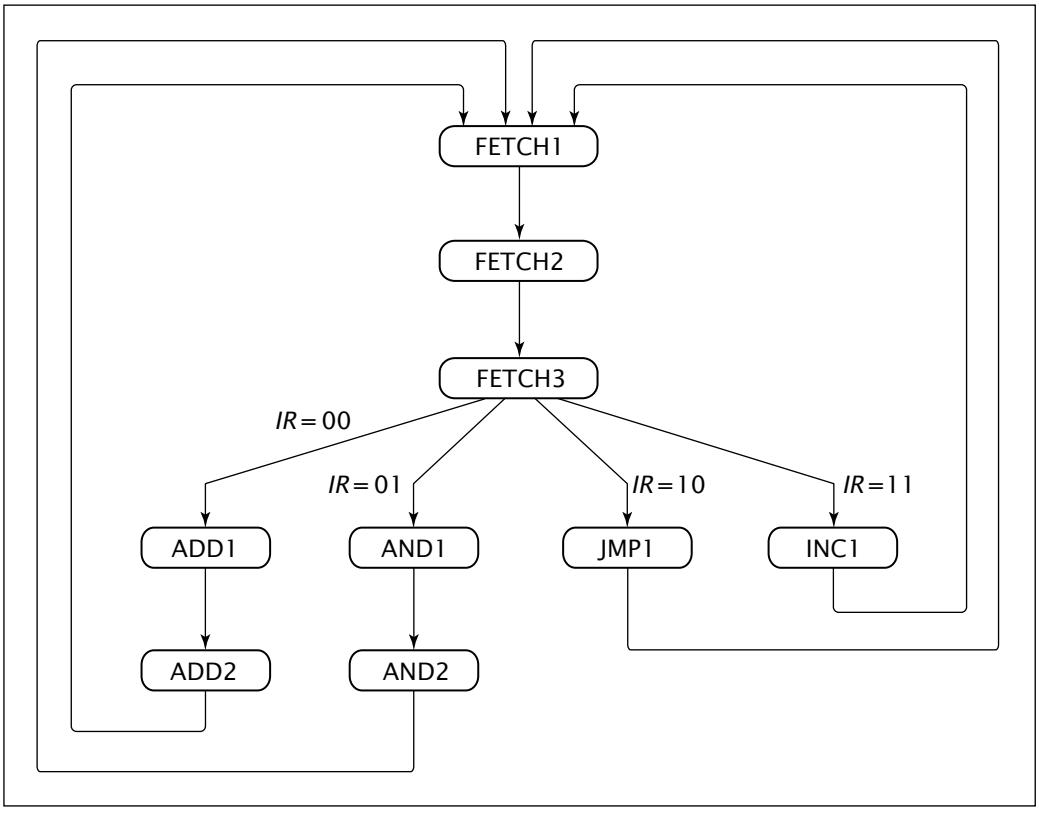
The state diagram for this CPU, including the fetch, decode, and execute cycles, is shown in Figure 6.4 on page 222.

6.2.5 Establishing Required Data Paths

The state diagram and register transfers specify what must be done in order to realize this CPU. Now we must design the CPU so that it actually does these things. First, we look at what data transfers can take place and design the internal data paths of the CPU so this can be done. The operations associated with each state for this CPU are

FETCH1: $AR \leftarrow PC$
FETCH2: $DR \leftarrow M$, $PC \leftarrow PC + 1$
FETCH3: $IR \leftarrow DR[7..6]$, $AR \leftarrow DR[5..0]$
ADD1: $DR \leftarrow M$
ADD2: $AC \leftarrow AC + DR$
AND1: $DR \leftarrow M$
AND2: $AC \leftarrow AC \wedge DR$
JMP1: $PC \leftarrow DR[5..0]$
INC1: $AC \leftarrow AC + 1$

Figure 6.4
Complete state diagram for the Very Simple CPU



(If this looks like RTL code, you're headed in the right direction!) Note that memory supplies its data to the CPU via pins D[7..0]. Also recall that the address pins A[5..0] receive data from the address register, so the CPU must include a data path from the outputs of *AR* to A.

To design the data paths, we can take one of two approaches. The first is to create **direct paths** between each pair of components that transfer data. We can use multiplexers or buffers to select one of several possible data inputs for registers that can receive data from more than one source. For example, in this CPU, *AR* can receive data from *PC* or *DR*[5..0], so the CPU would need a mechanism to select which one is to supply data to *AR* at a given time. This approach could work for this CPU because it is so small. However, as CPU complexity increases, this becomes impractical. A more sensible approach is to create a **bus** within the CPU and route data between components via the bus.

To illustrate the bus concept, consider an interstate highway that is 200 miles long and has about as many exits. Assume that each exit connects to one town. When building roads, the states had two choices: They could build a separate pair of roads (one in each direction) between every pair of towns, resulting in almost 40,000 roads, or one major highway with entrance and exit ramps connecting the towns. The bus is like the interstate highway: It consolidates traffic and reduces the number of roads (data paths) needed.

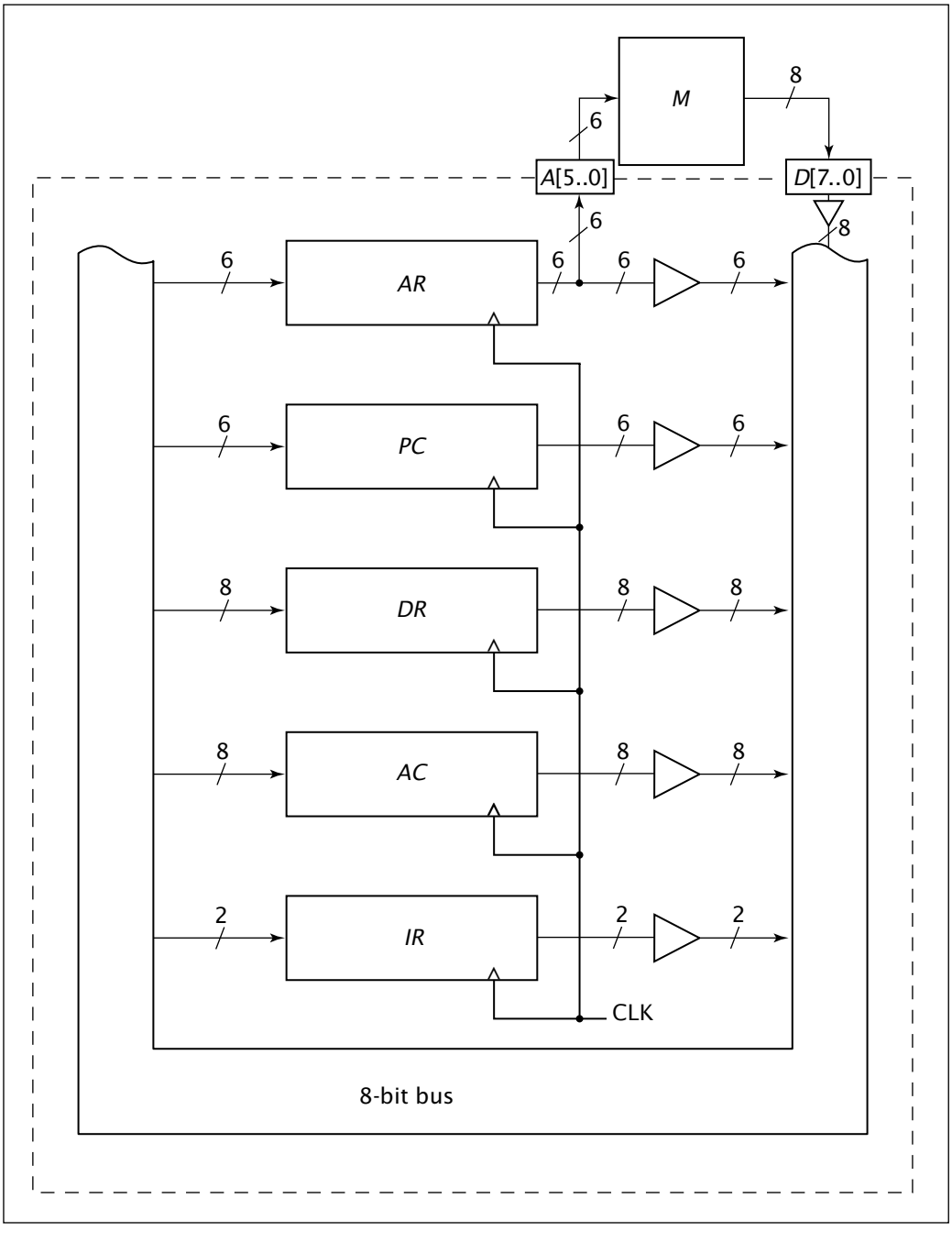
We begin by reviewing the data transfers that can occur to determine the functions of each individual component. Specifically, we look at the operations that load data into each component. It is not necessary to look at operations in which a component supplies the data or one of the operands; that will be taken care of when we look at the component whose value is being changed. First we regroup the operations, without regard for the cycles in which they occur, by the register whose contents they modify. This results in the following:

AR: $AR \leftarrow PC$; $AR \leftarrow DR[5..0]$
PC: $PC \leftarrow PC + 1$; $PC \leftarrow DR[5..0]$
DR: $DR \leftarrow M$
IR: $IR \leftarrow DR[7..6]$
AC: $AC \leftarrow AC + DR$; $AC \leftarrow AC \wedge DR$; $AC \leftarrow AC + 1$

Now we examine the individual operations to determine which functions can be performed by each component. *AR*, *DR*, and *IR* always load data from some other component, made available by the bus, so they only need to be able to perform a parallel load. *PC* and *AC* can load data from external sources, but they both need to be able to increment their values. We could create separate hardware that would increment the current contents of each register and make it available for the register to load back in, but it is easier to design each register as a counter with parallel load capability. In that way, the increment operations can be performed solely within the register; the parallel load is used to implement the other operations.

Next, we connect every component to the system bus, as shown in Figure 6.5 on page 224. Notice that we have included tri-state buffers between the outputs of the registers and the system bus. If we did not do this, all the registers would place their data onto the bus at all times, making it impossible to transfer valid data within the CPU. Also, the outputs of *AR* are connected to pins A[5..0], as required in the CPU specification. At this point, the CPU does not include the control unit, nor the control signals; we will design those later. Right now our goal is to ensure that all data transfers can occur. Later we will design the control unit to make sure that they occur properly.

Figure 6.5
Preliminary register section for the Very Simple CPU



Now we look at the actual transfers that must take place and modify the design accordingly. After reviewing the list of possible operations, we note several things:

1. *AR* only supplies its data to memory, not to other components. It is not necessary to connect its outputs to the internal bus.
2. *IR* does not supply data to any other component via the internal bus, so its output connection can be removed. (The output of *IR* will be routed directly to the control unit, as shown later.)
3. *AC* does not supply its data to any component; its connection to the internal bus can also be removed.
4. The bus is 8 bits wide, but not all data transfers are 8 bits; some are only 6 bits and one is 2 bits. We must specify which registers send data to and receive data from which bits of the bus.
5. *AC* must be able to load the sum of *AC* and *DR*, and the logical AND of *AC* and *DR*. The CPU needs to include an ALU that can generate these results.

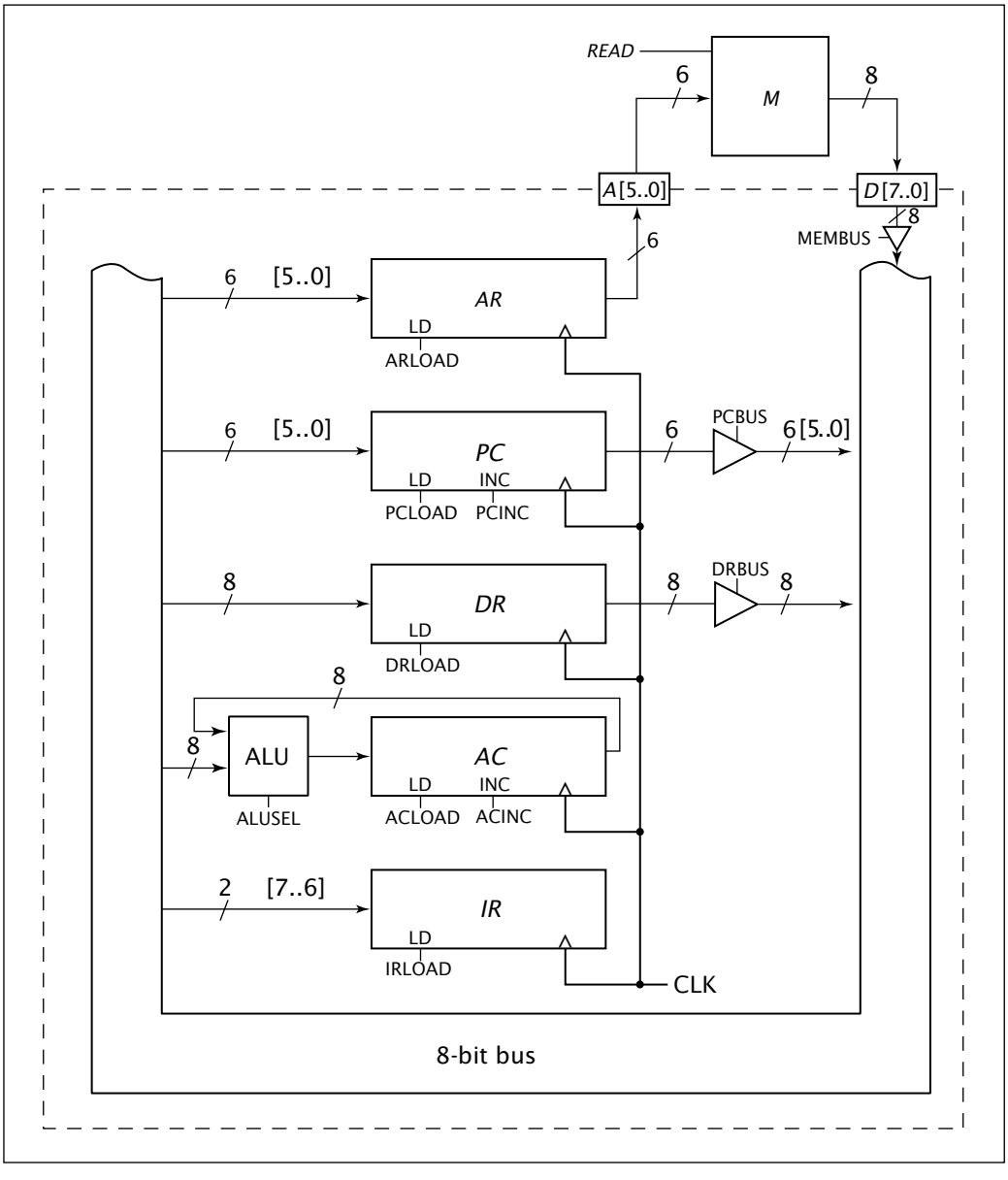
The first three changes are easy to make; we simply remove the unused connections. The fourth item is more of a bookkeeping matter than anything else. In most cases, we simply connect registers to the lowest order bits of the bus. For example, *AR* and *PC* are connected to bits 5..0 of the bus, since they are only 6-bit registers. The lone exception is *IR*. Since it receives data only from *DR*[7..6], it should be connected to the high-order 2 bits of the bus.

Now comes the tricky part. Since *AC* can load in one of two values, either *AC* + *DR* or *AC* ^ *DR*, the CPU must incorporate some arithmetic and logic circuitry to generate these values. (Most CPUs contain an arithmetic/logic unit to do just that.) In terms of the data paths, the ALU must receive *AC* and *DR* as inputs, and send its output to *AC*. There are a couple of ways to route the data to accomplish this. In this CPU we hardwire *AC* as an input to and output from the ALU, and route *DR* as an input to the ALU via the system bus.

