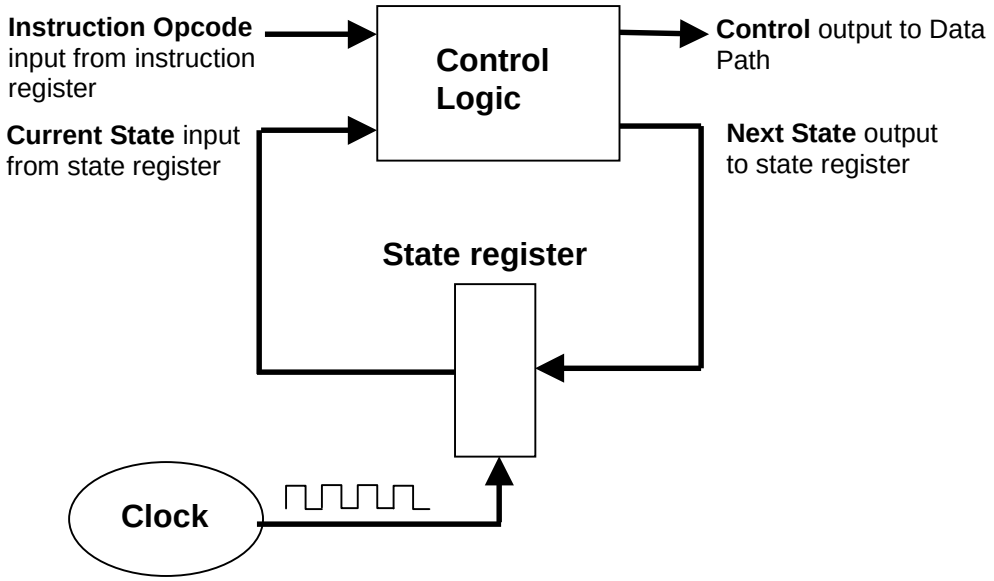
The Machine within the Machine--Control Logic

The data path described in the previous chapter is the tool kit of the computer. The registers and multiplexors, together with the ALU and the memory, have the power to perform all the operations of the instruction set we designed earlier. However, it is the control logic that really runs the computer. It coordinates and controls all the data path elements. The control logic takes as its input the instruction operation code, and generates a sequence of outputs that causes the data path to manipulate data or alter the flow of the computer program.

For each instruction, the control logic defines a series of steps. With each step, the control logic generates specific outputs. These outputs are fed to the data path multiplexor and register write inputs. Each step of the sequence is designed to be performed in a single clock cycle. If we want to perform a task that does not fit into one clock cycle, we need to break it up into sub-steps. The control logic keeps track of these steps by assigning a unique number to each one, called the state. The state is kept in a register in the control logic circuitry.

A device that uses a finite number of states in specified sequences to control a process is called a finite state machine. The machine works as follows. The cycle starts at some initial entry state, which we will assign the number 0. This value is placed into the state register. The control logic then looks at the state register and the current instruction opcode. Using standard AND/OR logic gates, it then creates two outputs. One output is the pattern of 1's and 0's that is sent to the data path elements (multiplexors and registers) to perform data and program flow manipulations. The other output is the next state. At the end of the clock cycle, the next state is written into the state register. At this point, the next state becomes the current state for the next cycle.



The state register is written every clock cycle, and therefore the computer clock is directly connected to its write input.

Different instructions can use the same states. For example, all instructions will start with state 0, which is the instruction fetch/program counter incrementation step. This will always be followed by state 1, which is the instruction interpretation step. If the processor is executing an arithmetic/logic instruction that has two operands, it will use a data fetch step. All arithmetic/logic instructions can use the same state for the data fetch step. The arithmetic/logical instructions that write the carry flip-flop can all use the same state, and the ones that do not can all use the same state. When the final state of a sequence is finished, the next state logic will give a 0 as its output, and a new sequence will start. Once the sequence of states is built into the control logic, the cycle operates endlessly.

How many states are required? Our instruction set has 16 operations, and each operation will need from 2 to 4 steps (or states) to operate. However, many states are shared by several operations, and this reduces the total number of states required. In fact, our computer can be built with a control logic that uses only 11 states (0 to 10). Modern complex instruction set processors, such as the Pentium, need hundreds of states.

By looking at the data path diagram, and thinking about what needs to happen in order for an instruction to be carried out, we can make a list of the states that will do everything our instruction set needs to do. Here are the states for this processor. The numbers we assign are somewhat arbitrary, although it is important to make state 0 the first step. This allows us to use a simple circuit to start the computer.

State	**Action**
0	Instruction fetch/program counter incrementation
1	Instruction interpretation
2	Data fetch
3	Arithmetic instruction, includes carry write
4	Logic instruction, no carry write
5	Load accumulator immediate (value in current instruction)
6	Load accumulator from memory
7	Store accumulator to memory, first step
8	Store accumulator to memory, second step
9	Jump immediate
10	Jump indirect

By referring to these states, we can now associate each instruction in our instruction set with the sequence of states needed to carry it out. We have already shown how the sequence of states 0,1,2,3 will perform the ADD operation. Here is the complete instruction set with the corresponding state sequences.