At this point the CPU is capable of performing all of the required data transfers. Before proceeding, we must make sure transfers that are to occur during the same state can in fact occur simultaneously. For example, if two transfers that occur in the same state both require that data be placed on the internal bus, they could not be performed simultaneously, since only one piece of data may occupy the bus at a given time. (This is another reason for implementing *PC*←*PC* + 1 by using a counter for *PC*; if that value was routed via the bus, both operations during FETCH2 would have required the bus.) As it is, no state of the state diagram for this CPU would require more than one value to be placed on the bus, so this design is OK in that respect.

The modified version of the internal organization of the CPU is shown in Figure 6.6. The control signals shown will be generated by the control unit.

Figure 6.6
Final register section for the Very Simple CPU

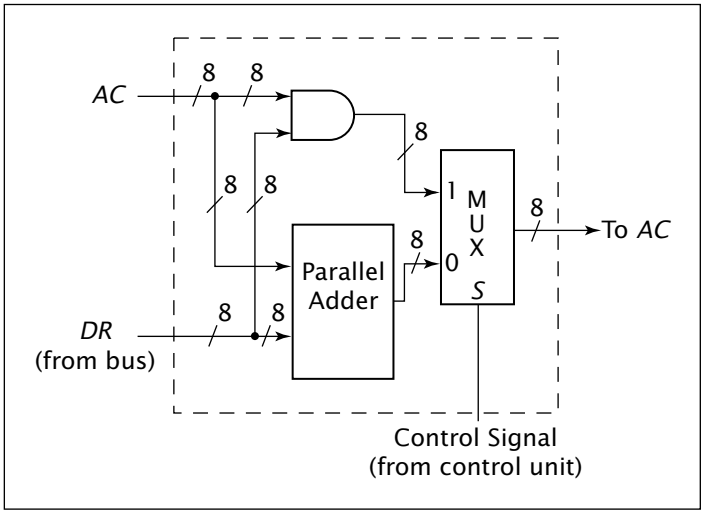


6.2.6 Design of a Very Simple ALU

The ALU for this CPU performs only two functions: adds its two inputs or logically ANDs its two inputs. The simplest way to design this ALU is to create separate hardware to perform each function and then use

a multiplexer to output one of the two results. The addition is implemented using a standard 8-bit parallel adder. The logical AND operation is implemented using eight 2-input AND gates. The outputs of the parallel adder and the AND gates are input to an 8-bit 2 to 1 multiplexer. The control input of the MUX is called *S* (for select). The circuit diagram for the ALU is shown in Figure 6.7.

Figure 6.7
A Very Simple ALU



6.2.7 Designing the Control Unit Using Hardwired Control

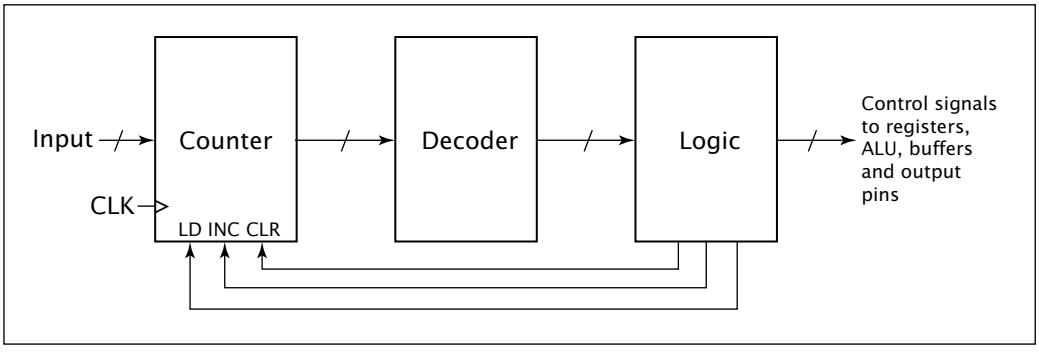
At this point it is possible for the CPU to perform every operation necessary to fetch, decode and execute the entire instruction set. The next task is to design the circuitry to generate the control signals to cause the operations to occur in the proper sequence. This is the **control unit** of the CPU.

There are two primary methodologies for designing control units. Hardwired control uses sequential and combinatorial logic to generate control signals, whereas microsequenced control uses a lookup memory to output the control signals. Each methodology has several design variants. This chapter focuses on hardwired control; microsequenced control is covered in Chapter 7.

This Very Simple CPU requires only a very simple control unit. The simplest control unit has three components: a counter, which contains the current state; a decoder, which takes the current state and generates individual signals for each state; and some combinatorial logic to take the individual state signals and generate the control signals for each component, as well as the signals to control the counter.

These signals cause the control unit to traverse the states in the proper order. A generic version of this type of hardwired control unit is shown in Figure 6.8.

Figure 6.8
Generic hardwired control unit



For this CPU, there are a total of 9 states. Therefore, a 4-bit counter and a 4-to-16-bit decoder are needed. Seven of the outputs of the decoder will not be used.

The first task is to determine how best to assign states to the outputs of the decoder, and thus values in the counter. The following guidelines may help.

1. *Assign FETCH1 to counter value 0 and use the CLR input of the counter to reach this state.* Looking at the state diagram for this CPU, we see that every state except FETCH1 can only be reached from one other state. FETCH1 is reached from four states, the last state of each execute routine. By allocating FETCH1 to counter value 0, these four branches can be realized by asserting the CLR signal of the counter, which minimizes the amount of digital logic needed to design the control unit.
2. *Assign sequential states to sequential counter values and use the INC input of the counter to traverse these states.* If this is done, the control unit can traverse these sequential states by asserting the INC signal of the counter, which also reduces the digital logic needed in the control unit. This CPU would assign FETCH2 to counter value 1 and FETCH3 to counter value 2. It would also assign ADD1 and ADD2 to consecutive counter values, as well as AND1 and AND2.
3. *Assign the first state of each execute routine based on the instruction opcodes and the maximum number of states in the execute routines. Use the opcodes to generate the data input to the counter and the LD input of the counter to reach the proper execute routine.* This point squarely addresses the implementation of in-

struction decoding. Essentially, it implements a mapping of the opcode to the execute routine for that instruction. It occurs exactly once in this and all CPUs, at the last state of the fetch cycle.

To load in the address of the proper execute routine, the control unit must do two things. First, it must place the address of the first state of the proper execute routine on the data inputs of the counter. Second, it must assert the LD signal of the counter. The LD signal is easy; it is directly driven by the last state of the fetch cycle, FETCH3 for this CPU. The difficulty comes in allocating counter values to the states.

Toward that end, consider the list of instructions, their first states, and the value in register *IR* for those instructions, as shown in Table 6.2. The input to the counter is a function of the value of *IR*. The goal is to make this function as simple as possible. Consider one possible mapping, 10*IR*[1..0]. That is, if *IR* = 00, the input to the counter is 1000; for *IR* = 01, the input is 1001, and so on. This would result in the assignment shown in Table 6.3.

Table 6.2

Instructions, first states, and opcodes for the Very Simple CPU

Instruction	**First State**	***IR***
ADD	ADD1	00
AND	AND1	01
JMP	JMP1	10
INC	INC1	11

Table 6.3

Counter values for the proposed mapping function

***IR*[1..0]**	**Counter Value**	**State**
00	1000 (8)	ADD1
01	1001 (9)	AND1
10	1010 (10)	JMP1
11	1011 (11)	INC1

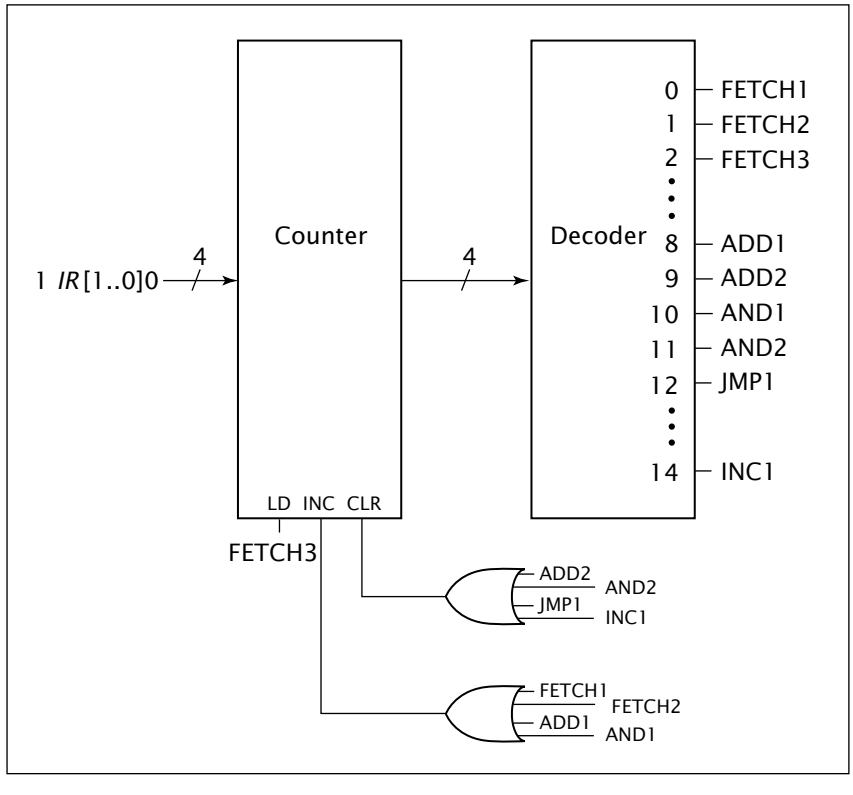
Although this would get to the proper execute routine, it causes a problem. Since state ADD1 has a counter value of 8, and state AND1 has a counter value of 9, what value should we assign to ADD2 and how would it be accessed from ADD1? This could be done by incorporating additional logic, but this is not the best solution for the design.

Looking at the state diagram for this CPU, we see that no execute routine contains more than two states. As long as the first states of

the execute routines have counter values at least two apart, it is possible to store the execute routines in sequential locations. This is accomplished by using the mapping function 1*IR*[1..0]0, which results in counter values of 8, 10, 12, and 14 for ADD1, AND1, JMP1, and INC1, respectively. To assign the execute routines to consecutive values, we assign ADD2 to counter value 9 and AND2 to counter value 11.

Now that we have decided which decoder output is assigned to each state, we can use these signals to generate the **control signals** for the counter of the control unit and for the components of the rest of the CPU. For the counter, we must generate the INC, CLR, and LD signals. INC is asserted when the control unit is traversing sequential states, during FETCH1, FETCH2, ADD1, and AND1. CLR is asserted at the end of each execute cycle to return to the fetch cycle; this happens during ADD2, AND2, JMP1, and INC1. Finally, as noted earlier, LD is asserted at the end of the fetch cycle during state FETCH3. Note that each state of the CPU's state diagram drives exactly one of these three control signals. The circuit diagram for the control unit at this point is shown in Figure 6.9.

Figure 6.9
Hardwired control unit for the Very Simple CPU



These state signals are also combined to create the control signals for *AR*, *PC*, *DR*, *IR*, *M*, the ALU, and the buffers. First consider register *AR*. It is loaded during states FETCH1 (*AR*←*PC*) and FETCH3 (*AR*←*DR*[5..0]). By logically ORing these two state signals together, the CPU generates the LD signal for *AR*. It doesn't matter which value is to be loaded into *AR*, at least as far as the LD signal is concerned. When the designers create the control signals for the buffers, they will ensure that the proper data is placed on the bus and made available to *AR*. Following this procedure, we create the following control signals for *PC*, *DR*, *AC*, and *IR*:

$$\begin{aligned}
&\text{PCLOAD} = \text{JMP1}\\
&\text{PCINC} = \text{FETCH2}\\
&\text{DRLOAD} = \text{FETCH1} \vee \text{ADD1} \vee \text{AND1}\\
&\text{ACLOAD} = \text{ADD2} \vee \text{AND2}\\
&\text{ACINC} = \text{INC1}\\
&\text{IRLOAD} = \text{FETCH3}
\end{aligned}$$

The ALU has one control input, ALUSEL. When ALUSEL = 0, the output of the ALU is the arithmetic sum of its two inputs; if ALUSEL = 1, the output is the logical AND of its inputs. Setting ALUSEL = AND2 routes the correct data from the ALU to *AC* when the CPU is executing an ADD or AND instruction. At other times, during the fetch cycle and the other execute cycles, the ALU is still outputting a value to *AC*. However, since *AC* does not load this value, the value output by the ALU does not cause any problems.

Many of the operations use data from the internal system bus. The CPU must enable the buffers so the correct data is placed on the bus at the proper time. Again, looking at the operations that occur during each state, we can generate the enable signals for the buffers. For example, *DR* must be placed onto the bus during FETCH3 (*IR*←*DR*[7..6], *AR*←*DR*[5..0]), ADD2 (*AC*←*AC* + *DR*), AND2 (*AC*←*AC* ^ *DR*) and JMP1 (*PC*←*DR*[5..0]). (Recall that the ALU receives *DR* input via the internal bus.) Logically ORing these state values produces the DRBUS signal. This procedure is used to generate the enable signals for the other buffers as well:

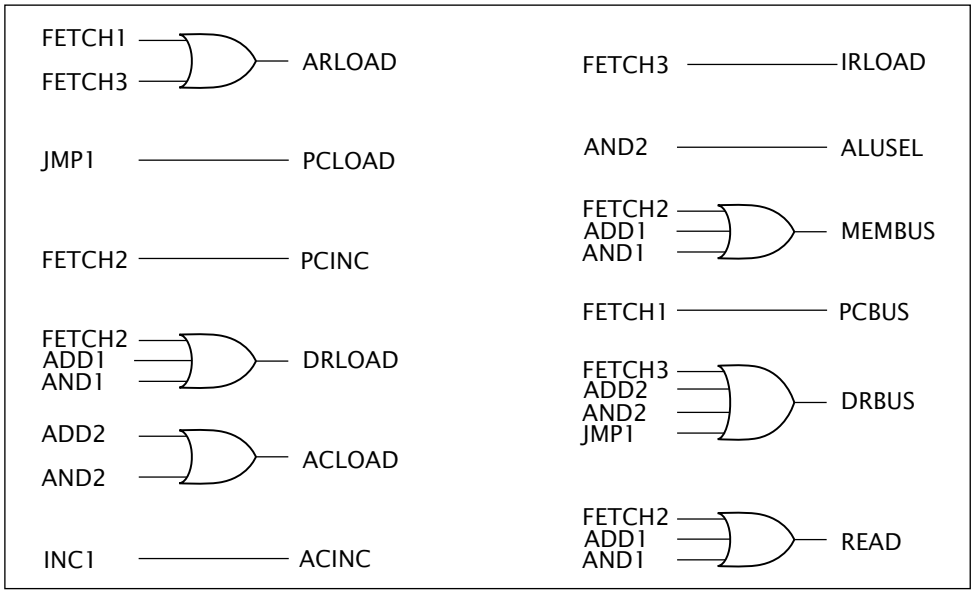
$$\begin{aligned}
&\text{MEMBUS} = \text{FETCH2} \vee \text{ADD1} \vee \text{AND1}\\
&\text{PCBUS} = \text{FETCH1}
\end{aligned}$$

Finally, the control unit must generate a READ signal, which is output from the CPU. This signal causes memory to output its data value. This occurs when memory is read during states FETCH2, ADD1, and AND1, so READ can be set as follows:

$$\text{READ} = \text{FETCH2} \vee \text{ADD1} \vee \text{AND1}$$

The circuit diagram for the portion of the control unit that generates these signals is shown in Figure 6.10. This completes the design of the Very Simple CPU.

Figure 6.10
Control signal generation for the Very Simple CPU



6.2.8 Design Verification

Now that we have designed the CPU, we must verify that it works properly. To do so, we trace through the fetch, decode, and execute cycles of each instruction. Consider this segment of code, containing each instruction once:

```
0:  ADD4
1:  AND5
2:  INC
3:  JMP 0
4:  27H
5:  39H
```

The CPU fetches, decodes, and executes each instruction following the appropriate state sequences from the state diagram:

```
ADD4:    FETCH1→FETCH2→FETCH3→ADD1→ADD2
AND5:    FETCH1→FETCH2→FETCH3→AND1→AND2
INC:     FETCH1→FETCH2→FETCH3→INC1
JMP 0:   FETCH1→FETCH2→FETCH3→JMP1
```

Table 6.4 shows the trace of the execution of one iteration of this program. We can see that the program processes every instruction correctly. Initially all registers contain the value 0.

Table 6.4
Execution trace

Instruction	State	Active Signals	Operations Performed	Next State
ADD 4	FETCH1	PCBUS, ARLOAD	*AR*←0	FETCH2
	FETCH2	READ, MEMBUS, DRLOAD, PCINC	*DR*←04H, *PC*←1	FETCH3
	FETCH3	DRBUS, ARLOAD, IRLOAD	*IR*←00, *AR*←04H	ADD1
	ADD1	READ, MEMBUS, DRLOAD	*DR*←27H	ADD2
	ADD2	DRBUS, ACLOAD	*AC*←0 + 27H = 27H	FETCH1
AND 5	FETCH1	PCBUS, ARLOAD	*AR*←1	FETCH2
	FETCH2	READ, MEMBUS, DRLOAD, PCINC	*DR*←45H, *PC*←2	FETCH3
	FETCH3	DRBUS, ARLOAD, IRLOAD	*IR*←01, *AR*←05H	AND1
	AND1	READ, MEMBUS, DRLOAD	*DR*←39H	AND2
	AND2	DRBUS, ALUSEL, ACLOAD	*AC*←27H ^ 39H = 31H	FETCH1
INC	FETCH1	PCBUS, ARLOAD	*AR*←2	FETCH2
	FETCH2	READ, MEMBUS, DRLOAD, PCINC	*DR*←C0H, *PC*←3	FETCH3
	FETCH3	DRBUS, ARLOAD, IRLOAD	*IR*←11, *AR*←00H	INC1
	INC1	ACINC	*AC*←21H + 1 = 22H	FETCH1
JMP 0	FETCH1	PCBUS, ARLOAD	*AR*←3	FETCH2
	FETCH2	READ, MEMBUS, DRLOAD, PCINC	*DR*←80H, *PC*←4	FETCH3
	FETCH3	DRBUS, ARLOAD, IRLOAD	*IR*←10, *AR*←00H	JMP1
	JMP1	DRBUS, PCLOAD	*PC*←0	FETCH1

6.3 Design and Implementation of a Relatively Simple CPU

The CPU designed in the previous section is named appropriately: It is indeed very simple. It illustrated design methods that are too simple to handle the complexity of a larger CPU. This section presents the design of a more complex, but still relatively simple CPU. This CPU has a larger instruction set with more complex instructions. Its design follows the same general procedure used to design the Very Simple CPU.

6.3.1 Specifications for a Relatively Simple CPU

Chapter 3 introduced the instruction set architecture for the Relatively Simple CPU. This CPU can access 64K bytes of memory, each 8 bits wide, via address pins A[15..0] and bidirectional data pins D[7..0].

Three registers in the ISA of this processor can be directly controlled by the programmer. The 8-bit accumulator, *AC*, receives the result of any arithmetic or logical operation and provides one of the operands for arithmetic and logical instructions, which use two operands. Whenever data is loaded from memory, it is loaded into the accumulator; data stored to memory also comes from *AC*. Register *R* is an 8-bit general purpose register. It supplies the second operand of all two-operand arithmetic and logical instructions. It can also be used to temporarily store data that the accumulator will soon need to access. Finally, there is a 1-bit zero flag, *Z*, which is set whenever an arithmetic or logical instruction is executed.

The final component of the instruction set architecture for this Relatively Simple CPU is its instruction set, shown in Table 6.5.

Table 6.5
Instruction set for a Relatively Simple CPU

Instruction	Instruction Code	Operation
NOP	0000 0000	No operation
LDAC	0000 0001 Γ	$AC \leftarrow M[\Gamma]$
STAC	0000 0010 Γ	$M[\Gamma] \leftarrow AC$
MVAC	0000 0011	$R \leftarrow AC$
MOVR	0000 0100	$AC \leftarrow R$
JUMP	0000 0101 Γ	GOTO Γ
JMPZ	0000 0110 Γ	IF (Z=1) THEN GOTO Γ
JPNZ	0000 0111 Γ	IF (Z=0) THEN GOTO Γ
ADD	0000 1000	$AC \leftarrow AC + R$, IF ($AC + R = 0$) THEN $Z \leftarrow 1$ ELSE $Z \leftarrow 0$
SUB	0000 1001	$AC \leftarrow AC - R$, IF ($AC - R = 0$) THEN $Z \leftarrow 1$ ELSE $Z \leftarrow 0$
INAC	0000 1010	$AC \leftarrow AC + 1$, IF ($AC + 1 = 0$) THEN $Z \leftarrow 1$ ELSE $Z \leftarrow 0$
CLAC	0000 1011	$AC \leftarrow 0$, $Z \leftarrow 1$
AND	0000 1100	$AC \leftarrow AC \wedge R$, IF ($AC \wedge R = 0$) THEN $Z \leftarrow 1$ ELSE $Z \leftarrow 0$
OR	0000 1101	$AC \leftarrow AC \vee R$, IF ($AC \vee R = 0$) THEN $Z \leftarrow 1$ ELSE $Z \leftarrow 0$
XOR	0000 1110	$AC \leftarrow AC \oplus R$, IF ($AC \oplus R = 0$) THEN $Z \leftarrow 1$ ELSE $Z \leftarrow 0$
NOT	0000 1111	$AC \leftarrow AC'$, IF ($AC' = 0$) THEN $Z \leftarrow 1$ ELSE $Z \leftarrow 0$

As in the Very Simple CPU, this Relatively Simple CPU contains several registers in addition to those specified in its instruction set architecture. Differences between these registers and those of the Very Simple CPU are italicized:

- A *16-bit* address register, *AR,* which supplies an address to memory via A[*15..0*]
- A *16-bit* program counter, *PC,* which contains the address of the next instruction to be executed *or the address of the next required operand of the instruction*
- An 8-bit data register, *DR,* which receives instructions and data from memory *and transfers data to memory* via D[7..0]
- An *8-bit* instruction register, *IR,* which stores the opcode fetched from memory
- *An* 8-bit *temporary register,* TR, *which temporarily stores data during instruction execution*

Besides the differences in register size, there are several differences between the registers for this CPU and the Very Simple CPU. These changes are all necessary to accommodate the more complex instruction set.

First of all, notice that the program counter can hold not only the address of the next instruction, but also the address of the next operand. In the Very Simple CPU, the only operand is an address that is fetched along with the opcode. The Relatively Simple CPU uses 8-bit opcodes and 16-bit addresses. If the opcode and address were packed into one word, it would have to be 24 bits wide. For instructions that do not access memory, the 16-bit address portion of the instruction code would be wasted. To minimize unused bits, the CPU keeps each word/byte 8 bits wide, but uses multiple bytes to store the instruction and its address. Part of the time the *PC* will be pointing to the memory byte containing the opcode, but at other times it will be pointing to the memory bytes containing the address. This may seem a bit confusing, but it will become clearer during the design of this CPU.

The Very Simple CPU could not output data. The Relatively Simple CPU provides this capability, and it does so by making data available on the bidirectional pins D[7..0]. For this design, this data is provided solely from *DR*.

Most CPUs have more than one internal register for manipulating data. For this reason, the Relatively Simple CPU includes a general purpose register, *R*. Internal registers improve the performance of the CPU by reducing the number of times memory must be accessed. To illustrate this, consider the ADD instruction of the Very Simple CPU. After fetching and decoding the instruction, the CPU had to fetch the operand from memory before adding it to the accumulator. The Relatively Simple CPU adds the contents of register *R* to *AC*, eliminating the memory access and reducing the time needed to perform the addition.

Most CPUs have several general purpose registers; this CPU has only one to illustrate the use of general purpose registers while still keeping the design relatively simple.

Most CPUs contain internal data registers that cannot be accessed by the programmer. This CPU contains temporary register *TR*, which it uses to store data during the execution of instructions. As we will see, the CPU can use this register to save data while it fetches the address for memory reference instructions. Unlike the contents of *AC* or *R*, which are directly modified by the user, no instruction causes a permanent change in the contents of *TR*.

Finally, most CPUs contain **flag registers**, or **flags**, which show the result of a previous operation. Typical flags indicate whether or not an operation generated a carry, the sign of the result, or the parity of the result. The Relatively Simple CPU contains a zero flag, *Z*, which is set to 1 if the last arithmetic or logical operation produced a result equal to 0. Not every instruction changes the contents of *Z* in this and other CPUs. For example, an ADD instruction sets *Z*, but a MOVR (move data from *R* into *AC*) instruction does not. Most CPUs contain conditional instructions that perform different operations, depending on the value of a given flag. The JMPZ and JPNZ instructions for this CPU fall into this category.

6.3.2 Fetching and Decoding Instructions

This CPU fetches instructions from memory in exactly the same way as the Very Simple CPU does, except at the end of the fetch cycle. Here *IR* is 8 bits and receives the entire contents of *DR*. Also, *AR*←*PC*, instead of *DR*[5..0] because that is the next address it would need to access. The fetch cycle thus becomes

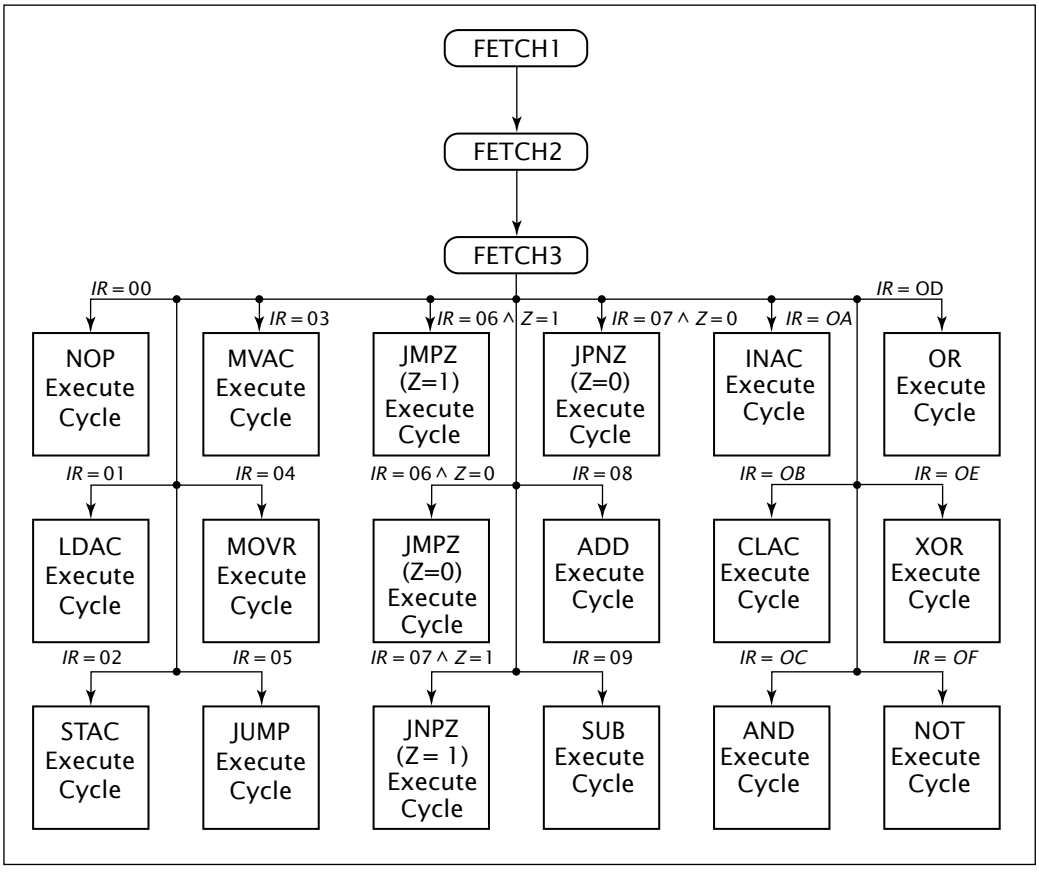
FETCH1: *AR*←*PC*
FETCH2: *DR*←*M*, *PC*←*PC* + 1
FETCH3: *IR*←*DR*, *AR*←*PC*

The state diagram for this fetch cycle is exactly the same as that of the Very Simple CPU shown in Figure 6.2.

We also follow the same process for decoding instructions that we used for the Very Simple CPU. Here, *IR* is 8 bits wide and there will be more possible branches. The state diagram for the fetch and decode cycles is shown in Figure 6.11.

There is one particularly unusual feature of the state diagram in Figure 6.11. Two of the instructions, JMPZ and JPNZ, have two different execute routines. These conditional instructions will be executed in one of two ways, depending on the value of *Z*. Either they will jump to address Γ or they will not. Each execute routine implements one of these two possibilities; the value of *Z* determines which is selected.

Figure 6.11
Fetch and decode cycles for the Relatively Simple CPU



6.3.3 Executing Instructions

The final task in creating the state diagram for this CPU is to prepare the state diagrams for the execute routines. As before, we develop them individually and combine them into a final state diagram.

6.3.3.1 NOP Instruction

The NOP is the easiest instruction to implement. The CPU does nothing and then goes to the fetch routine to fetch the next instruction. This could be accomplished either by having the fetch routine branch back to its own beginning or by creating a single state that does nothing as the execute routine. In this CPU we use the latter approach. The state diagram for this execute routine contains the single state

NOP1: (No operation)

6.3.3.2 LDAC Instruction

LDAC is the first of the multiword instructions in this CPU. It contains three words: the opcode, the low-order half of the address, and the high-order half of the address. The execute routine must get the address from memory, then get data from that memory location and load it into the accumulator.

Remember that, after the instruction has been fetched from memory, the program counter contains the next address in memory. If the instruction consisted of a single byte, the *PC* would contain the address of the next instruction. Here, however, it contains the address of the first operand, the low-order half of the address Γ. This CPU uses this value of *PC* to access the address.

First the CPU must get the address from memory. Since the address of the low-order half of address Γ was loaded into *AR* during FETCH3, this value can now be read in from memory. The CPU must also do two other things at this time: Because the CPU has read in the data whose address is stored in *PC*, it must increment *PC*, and because it will need to get the high-order half of the address from the next memory location, it must also increment *AR*. The CPU could simply increment *PC* now and then load it into *AR* during the next state, but incrementing *AR* now will reduce the number of states needed to execute the LDAC instruction. Thus the first state of this execute routine is

$$\text{LDAC1:} \quad DR \leftarrow M,\ PC \leftarrow PC + 1,\ AR \leftarrow AR + 1$$

Having fetched the low-order half of the address, the CPU now must fetch the high-order half. It must also save the low-order half somewhere other than *DR*; otherwise it will be overwritten by the high-order half of address Γ. Here we make use of the temporary register *TR*. Again, the CPU must increment *PC* or it will not have the correct address for the next fetch routine. The second state is

$$\text{LDAC2:} \quad TR \leftarrow DR,\ DR \leftarrow M,\ PC \leftarrow PC + 1$$

Now that the CPU contains the address, it can read the data from memory. To do this, the CPU first copies the address into *AR*, then reads data from memory into *DR*. Finally, it copies that data into the accumulator and branches back to the fetch routine. The states to perform these operations are

$$\begin{aligned} &\text{LDAC3:} \quad AR \leftarrow DR, TR \\ &\text{LDAC4:} \quad DR \leftarrow M \\ &\text{LDAC5:} \quad AC \leftarrow DR \end{aligned}$$

6.3.3.3 STAC Instruction

Although the STAC instruction performs the opposite operation of LDAC, it duplicates several of its states. Specifically, it fetches the

memory address in exactly the same way as LDAC; states STAC1, STAC2, and STAC3 are identical to LDAC1, LDAC2, and LDAC3, respectively.