Instruction Opcode	Condition	Opcode Mnemonic	State sequence
0000		ADD	0,1,2,3
0001		ADC	0,1,2,3
0010		SUB	0,1,2,3
0011		SBC	0,1,2,3
0100		AND	0,1,2,4
0101		OR	0,1,2,4
0110		XOR	0,1,2,4
0111		NOT	0,1,4
1000		LDI	0,1,5
1001		LDM	0,1,6
1010		STM	0,1,7,8
1011		JMP	0,1,9
1100		JPI	0,1,10
1101	Z = 0 (not zero)	JPZ	0,1
1101	Z = 1 (zero)	JPZ	0,1,9
1110	M = 0 (not minus)	JPM	0,1
1110	M = 1 (minus)	JPM	0,1,9
1111	C = 0 (not carry)	JPC	0,1
1111	C = 1 (carry)	JPC	0,1,9

Notice that the arithmetic operations all use the same sequence of states. The difference between addition and subtraction, and the use of the carry-in, depends on the three-bit ALU opcode that is imbedded in the instruction opcode, so we do not need separate states for each. Similarly, the logical operations share the same sequence, the difference being that state 5 does not write the carry-out to the carry flip-flop. The NOT operation, which does not have a second operand, skips the data fetch state. The conditional jump instructions use the same state sequence as the unconditional direct jump when the corresponding condition is met. If the condition is not met, the direct jump state is simply omitted and the instruction performs no operation at all. Since each state is one clock cycle long, we can tell how long each instruction will take to execute. If we assume a clock rate of one megahertz, then the processor should be able to execute an addition (or any other 4-state instruction) in 4 microseconds.

We will create the control logic so that the control output depends only on the current state. There are other ways to do this, such as making the control output for the conditional jumps depend on the condition, but I think this way is simpler. I will now show a table of the control outputs that correspond to each state. There is one control output for each data path control input. Note that a three-input multiplexor needs two control inputs, for the low order and high order bits (bits 0 and 1). Refer to the data path diagram for the elements that use control inputs.

Control input	State										
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Program Counter Multiplexor Low	1	X	X	X	X	X	X	X	X	0	0
Program Counter Multiplexor High	0	X	X	X	X	X	X	X	X	0	1
Program Counter Write	1	0	0	0	0	0	0	0	0	1	1
Memory Address Multiplexor	1	X	0	X	X	X	0	0	0	X	0
Memory Write	0	0	0	0	0	0	0	1	0	0	0
Memory Data Out	0	0	0	0	0	0	0	0	1	0	0
Instruction Register Write	1	0	0	0	0	0	0	0	0	0	0
Accumulator Multiplexor Low	X	X	X	0	0	1	0	X	X	X	X
Accumulator Multiplexor High	X	X	X	0	0	0	1	X	X	X	X
Accumulator Write	0	0	0	1	1	1	1	0	0	0	0
Data Register Write	0	0	1	0	0	0	0	0	0	0	0
ALU A Multiplexor	1	X	X	0	0	X	X	X	X	X	X
ALU B Multiplexor	1	X	X	0	0	X	X	X	X	X	X
ALU Operation Multiplexor	1	X	X	0	0	X	X	X	X	X	X
Carry flip-flop Write	0	0	0	1	0	0	0	0	0	0	0