Once *AR* contains the address, this routine must copy the data from *AC* to *DR*, then write it to memory. The states that comprise this execute routine are

STAC1: $DR \leftarrow M$, $PC \leftarrow PC + 1$, $AR \leftarrow AR + 1$
STAC2: $TR \leftarrow DR$, $DR \leftarrow M$, $PC \leftarrow PC + 1$
STAC3: $AR \leftarrow DR, TR$
STAC4: $DR \leftarrow AC$
STAC5: $M \leftarrow DR$

At first glance, it may appear that STAC3 and STAC4 can be combined into a single state. However, when constructing the data paths later in the design process, we decided to route both transfers via an internal bus. Since both values cannot occupy the bus simultaneously, we chose to split the state in two rather than create a separate data path. This process is not uncommon, and the designer should not be concerned about needing to modify the state diagram because of data path conflicts. Consider it one of the tradeoffs inherent to engineering design.

6.3.3.4 MVAC and MOVR Instructions

The MVAC and MOVR instructions are both fairly straightforward. The CPU simply performs the necessary data transfer in one state and goes back to the fetch routine. The states that comprise these routines are

MVAC1: $R \leftarrow AC$

and

MOVR1: $AC \leftarrow R$

6.3.3.5 JUMP Instruction

To execute the JUMP instruction, the CPU fetches the address just as it did for the LDAC and STAC instructions, except it does not increment *PC*. Instead of loading the address into *AR*, it copies the address into *PC*, so any incremented value of *PC* would be overwritten anyway. This instruction can be implemented using three states.

JUMP1: $DR \leftarrow M$, $AR \leftarrow AR + 1$
JUMP2: $TR \leftarrow DR$, $DR \leftarrow M$
JUMP3: $PC \leftarrow DR, TR$

6.3.3.6 JMPZ and JPNZ Instructions

The JMPZ and JPNZ instructions each have two possible outcomes, depending on the value of the *Z* flag. If the jump is to be taken, the CPU follows execution states exactly the same as those used by the JUMP

instruction. However, if the jump is not taken, the CPU cannot simply return to the fetch routine. After the fetch routine, the *PC* contains the address of the low-order half of the jump address. If the jump is not taken, the CPU must increment the *PC* twice so that it points to the next instruction in memory, not to either byte of Γ. The states to perform the JMPZ instruction are as follows. Note that the JMPZY states are executed if $Z = 1$ and the JMPZN states are executed if $Z = 0$.

JMPZY1: $DR \leftarrow M$, $AR \leftarrow AR + 1$
JMPZY2: $TR \leftarrow DR$, $DR \leftarrow M$
JMPZY3: $PC \leftarrow DR$, TR

JMPZN1: $PC \leftarrow PC + 1$
JMPZN2: $PC \leftarrow PC + 1$

The states for JPNZ are identical but are accessed under opposite conditions—that is, JPNZY states are executed when $Z = 0$ and JPNZN states are traversed when $Z = 1$.

JPNZY1: $DR \leftarrow M$, $AR \leftarrow AR + 1$
JPNZY2: $TR \leftarrow DR$, $DR \leftarrow M$
JPNZY3: $PC \leftarrow DR$, TR

JPNZN1: $PC \leftarrow PC + 1$
JPNZN2: $PC \leftarrow PC + 1$

6.3.3.7 The Remaining Instructions

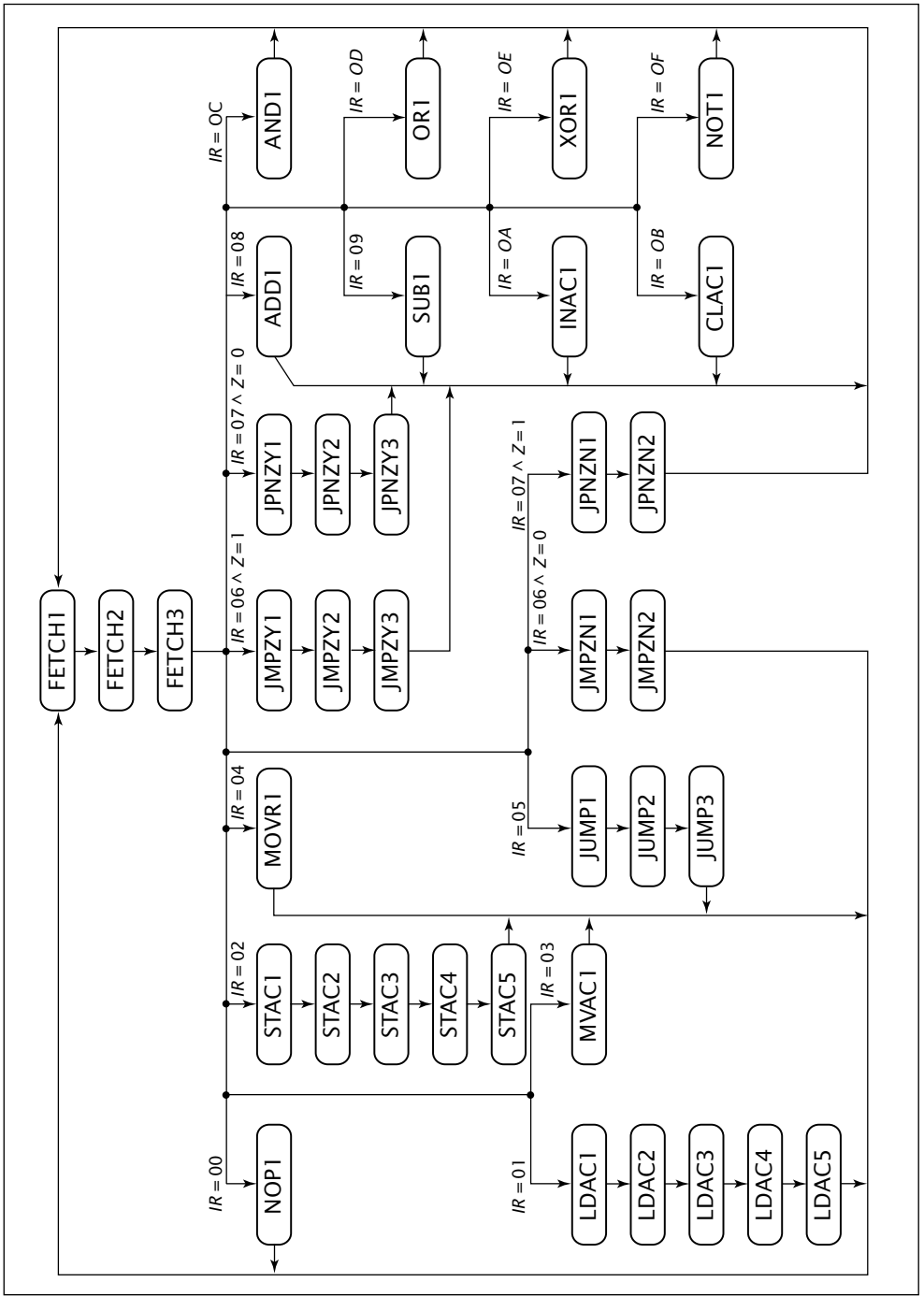
The remaining instructions are each executed in a single state. For each state, two things happen: The correct value is generated and stored in *AC*, and the zero flag is set. If the result of the operation is 0, *Z* is set to 1; otherwise it is set to 0. Since this happens during a single state, the CPU cannot first store the result in *AC* and then set *Z*: It must perform both operations simultaneously. For now we simply specify the states and defer the implementation until later in the design process. The states for these execute routines are as follows.

ADD1: $AC \leftarrow AC + R$, IF $(AC + R = 0)$ THEN $Z \leftarrow 1$ ELSE $Z \leftarrow 0$
SUB1: $AC \leftarrow AC - R$, IF $(AC - R = 0)$ THEN $Z \leftarrow 1$ ELSE $Z \leftarrow 0$
INAC1: $AC \leftarrow AC + 1$, IF $(AC + 1 = 0)$ THEN $Z \leftarrow 1$ ELSE $Z \leftarrow 0$
CLAC1: $AC \leftarrow 0$, $Z \leftarrow 1$
AND1: $AC \leftarrow AC \wedge R$, IF $(AC \wedge R = 0)$ THEN $Z \leftarrow 1$ ELSE $Z \leftarrow 0$
OR1: $AC \leftarrow AC \vee R$, IF $(AC \vee R = 0)$ THEN $Z \leftarrow 1$ ELSE $Z \leftarrow 0$
XOR1: $AC \leftarrow AC \oplus R$, IF $(AC \oplus R = 0)$ THEN $Z \leftarrow 1$ ELSE $Z \leftarrow 0$
NOT1: $AC \leftarrow AC'$, IF $(AC' = 0)$ THEN $Z \leftarrow 1$ ELSE $Z \leftarrow 0$

The state diagram for this entire CPU is shown in Figure 6.12.

Figure 6.12

Complete state diagram for the Relatively Simple CPU



6.3.4 Establishing Data Paths

As with this Very Simple CPU, the Relatively Simple CPU uses an internal data bus to move data between components. First we regroup the data transfers by destination.

AR: $AR \leftarrow PC$; $AR \leftarrow AR + 1$; $AR \leftarrow DR, TR$
PC: $PC \leftarrow PC + 1$; $PC \leftarrow DR, TR$
DR: $DR \leftarrow M$, $DR \leftarrow AC$
IR: $IR \leftarrow DR$
R: $R \leftarrow AC$
TR: $TR \leftarrow DR$
AC: $AC \leftarrow DR$; $AC \leftarrow R$; $AC \leftarrow AC + R$; $AC \leftarrow AC - R$; $AC \leftarrow AC + 1$; $AC \leftarrow 0$; $AC \leftarrow AC \wedge R$; $AC \leftarrow AC \vee R$; $AC \leftarrow AC \oplus R$; $AC \leftarrow AC'$
Z: $Z \leftarrow 1$; $Z \leftarrow 0$ (both conditional)

From these operations, we select the functions of each component:

- *AR* and *PC* must be able to perform a parallel load and increment. Both registers receive their data from the internal bus.
- *DR*, *IR*, *R*, and *TR* must be able to load data in parallel. For now each register will receive its data from the internal bus. As we will see later in the design process, this will not work and more than one connection will have to be changed.
- *AC* will require a lot of work, as will *Z*. This CPU will utilize an ALU to perform all of these functions. The ALU will receive *AC* as one input and the value on the internal bus as the other input. *AC* will always receive its input from the ALU. The CPU will also use the output of the ALU to determine whether or not the result is 0 for the purpose of setting *Z*.

Although the CPU could use a register with parallel load, increment, and clear signals for *AC*, we will only use a register with parallel load and have the ALU create values *AC* + 1 and 0 when necessary. This is done to facilitate the proper setting of *Z*. The *Z* flag is implemented as a 1-bit register with "parallel" load.

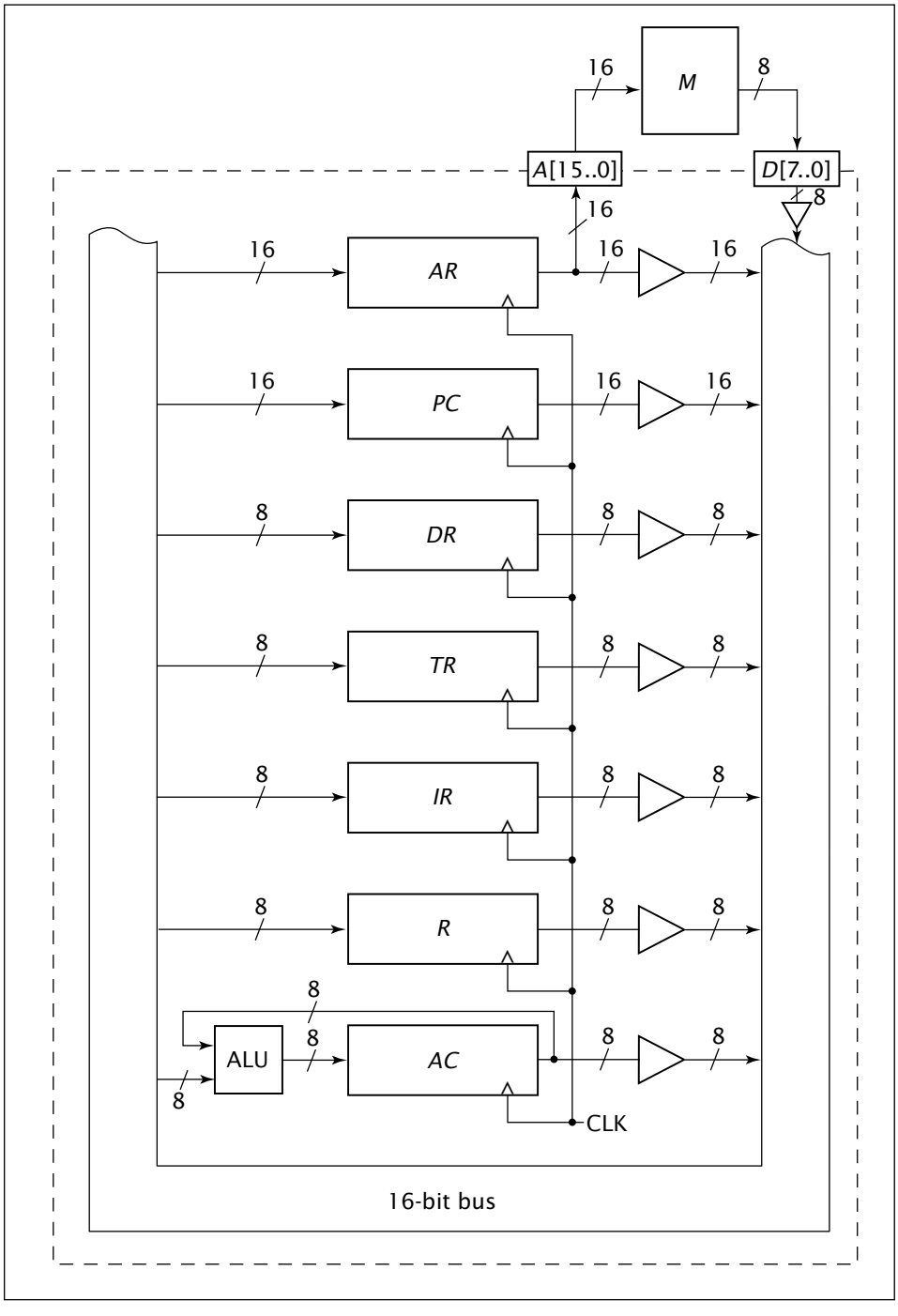
Now we connect every component to the system bus, including tri-state buffers where necessary. We also connect output pins A[15..0] and bidirectional pins D[7..0]. The preliminary connections are shown in Figure 6.13.

Next we modify the design based on the following considerations.

1. As in the Very Simple CPU, *AR* and *IR* of the Relatively Simple CPU do not supply data to other components. We can remove their outputs to the internal bus.
2. Pins D[7..0] are bidirectional, but the current configuration does not allow data to be output from these pins.

Figure 6.13

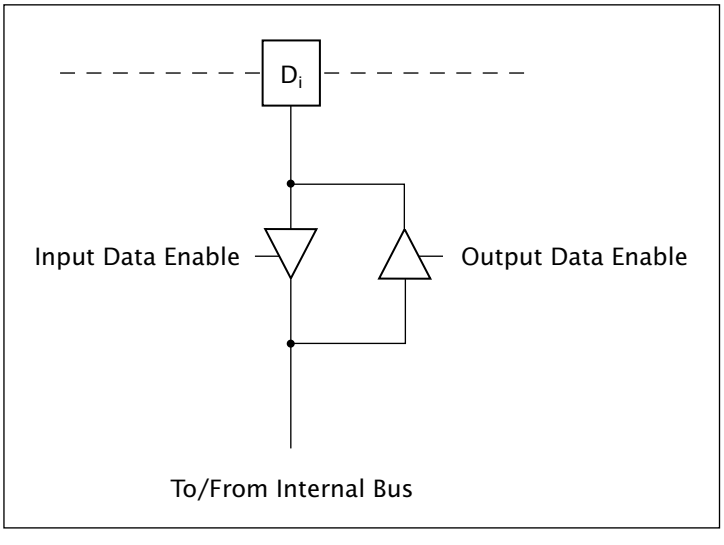
Preliminary register section for the Relatively Simple CPU



3. The 16-bit bus is not fully used by all registers. We must specify which bits of the data bus are connected to which bits of the registers.
4. Register *Z* is not connected to anything.

To address the first point, we simply remove the unused connections. The second point is also straightforward: A standard way to implement bidirectional pins is to use a pair of buffers, one in each direction. One buffer is used to input data from the pins and the other outputs data to the pins. The two buffers must never be enabled simultaneously. This configuration is shown in Figure 6.14.

Figure 6.14
Generic bidirectional data pin



Unlike the Very Simple CPU, it is not a trivial matter to assign connections between the registers and the bits of the data bus in the Relatively Simple CPU. *AR* and *PC* are 16-bit registers connected to a 16-bit bus, so they present no problem. The remaining 8-bit registers can be connected to bits 7..0 of the bus. Although this configuration allows almost every individual transfer to take place, it causes problems for several states:

- During FETCH3, the CPU must transfer *IR*←*DR* and *AR*←*PC* simultaneously. As configured, both transfers would need to use bits 7..0 of the internal bus at the same time, which is not allowable. Since *IR* receives data only from *DR*, it is possible to establish a direct path

from the output of *DR* to the input of *IR*, allowing *IR*←*DR* to occur without using the internal bus. This allows the CPU to perform both operations simultaneously. We can also disconnect the input of *IR* from the internal bus, since it no longer receives data from the bus.

- During LDAC2 and several other states, *TR*←*DR* and *DR*←*M* need to use the bus simultaneously. Fortunately, *TR* also receives data only from *DR*, so the CPU can include a direct path from the output of *DR* to the input of *TR*, just as we did for *IR*. The input of *TR* is also disconnected from the internal bus.
- During LDAC3 and several other states, *DR* and *TR* must be placed on the bus simultaneously, *DR* on bits 15..8 and *TR* on bits 7..0. However, *DR* is connected to bits 7..0 of the bus. One way to handle this is simply to connect the output of *DR* to bits 15..8 instead of bits 7..0, but that would cause a problem during LDAC5 and other states, which need *DR* on bits 7..0. Another solution, implemented here, is to route the output of *DR* to both bits 15..8 and bits 7..0. Separate buffers with different enable signals must be used because *DR* should not be active on both halves of the bus simultaneously.

Finally, we must connect register *Z*. Reviewing the states and their functions, we see that *Z* is only set when an ALU operation occurs. It is set to 1 if the value to be stored in *AC* (which is the output of the ALU) is 0; otherwise it is set to 0. To implement this, we NOR together the bits output from the ALU. The NOR will produce a value of 1 only if all bits are 0; thus the output of the NOR gate can serve as the input of *Z*. This is why we implemented the increment and clear operations via the ALU, rather than incorporating them directly into the *AC* register.

Figure 6.15 on page 246 shows the internal organization of the CPU after incorporating these changes.

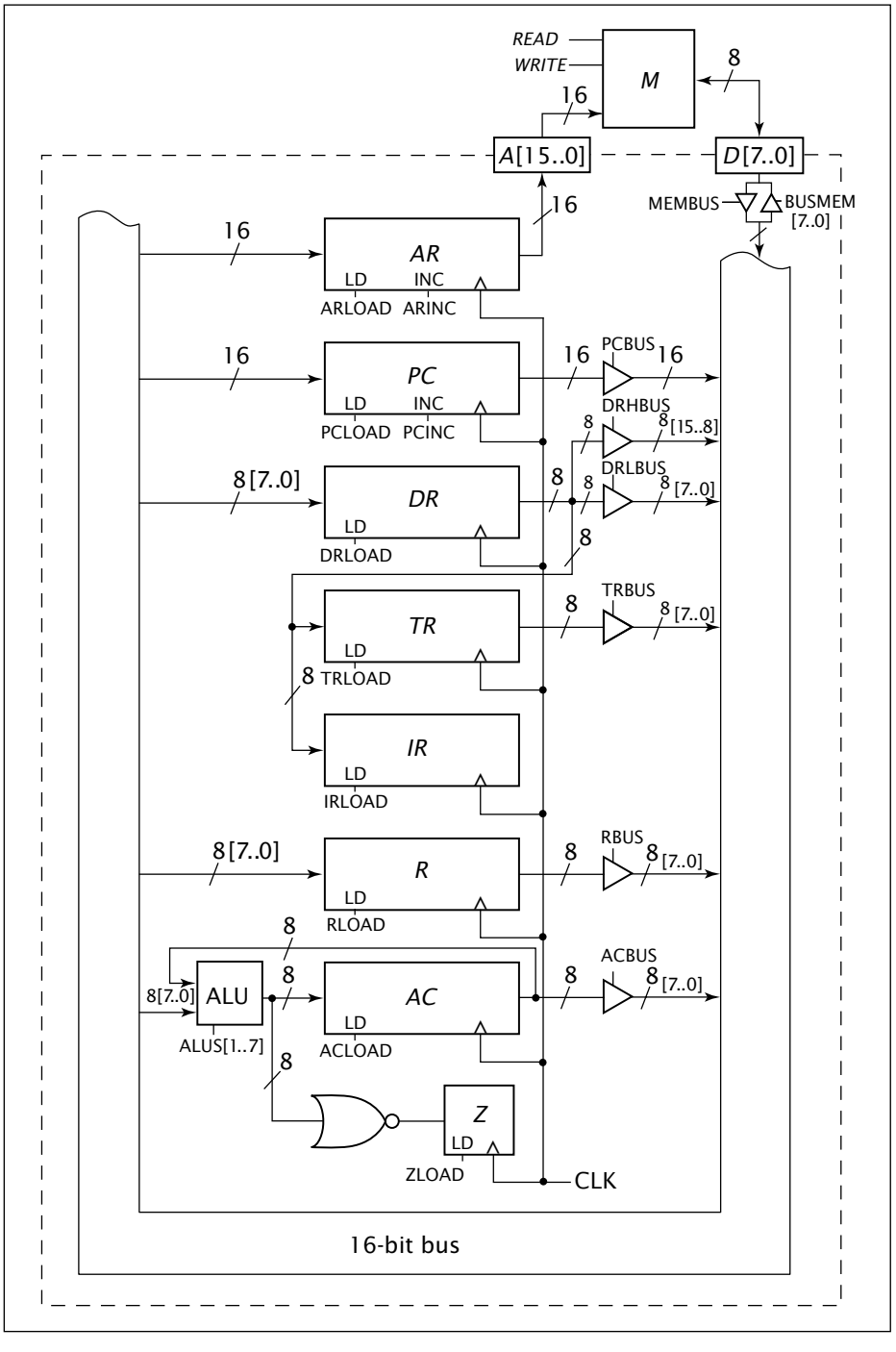
6.3.5 Design of a Relatively Simple ALU

All data that is to be loaded into *AC* must pass through the ALU. To design the ALU, we first list all transfers that modify the contents of *AC*.

LDAC5:	$AC \leftarrow DR$
MOVR1:	$AC \leftarrow R$
ADD1:	$AC \leftarrow AC + R$
SUB1:	$AC \leftarrow AC - R$
INAC1:	$AC \leftarrow AC + 1$
CLAC1:	$AC \leftarrow 0$
AND1:	$AC \leftarrow AC \wedge R$
OR1:	$AC \leftarrow AC \vee R$
XOR1:	$AC \leftarrow AC \oplus R$
NOT1:	$AC \leftarrow AC'$

Figure 6.15

Final register section for the Relatively Simple CPU



An arithmetic/logic unit (ALU) can be designed just as its name implies: We can design one section to perform the arithmetic instructions and another section to perform the logical instructions. A multiplexer selects data from the correct section for output to *AC*.

First we design the arithmetic section. To do this, we rewrite the arithmetic instructions to indicate the source of their operands:

LDAC5: $AC \leftarrow BUS$
MOVR1: $AC \leftarrow BUS$
ADD1: $AC \leftarrow AC + BUS$
SUB1: $AC \leftarrow AC - BUS$
INAC1: $AC \leftarrow AC + 1$
CLAC1: $AC \leftarrow 0$

Each of these instructions can be implemented by using a parallel adder with carry in by modifying the input values, rewriting each operation as the sum of two values and a carry:

LDAC5: $AC \leftarrow 0 + BUS + 0$
MOVR1: $AC \leftarrow 0 + BUS + 0$
ADD1: $AC \leftarrow AC + BUS + 0$
SUB1: $AC \leftarrow AC + BUS' + 1$
INAC1: $AC \leftarrow AC + 0 + 1$
CLAC1: $AC \leftarrow 0 + 0 + 0$

Note that subtraction is implemented via two's complement addition, as described in Chapter 1. For now we design the data paths; we implement the control logic later in the design process.

The first input to the parallel adder is either the contents of *AC* or 0. The ALU can use a multiplexer to select one of these two values and pass it to one input of the parallel adder. Similarly, the ALU uses a multiplexer to send *BUS*, *BUS′*, or 0 to the second input. The ALU could also use a multiplexer to supply the carry input, but that would be overkill. We simply use a control input to directly generate this value.

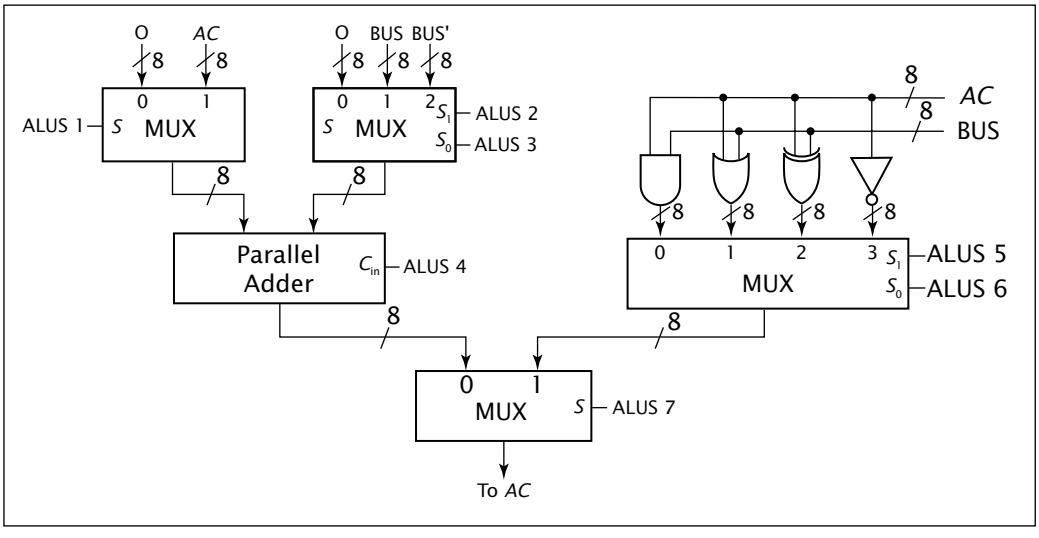
The logical operations are relatively straightforward. Since there are four logical operations, we use an 8-bit 4 to 1 multiplexer. The inputs to the MUX are $AC \wedge BUS$, $AC \vee BUS$, $AC \oplus BUS$, and AC'.

Finally, a multiplexer selects the output of either the parallel adder or the logic multiplexer to output to *AC*. The entire ALU design is shown in Figure 6.16 on page 248.

6.3.6 Designing the Control Unit Using Hardwired Control

The Relatively Simple CPU has a total of 37 states, making it too complex to implement efficiently using the same design as the Very Simple CPU's control unit. Instead of using one register to generate the state, this control unit uses two registers and combines their outputs to

Figure 6.16
A Relatively Simple ALU



generate the state value. One value is the opcode of the instruction. The other is a counter to keep track of which state in the fetch or execute routine should be active.

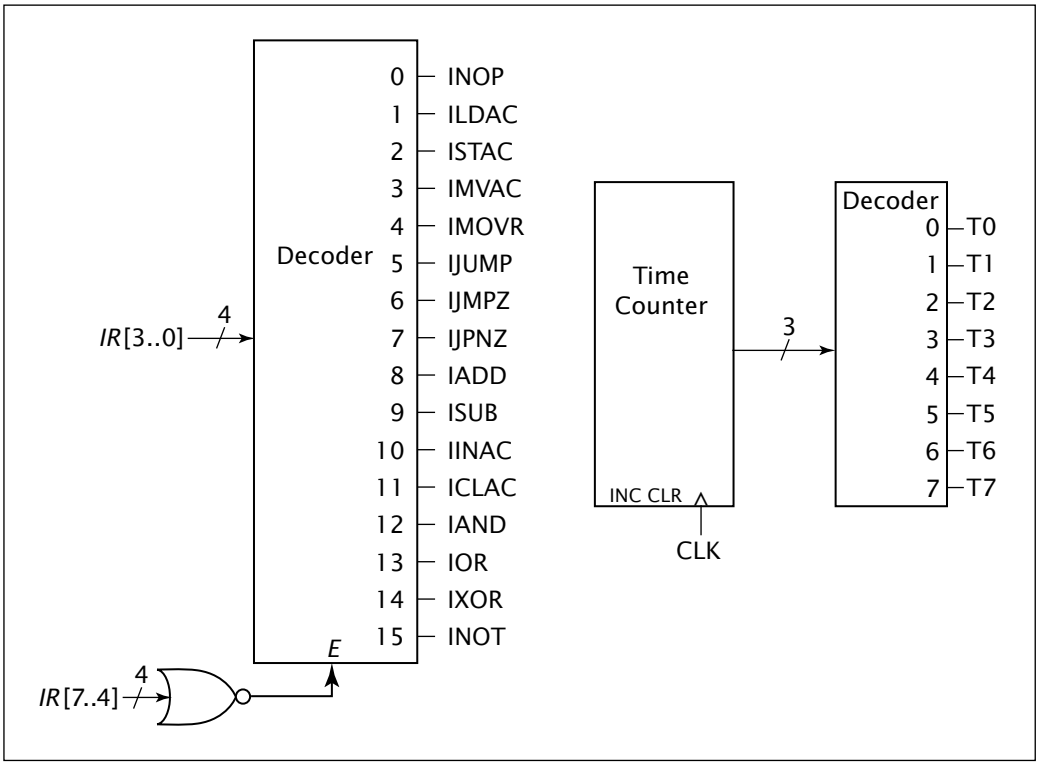
The opcode value is relatively easy to design. The opcode is stored in *IR*, so the control unit can use that register's outputs as inputs to a decoder. Since the instruction codes are all of the form 0000 XXXX, we only need to decode the four low-order bits. We NOR together the four high-order bits to enable the decoder. Then the counter can be set up so that it only has to be incremented and cleared, and never loaded; this greatly simplifies the design. These components, and the labels assigned to their outputs, are shown in Figure 6.17.

The fetch routine is the only routine that does not use a value from the instruction decoder. Since the instruction is still being fetched during these states, this decoder could have any value during the instruction fetch. Just as with the Very Simple CPU, this control unit assigns T0 to FETCH1, since it can be reached by clearing the time counter. We assign T1 and T2 to FETCH2 and FETCH3, respectively.

The states of the execute routines depend on both the opcode and time counter values. T3 is the first time state of each execute routine, T4 is the second, and so on. The control unit logically ANDs the correct time value with the output of the instruction multiplexer corresponding to the proper instruction. For example, the states of the LDAC execute routine are

$$\text{LDAC1} = \text{ILDAC} \wedge \text{T3}$$
$$\text{LDAC2} = \text{ILDAC} \wedge \text{T4}$$

Figure 6.17
Hardwired control unit for the Relatively Simple CPU



$$LDAC3 = ILDAC \wedge T5$$
$$LDAC4 = ILDAC \wedge T6$$
$$LDAC5 = ILDAC \wedge T7$$

The complete list of states is given in Table 6.6 on page 250.

Having generated the states, we must generate the signals to supply the CLR and INC inputs of the time counter. The counter is cleared only at the end of each execute routine. To do this, we logically OR the last state of each execute routine to generate the CLR input. The INC input should be asserted at all other times, so it can be implemented by logically ORing the remaining states together. As an alternative, the INC input can be the complement of the CLR input, since, if the control unit is not clearing the counter, it is incrementing the counter.

Following the same procedure we used for the Very Simple CPU, we generate the register and buffer control signals. Table 6.7 on page 251 shows the values for the buffers and *AR*. The remaining control signals are left as design problems for the reader.

Table 6.6
State generation for a Relatively Simple CPU

State	Function	State	Function
FETCH1	T0	JMPZY1	IJMPZ ∧ Z ∧ T3
FETCH2	T1	JMPZY2	IJMPZ ∧ Z ∧ T4
FETCH3	T2	JMPZY3	IJMPZ ∧ Z ∧ T5
NOP1	INOP ∧ T3	JMPZN1	IJMPZ ∧ Z′ ∧ T3
LDAC1	ILDAC ∧ T3	JMPZN2	IJMPZ ∧ Z′ ∧ T4
LDAC2	ILDAC ∧ T4	JPNZY1	IJPNZ ∧ Z ∧ T3
LDAC3	ILDAC ∧ T5	JPNZY2	IJPNZ ∧ Z ∧ T4
LDAC4	ILDAC ∧ T6	JPNZY3	IJPNZ ∧ Z ∧ T5
LDAC5	ILDAC ∧ T7	JPNZN1	IJPNZ ∧ Z′ ∧ T3
STAC1	ISTAC ∧ T3	JPNZN2	IJPNZ ∧ Z′ ∧ T4
STAC2	ISTAC ∧ T4	ADD1	IADD ∧ T3
STAC3	ISTAC ∧ T5	SUB1	ISUB ∧ T3
STAC4	ISTAC ∧ T6	INAC1	IINAC ∧ T3
STAC5	ISTAC ∧ T7	CLAC1	ICLAC ∧ T3
MVAC1	IMVAC ∧ T3	AND1	IAND ∧ T3
MOVR1	IMOVR ∧ T3	OR1	IOR ∧ T3
JUMP1	IJUMP ∧ T3	XOR1	IXOR ∧ T3
JUMP2	IJUMP ∧ T4	NOT1	INOT ∧ T3
JUMP3	IJUMP ∧ T5		

Finally, we generate the ALU control signals in the same manner. For example, ALUS1 = ADD1 ∨ SUB1 ∨ INAC1 and ALUS4 = SUB1 ∨ INAC1. The remaining control signals are left as exercises for the reader.

6.3.7 Design Verification

To verify the design of this CPU, the designer should prepare a trace of the execution, as was done for the Very Simple CPU. For the JMPZ and JPNZ instructions, the trace should show the execution under all possible circumstances, in this case $Z = 0$ and $Z = 1$. This is left as an exercise for the reader.

To perform the trace, students may use the RS-CPU simulator package. This package is a Java applet that can be run using any standard Web browser with Java enabled. Using this package, the reader can enter a program and step through the fetch, decode, and execution of the individual instructions. The package may be accessed at the textbook's companion Web site, along with its instructions.

Table 6.7
Control signal values for a Relatively Simple CPU

Signal	Value
PCBUS	FETCH1 ∨ FETCH3
DRHBUS	LDAC3 ∨ STAC3 ∨ JUMP3 ∨ JMPZY3 ∨ JPNZY3
DRLBUS	LDAC5 ∨ STAC5
TRBUS	LDAC3 ∨ STAC3 ∨ JUMP3 ∨ JMPZY3 ∨ JPNZY3
RBUS	MOVR1 ∨ ADD1 ∨ SUB1 ∨ AND1 ∨ OR1 ∨ XOR1
ACBUS	STAC4 ∨ MVAC1
MEMBUS	FETCH2 ∨ LDAC1 ∨ LDAC2 ∨ LDAC4 ∨ STAC1 ∨ STAC2 ∨ JUMP1 ∨ JUMP2 ∨ JMPZY1 ∨ JMPZY2 ∨ JPNZY1 ∨ JPNZY2
BUSMEM	STAC5
ARLOAD	FETCH1 ∨ FETCH3 ∨ LDAC3 ∨ STAC3
ARINC	LDAC1 ∨ STAC1 ∨ JUMP1 ∨ JMPZY1 ∨ JPNZY1

6.4 Shortcomings of the Simple CPUs

The CPUs presented in this chapter were designed as educational tools. Although they share many features with commonly used microprocessors, they are not representative of the current state of CPU design. Several common features were excluded from the Very Simple and Relatively Simple CPUs in an attempt to incorporate the essential features without overwhelming the reader. Consider the feature sets of these CPUs to be the result of an engineering education design tradeoff.

Following are some of the features found in many CPUs that are not present in either of the CPUs developed in this chapter.

6.4.1 More Internal Registers and Cache

One of the best ways to improve the performance of a microprocessor is to incorporate more storage within the CPU. Adding registers and cache makes it possible to replace some external memory accesses with much faster internal accesses.

To illustrate this, consider the ADD instructions for the Very Simple and Relatively Simple CPUs. The ADD instruction for the Very Simple CPU adds the contents of the accumulator to that of a memory location. It requires two states: one to read the value from memory (ADD1), and another to add the two values and store the result in the accumulator (ADD2). The Relatively Simple CPU, however, adds the contents of the accumulator and register *R*. Because the CPU does not access memory, it executes the ADD instruction in a single state

(ADD1). Removing memory accesses from other instructions by using internal registers reduces the time needed to execute the instructions in a similar manner.

Having more registers within the CPU also improves performance in programs that have subroutines. Consider a program for a CPU with no internal data registers, other than an accumulator. Assume this program invokes a subroutine, and this subroutine must receive six data values from the main program as passed parameters. The main program would have to write those six values to predetermined memory locations. The subroutine would have to read the values from memory and write its results back to memory. Finally, the main program would have to read the results from memory. If the CPU contained enough registers, the main program could store the parameters in its internal registers. The subroutine would not need to access memory because the CPU already contained the data in its registers. On completion, the main program would receive the results via the registers. Overall, a large number of memory accesses are thus avoided.

As processors have become more complex, designers have included more storage within the CPU, both in registers and internal cache memory. See Historical Perspective: Storage in Intel Microprocessors.

HISTORICAL PERSPECTIVE: Storage in Intel Microprocessors

Since the introduction of its first microprocessor in 1971, Intel has steadily increased the number of general purpose registers in its microprocessors. The 4004, Intel's first microprocessor, had no general purpose registers per se, although a complete 4-chip computer, consisting of the 4001, 4002, 4003, and 4004 chips, included 16 RAM locations that were used as registers. Its successors, the 8008, 8080, and 8085, incorporated six general purpose registers, as well as an accumulator, within the processor chip itself. The 8086 microprocessor has eight general purpose registers, as do the 80286, 80386, and 80486 microprocessors. The Pentium microprocessor also has 8 internal general purpose registers, but they are 32 bits wide, as opposed to the 16 bits of its predecessors. Intel's most recent microprocessor (as of this writing), the Itanium microprocessor, has 128 general purpose integer registers and an additional 128 general purpose floating point registers.

Intel first introduced cache memory into its Pentium microprocessor, starting with 16K of cache memory. It soon increased this to 32K, and further increased the amount in later processors. The Itanium microprocessor contains three levels of cache with over 4 MB of cache memory.

6.4.2 Multiple Buses Within the CPU

Buses are efficient media for routing data between components within a CPU. However, a bus may only contain one value at any given time. For that reason, most CPUs contain several buses. Multiple buses allow multiple data transfers to occur simultaneously, one via each bus. This reduces the time needed to fetch, decode, and execute instructions, thus improving system performance.

Consider the register section of the Relatively Simple CPU, shown in Figure 6.15. Most data transfers are routed via the 16-bit bus; every register except *IR* either outputs data to or inputs data from the bus. However, most of these components never need to communicate with each other. For example, it is possible to route data from *R* to *AR*, but it is never necessary to do so. If multiple buses were used, components that transfer data among themselves could be connected to more than one bus, or there could be connections established between buses. For the Relatively Simple CPU, one bus could be set up for address information and another for data. One possible configuration, which uses three buses, is shown in Figure 6.18 on page 254.

Another benefit of multiple buses is the elimination of direct connections between components. Recall that the Relatively Simple CPU included direct connections from *DR* to *TR* and *IR* so that multiple data transfers could occur simultaneously during FETCH3, LDAC2, STAC2, JUMP2, JMPZY2, and JPNZY2. As the number of registers within the CPU increases, this becomes increasingly important.

6.4.3 Pipelined Instruction Processing

In the CPUs developed in this chapter, one instruction is fetched, decoded, and executed before the next instruction is processed. In pipelining, instructions are processed like goods on an assembly line. While one instruction is being decoded, the next instruction is fetched, and while the first instruction is being executed, the second is decoded and a third instruction is fetched. Overlapping the fetch, decode, and execute of several instructions allows programs to be executed more quickly, even though each individual instruction requires the same amount of time.

Although this process has some problems, particularly with conditional and unconditional jump instructions, it offers significant increases in performance. Pipelining is discussed in detail in Chapter 11.

6.4.4 Larger Instruction Sets

Having a larger number of instructions in a processor's instruction set generally allows a program to perform a function using fewer instructions. For example, consider a CPU that can logically AND two values and complement one value, but cannot logically OR two values. To

Figure 6.18

Register section for the Relatively Simple CPU using multiple buses

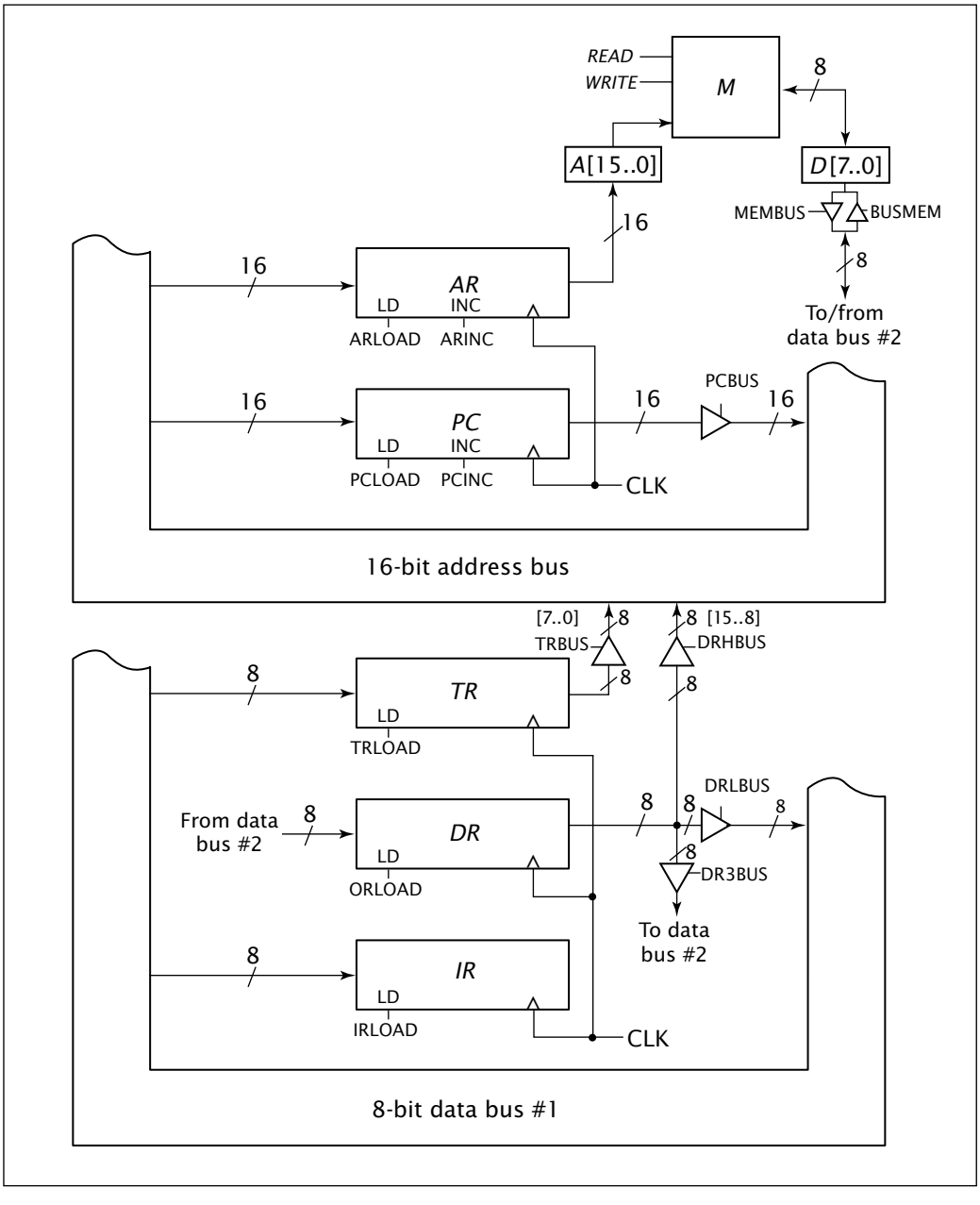
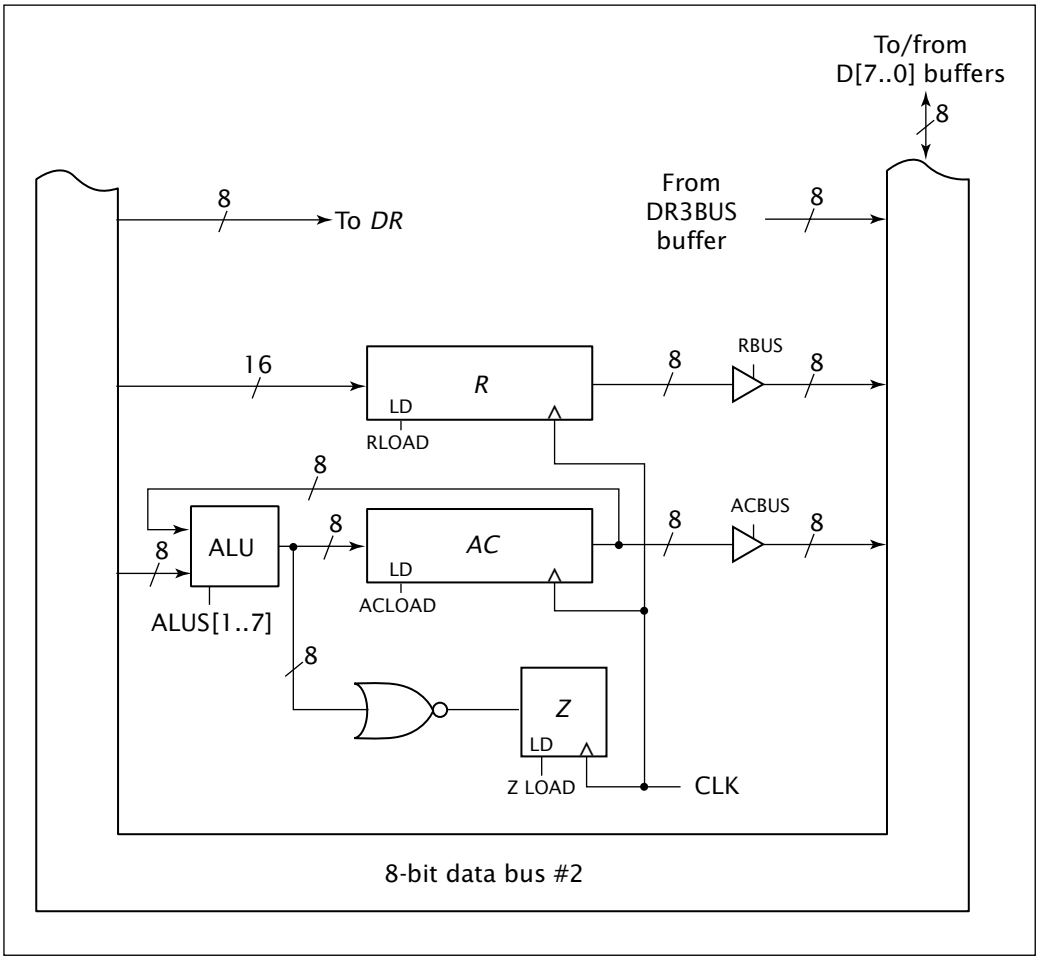


Figure 6.18
(continued)



perform a logical OR of two values, A and B, it would perform the following instructions:

Complement A
Complement B
AND A and B
Complement the result

If this CPU contained an OR instruction, only one instruction would be needed instead of four.

There is considerable debate over how many instructions a CPU should have. As the number of instructions increases, so does the time needed to decode the instructions, which limits the clock speed of the

CPU. This is the basis of the complex versus reduced instruction set computing debate, which is examined more closely in Chapter 11.

6.4.5 Subroutines and Interrupts

Almost all CPUs have hardware to handle subroutines, typically a stack pointer, and instructions to call and return from the subroutine. Most CPUs also have interrupt inputs to allow external hardware to interrupt the current operations of the CPU. This is useful for such things as updating the computer's display, since it is preferable for the CPU to perform useful work and to be interrupted when the screen must be refreshed, rather than spending time polling the screen controller to determine whether it needs to be updated. Interrupts are described in more detail in Chapter 10.

6.5 Real World Example: Internal Architecture of the 8085 Microprocessor

In Chapters 3 and 4, we examined the instruction set architecture of the 8085 microprocessor and a computer based on this microprocessor. In this section, we look inside the 8085 and compare its organization to that of the Relatively Simple CPU.

The internal organization of Intel's 8085 microprocessor is shown in Figure 6.19. (Note that some elements of the design, such as internal control signals, are present but not shown in the figure.) As with the other CPUs described so far, the 8085 contains a register section, an ALU and a control unit. Note that the interrupt control and serial I/O control blocks are not exclusively a part of any one section. In fact, part of these blocks are components of the control unit and the rest of the blocks are parts of the register section. Let's look at these sections in some detail.

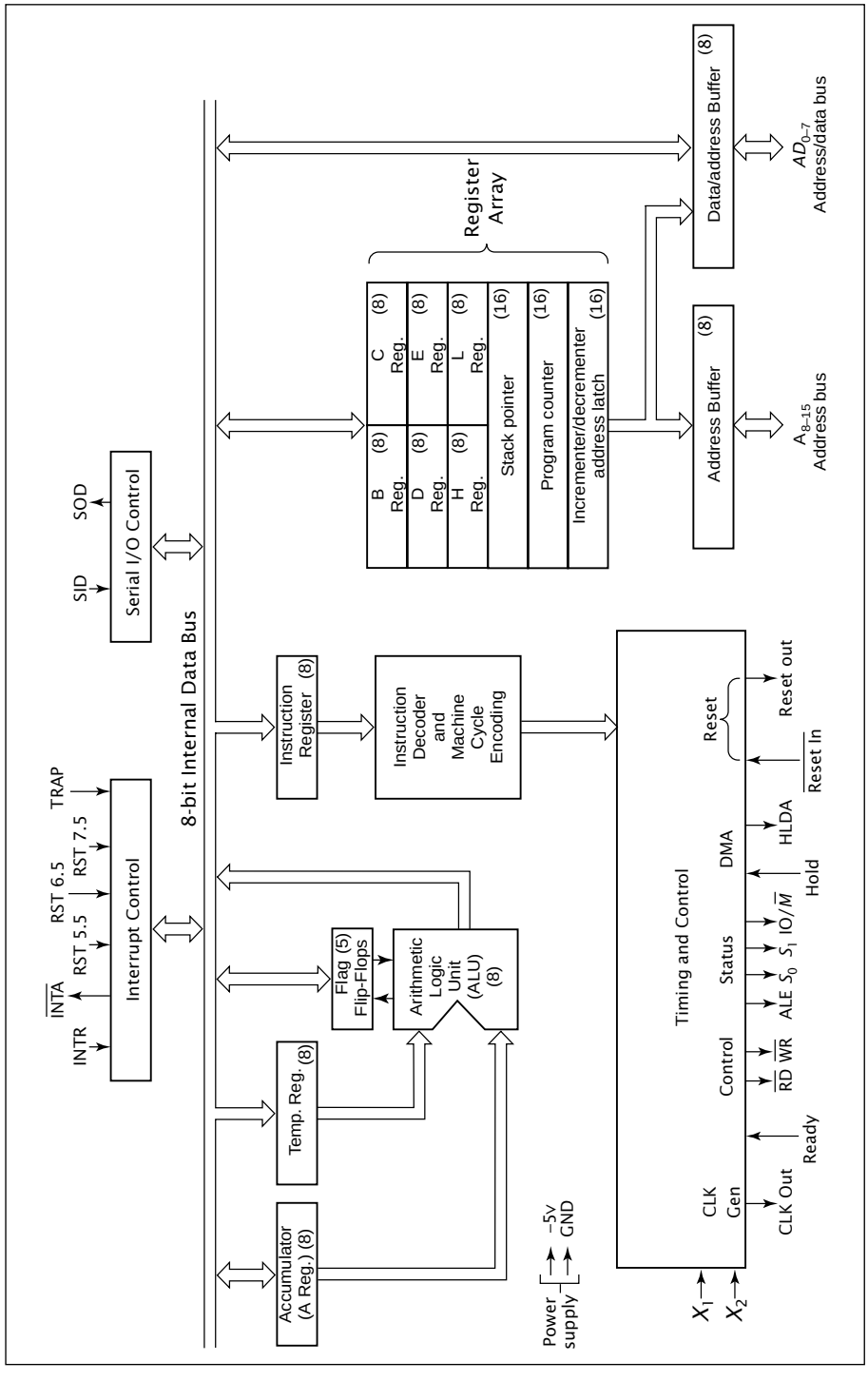
The easiest component to examine is the 8085's ALU. It performs all arithmetic, logic, and shift instructions, making its result available to the registers via the 8-bit internal data bus. Control signals from the control unit, not shown in Figure 6.19, select the function to be performed by the ALU.

The register section contains the user-accessible registers specified in the 8085's instruction set architecture: *A*, *B*, *C*, *D*, *E*, *H*, *L*, *SP*, and the flags. This section also contains the microprocessor's instruction register and program counter, a temporary register that it uses to input data to the ALU, and an address latch, which is equivalent to the *AR* register in the Relatively Simple CPU. Although not shown in Figure 6.19, two additional temporary registers are used by the microprocessor to store data during the execution of an instruction. They serve the same purpose as the *TR* register in the Relatively Simple CPU.

Although not registers, the address and data/address buffers are included in the register section. Under certain conditions, the 8085 does not access the system address and data buses. During these

Figure 6.19

Internal organization of the 8085 microprocessor (MCS 80/85™ Family User's Manual. Reprinted by permission of Intel Corporation, Copyright Intel Corporation 1979.)



times, it must tri-state its connections to these buses; this is the function of these buffers. This happens when the computer is performing a DMA transfer, described in detail in Chapter 10. In addition, the data/address buffers determine whether data is input to or output from the CPU, just as was done with the Relatively Simple CPU.

The interrupt control block contains the interrupt mask register. The user can read the value from this register or store a value into that register, so it is included in the microprocessor's instruction set architecture and its register section. The serial I/O control block also contains a register to latch serial output data.

The registers communicate within the CPU via the 8-bit internal data bus. Although it is not very clear in Figure 6.19, the connection from the register array (the block containing registers *B*, *C*, *D*, *E*, *H*, *L*, *SP*, and *PC*) is wide enough for one register to place data onto the bus while another register reads the data from the bus, as when the instruction MOV B,C is executed. When data is read from memory, such as during an instruction fetch, or from an I/O device, the data is passed through the data/address buffer on to the internal data bus. From there, it is read in by the appropriate register.

The control section consists of several parts. The timing and control block is equivalent to almost the entire control unit of the Relatively Simple CPU. It sequences through the states of the microprocessor and generates external control signals, such as those used to read from and write to memory. Although not shown, it also generates all of the internal control signals used to load, increment and clear registers; to enable buffers; and to specify the function to be performed by the ALU.

The instruction decoder and machine cycle encoding block takes the current instruction (stored in the instruction register) as its input and generates state signals that are input to the timing and control block. This is similar to the function performed by the 4-to-16 decoder in the control unit of the Relatively Simple CPU, as shown in Figure 6.17. Essentially, it decodes the instruction. The decoded signals are then combined with the timing signals in the timing and control block to generate the internal control signals of the microprocessor.

Finally, the interrupt control and serial I/O control blocks are partially elements of the control unit. The interrupt control block accepts external interrupt requests, checks whether the requested interrupts are enabled, and passes valid requests to the rest of the control unit. (As with the internal control signals, the path followed by these requests is not shown in Figure 6.19 but it is present nonetheless.) The serial I/O control block contains logic to coordinate the serial transfer of data into and out of the microprocessor.

The 8085 microprocessor addresses several but not all of the shortcomings of the Relatively Simple CPU. First of all, it contains more general purpose registers than the Relatively Simple CPU. This allows the 8085 to use fewer memory accesses than the Relatively

Simple CPU to perform the same task. The 8085 microprocessor also has a larger instruction set, and has the ability to handle subroutines and interrupts. However, it still uses only one internal bus to transfer data, which limits the number of data transfers that can occur at any given time. The 8085 also does not use an instruction pipeline. Like the Relatively Simple CPU, it processes instructions sequentially—it fetches, decodes, and executes one instruction before fetching the next instruction.

6.6 Summary

In previous chapters, we looked at the CPU from the point of view of the programmer (instruction set architecture) and the system designer (computer organization). In this chapter, we examined the CPU from the perspective of the computer architect.

To design a CPU, we first develop its instruction set architecture, including its instruction set and its internal registers. We then create a finite state machine model of the micro-operations needed to fetch, decode, and execute every instruction in its instruction set. Then we develop an RTL specification for this state machine.

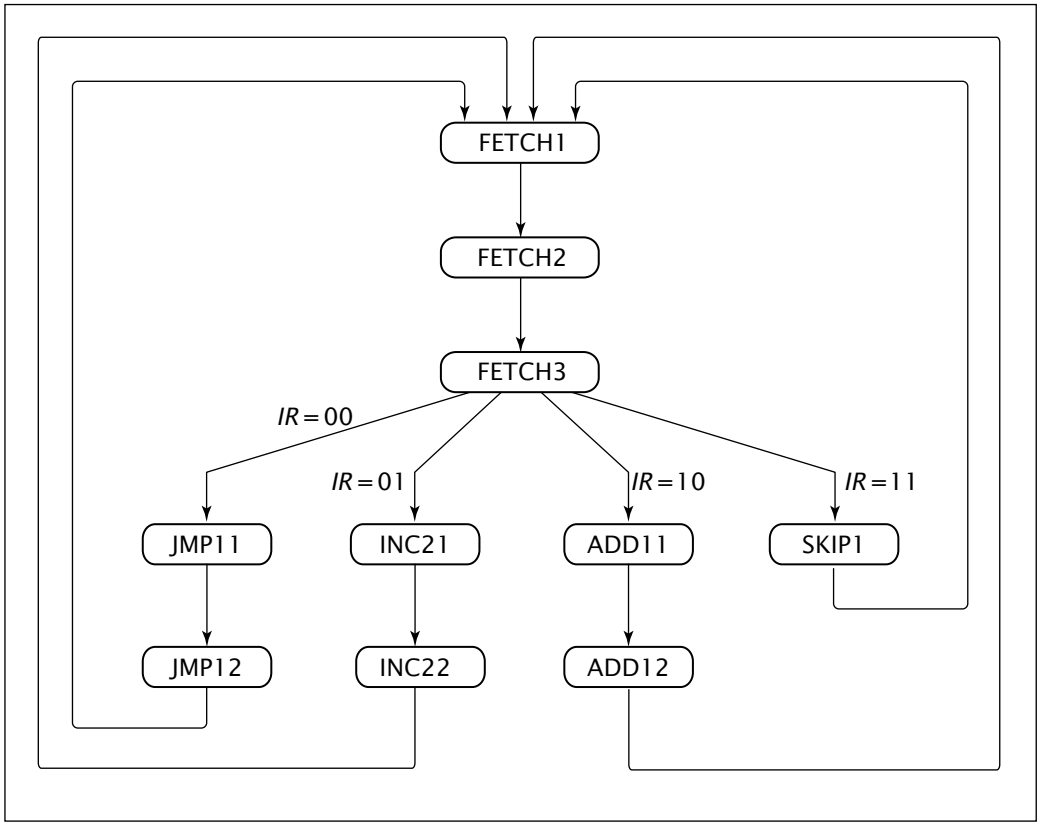
A CPU contains three primary sections: the register section, consisting of the registers in the CPU's ISA as well as other registers not directly available to the programmer, the ALU, and the control unit. The micro-operations in its RTL code specify the functions to be performed by the register section and the ALU. These micro-operations are used to design the data paths within the register section, including direct connections and buses, and the functions of each register. The micro-operations also specify the functions of the ALU. Since the ALU must perform all of its calculations in a single clock cycle, it is constructed using only combinatorial logic.

The conditions under which each micro-operation occurs dictate the design of the control unit. The control unit generates the control signals that load, increment, and clear the registers in the register section. The control unit also enables the buffers used to control the CPU's internal buses. The function to be performed by the ALU is specified by the control unit. By outputting the control signals in the proper sequence, the control unit causes the CPU to properly fetch, decode, and execute every instruction in its instruction set.

Problems

1 A CPU with the same registers as the Very Simple CPU, connected as shown in Figure 6.6, has the following instruction set and state diagram. Show the RTL code for the execute cycles for each instruction. Assume the RTL code for the fetch routine is the same as that of the Very Simple CPU.

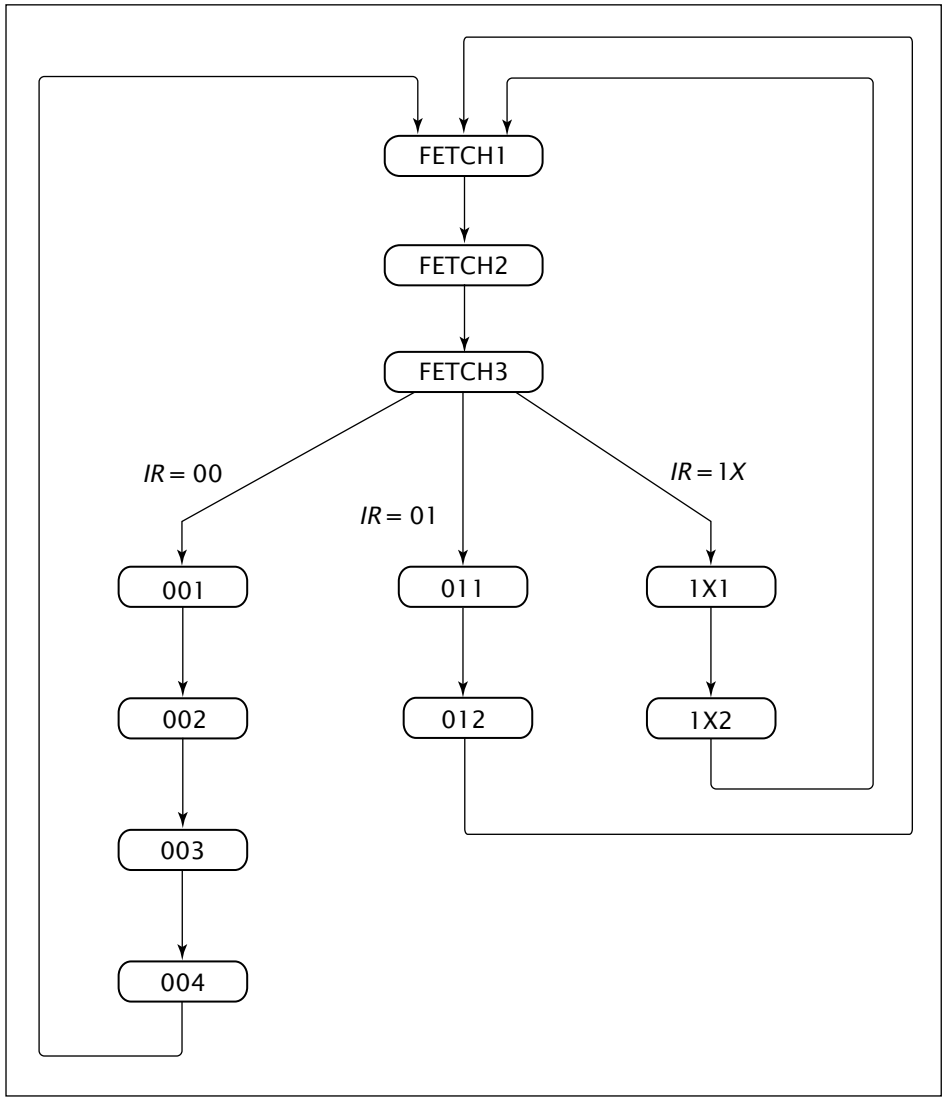
Instruction	Instruction Code	Operation
JMP1	00AAAAAA	*PC*←AAAAAA + 1
INC2	01XXXXXX	*AC*←*AC* + 2
ADD1	10AAAAAA	*AC*←*AC* + *M*[AAAAAA] + 1
SKIP	11XXXXXX	*PC*←*PC* + 1



2 A CPU with the same registers as the Very Simple CPU, connected as shown in Figure 6.6, has the state diagram on the next page and following RTL code. Show the instruction set for this CPU.

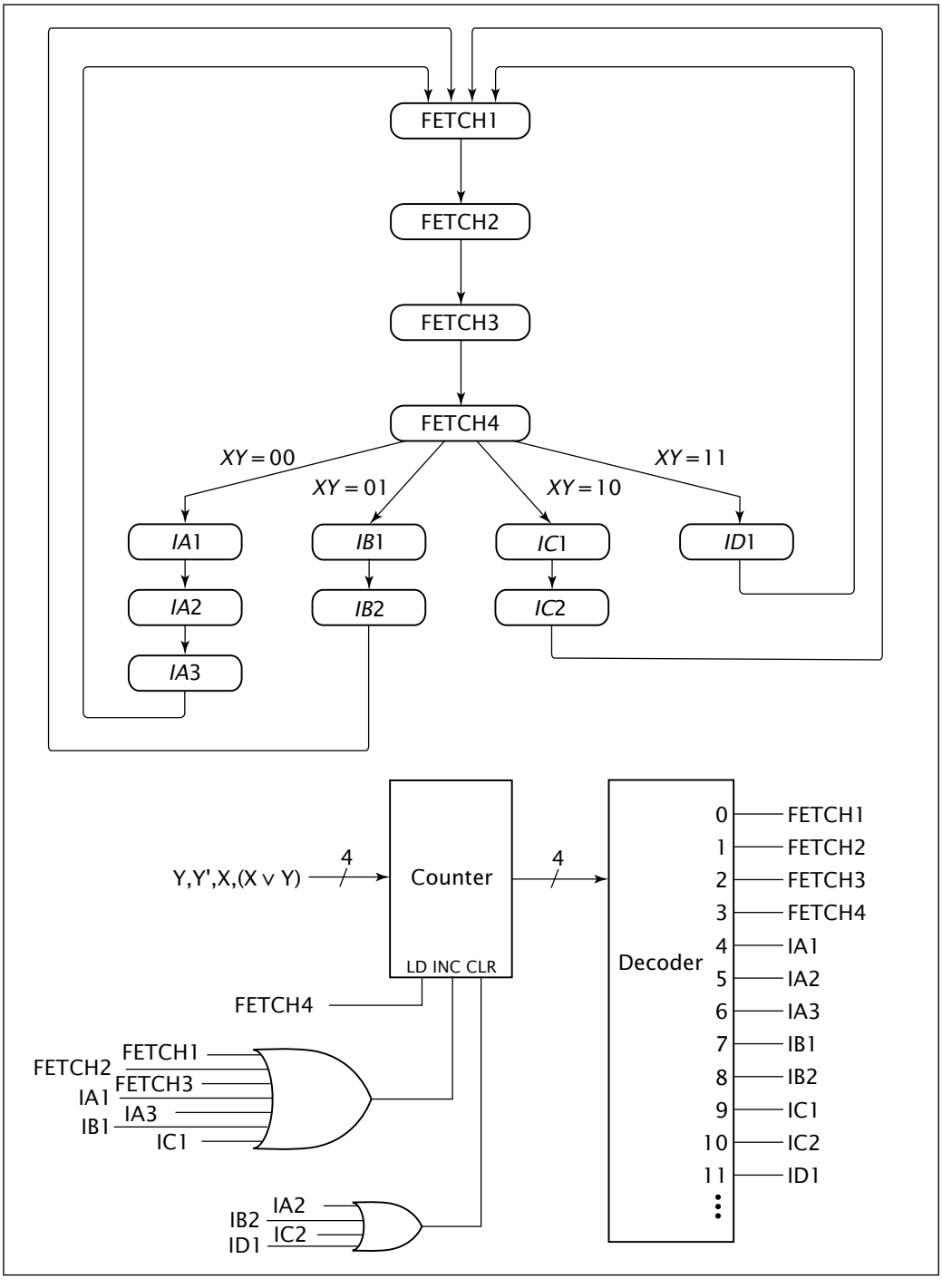
FETCH1: *AR*←*PC*
FETCH2: *DR*←*M*, *PC*←*PC* + 1
FETCH3: IR←*DR* [7..6], *AR*←*DR*[5..0]
001: *DR*←*M*, *AR*←*AR* + 1
002: AC←AC + *DR*
003: *DR*←*M*

004: $AC \leftarrow AC + DR$
011: $DR \leftarrow M$, $PC \leftarrow PC + 1$
012: $AC \leftarrow AC \wedge DR$
1X1: $AC \leftarrow AC + 1$, $DR \leftarrow M$
1X2: $AC \leftarrow AC \wedge DR$



3 Develop a control unit for the state diagram in Problem 2.

4 The following control unit is supposed to realize the state diagram, also shown, but it does not. Show the state diagram it actually realizes.



5 Modify the control unit of Problem 4 so that it realizes the state diagram properly.

6 We wish to modify the Very Simple CPU to incorporate a new instruction, CLEAR, which sets $AC \leftarrow 0$; the instruction code for CLEAR is 111X XXXX. The new instruction code for INC is 110X XXXX; all other instruction codes remain unchanged. Show the new state diagram and RTL code for this CPU.

7 For the CPU of Problem 6, show the modifications necessary for the register section.

8 For the CPU of Problem 6, show the modifications necessary for the control unit. Include the hardware needed to generate any new or modified control signals.

9 Verify the functioning of the CPU of Problems 6, 7, and 8 for the new instruction.

10 We wish to modify the Very Simple CPU to incorporate a new 8-bit register, R, and two new instructions. MVAC performs the transfer $R \leftarrow AC$ and has the instruction code 1110 XXXX; MOVR performs the operation $AC \leftarrow R$ and has the instruction code 1111 XXXX. The new instruction code for INC is 110X XXXX; all other instruction codes remain unchanged. Show the new state diagram and RTL code for this CPU.

11 For the CPU of Problem 10, show the modifications necessary for the register section.

12 For the CPU of Problem 10, show the modifications necessary for the control unit. Include the hardware needed to generate any new or modified control signals.

13 Verify the functioning of the CPU of Problems 10, 11, and 12 for the new instructions.

14 Enhance the Very Simple ALU to perform the following operations, in addition to those it currently performs.

shl: $AC \leftarrow AC + AC$
neg: $AC \leftarrow AC' + 1$
ad1: $AC \leftarrow AC + DR + 1$

15 Show the logic needed to generate the control signals for registers PC, DR, TR, and IR of the Relatively Simple CPU.

16 Show the logic needed to generate the control signals for registers *R*, *AC*, and *Z* of the Relatively Simple CPU.

17 Show the logic needed to generate the control signals for the ALU of the Relatively Simple CPU.

18 Verify the functioning of the Relatively Simple CPU for all instructions, either manually or using the CPU simulator.

19 Modify the Relatively Simple CPU to include a new instruction, SETR, which performs the operation $R \leftarrow 1111\ 1111$. Its instruction code is 0001 0000. Show the modified state diagram and RTL code for this CPU. (*Hint:* One way to implement this is to clear *R* and then decrement it.)

20 For the CPU of Problem 19, show the modifications necessary for the register section.

21 For the CPU of Problem 19, show the modifications necessary for the control unit. Include the hardware needed to generate any new or modified control signals.

22 Verify the functioning of the CPU of Problems 19, 20, and 21 for the new instruction.

23 Modify the Relatively Simple CPU to include a new 8-bit register, *B*, and five new instructions as follows. Show the modified state diagram and RTL code for this CPU.

Instruction	Instruction Code	Operation
ADDB	0001 1000	$AC \leftarrow AC + B$
SUBB	0001 1001	$AC \leftarrow AC - B$
ANDB	0001 1100	$AC \leftarrow AC \wedge B$
ORB	0001 1101	$AC \leftarrow AC \vee B$
XORB	0001 1110	$AC \leftarrow AC \oplus B$

24 For the CPU of Problem 23, show the modifications necessary for the register section and the ALU.

25 For the CPU of Problem 23, show the modifications necessary for the control unit. Include the hardware needed to generate any new or modified control signals.

26 Verify the functioning of the CPU of Problems 23, 24, and 25 for the new instructions.

27 For the Relatively Simple CPU, assume the CLAC and INAC instructions are implemented via the CLR and INC signals of the AC register, instead of through the ALU. Modify the input and control signals of *Z* so it is set properly for all instructions.

28 Design a CPU that meets the following specifications.

- It can access 64 words of memory, each word being 8 bits wide. The CPU does this by outputting a 6-bit address on its output pins A[5..0] and reading in the 8-bit value from memory on its inputs D[7..0].
- The CPU contains a 6-bit address register (*AR*) and program counter (*PC*); an 8-bit accumulator (*AC*) and data register (*DR*); and a 2-bit instruction register (*IR*).
- The CPU must realize the following instruction set.

Instruction	Instruction Code	Operation
COM	00XXXXXX	$AC \leftarrow AC'$
JREL	01AAAAAA	$PC \leftarrow PC + 00AAAAAA$
OR	10AAAAAA	$AC \leftarrow AC \vee M[00AAAAAA]$
SUB1	11AAAAAA	$AC \leftarrow AC - M[00AAAAAA] - 1$

29 Design a CPU that meets the following specifications.

- It can access 256 words of memory, each word being 8 bits wide. The CPU does this by outputting an 8-bit address on its output pins A[7..0] and reading in the 8-bit value from memory on its inputs D[7..0].
- The CPU contains an 8-bit address register (*AR*), program counter (*PC*), accumulator (*AC*), and data register (*DR*), and a 3-bit instruction register (*IR*).
- The CPU must realize the following instruction set. Note that α is an 8-bit value stored in the location immediately following the instruction.

Instruction	Instruction Code	Operation
LDI	000XXXXX α	$AC \leftarrow \alpha$
STO	001XXXXX α	$M[\alpha] \leftarrow AC$
ADD	010XXXXX α	$AC \leftarrow AC + M[\alpha]$
OR	011XXXXX α	$AC \leftarrow AC \vee M[\alpha]$
JUMP	100XXXXX α	$PC \leftarrow \alpha$
JREL	101AAAAA	$PC \leftarrow PC + 000AAAAA$
SKIP	110XXXXX	$PC \leftarrow PC + 1$
RST	111XXXXX	$PC \leftarrow 0$, $AC \leftarrow 0$

30 Modify the Relatively Simple CPU so that it can use a stack. The changes required to do this are as follows.

- Include a 16-bit stack pointer (*SP*) register that holds the address of the top of the stack.
- The CPU must realize the following additional instructions. Note that operations separated by semicolons occur sequentially, and operations separated by commas occur simultaneously. Also note that the value of *PC* used by the CALL instruction is the value of *PC* after Γ has been fetched from memory.

Instruction	**Instruction Code**	**Operation**
LDSP	10000000 Γ	$SP \leftarrow \Gamma$
CALL	10000010 Γ	$SP \leftarrow SP - 1$; $M[SP] \leftarrow PC[15..8]$, $SP \leftarrow SP - 1$; $M[SP] \leftarrow PC[7..0]$, $PC \leftarrow \Gamma$
RET	10000011	$PC[7..0] \leftarrow M[SP]$, $SP \leftarrow SP + 1$; $PC[15..8] \leftarrow M[SP]$, $SP \leftarrow SP + 1$
PUSHAC	10000100	$SP \leftarrow SP - 1$; $M[SP] \leftarrow AC$
POPAC	10000101	$AC \leftarrow M[SP]$, $SP \leftarrow SP + 1$
PUSHR	10000110	$SP \leftarrow SP - 1$; $M[SP] \leftarrow R$
POPR	10000111	$R \leftarrow M[SP]$, $SP \leftarrow SP + 1$