X = don't care

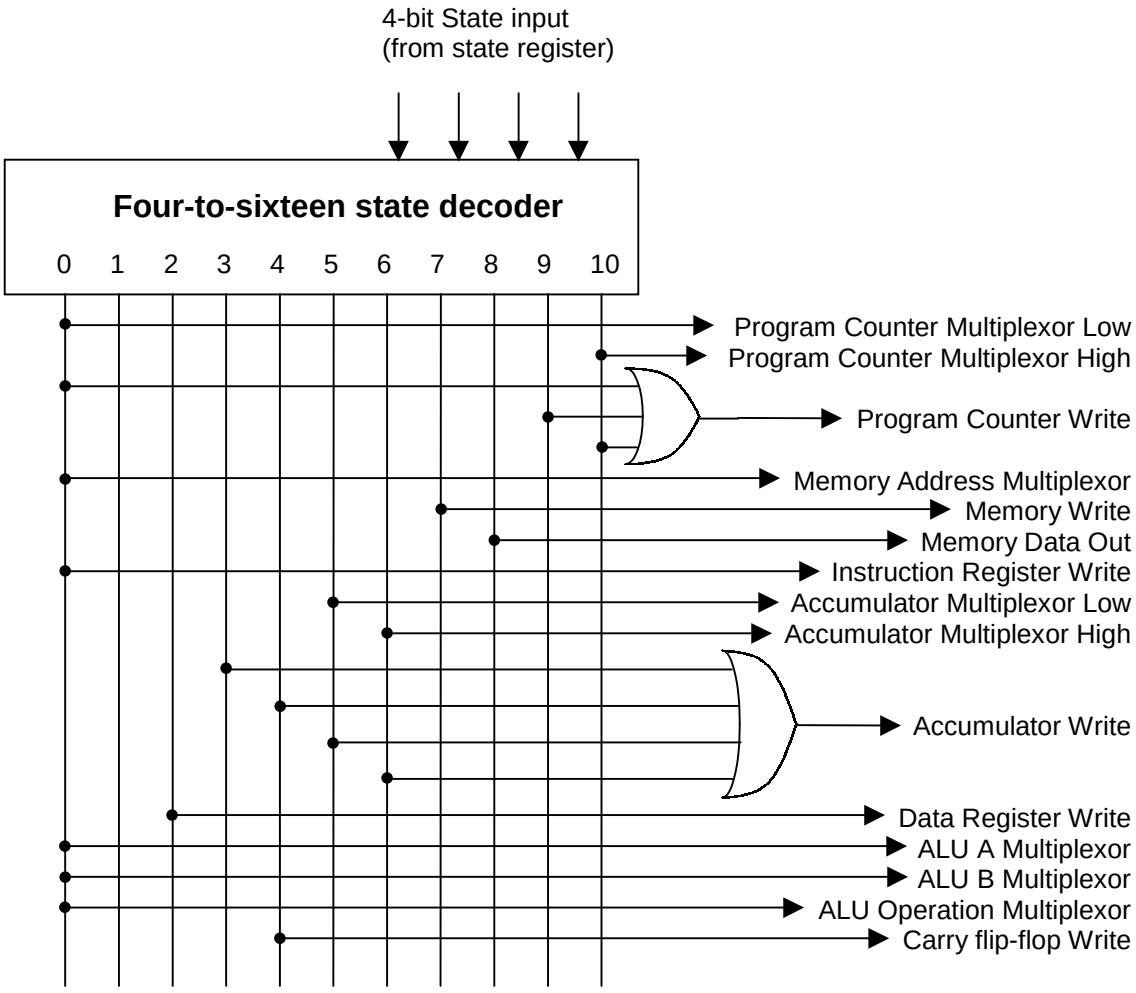
Notice that we don't care what the multiplexor control inputs are if we are not going to write the registers (or memory) whose input is supplied by that multiplexor. The "don't cares" allow us to design the logic more efficiently. Also, note that some of the control lines behave identically across all states. It is easy to see that the ALU A and B multiplexor control lines, and the ALU operation multiplexor control line could all be replaced by a single control line. Other control lines might also be consolidated, such as the program counter multiplexor low and the instruction register write. However, I felt that extensive consolidation of control lines might be confusing when it came to troubleshooting the processor, so I only consolidated the ALU multiplexor lines. It is more important to minimize the number of states needed. In the control design above, no two states can be consolidated.

In order to make a circuit that will "map" one state to one control output we have two main options. The option used by most processor manufacturers is to use microcode. The microcode technique uses a read-only memory to hold the control output configurations, in the form of binary numbers. The control output for each state in the above table could be considered as a 15-bit binary number. We can store the control output configuration for each state in such a way that Carry Flip-Flop Write is in the rightmost (low) bit, and the Program Counter Multiplexor Low is in the leftmost (high) bit. If we replace the don't cares with 0, then the control line output for state 0 would be the binary number 101 1001 0000 1110. The control line output simply becomes a 15-bit wide memory with 11 locations, one for each state.

For the hobbyist, this arrangement is attractive because the control output function can be changed without rewiring circuits. However, it requires the programming of ROM, and this requires extra equipment. But it is likely that you will need a PROM programmer anyway, so it is reasonable to consider this.

The other way to make a circuit that "maps" a state onto a control output is to use a decoder, together with AND-OR logic. In this method, the control output for each state is hard-wired into the board. If you like to wire things up, this is the way to go, and this is the way I did it.

Without showing the whole circuit, here is an example of how the states might be decoded into the corresponding control outputs



The 4-bit state value is placed on the decoder inputs, and the corresponding output becomes 1. The other outputs are 0. The Program Counter Write and Accumulator Write control lines are on in several different states, so multiple input OR gates are required to provide these outputs. The other control lines are connected directly to only one state output. If you compare this circuit with the table on the previous page, replacing the X

don't cares with zeros, you can see that this simple circuit faithfully provides the appropriate control outputs. This circuit can be built with one 4-to-16 decoder and 6 two-input OR gates, a total of 3 IC's. The control logic is half done.

The other function of the control logic is to provide the sequence of states for each instruction. We call this the next-state function. The next-state function circuitry takes as its inputs the current state, the current instruction, and the conditions (zero, minus and carry). From these inputs, the next state is determined.

The next-state function circuit is more difficult to design than the control output circuit. This is because it has a much more complex input. If we count each bit of the current state, current instruction and conditions, there are 4 + 4 + 3 = 11 bits of input, instead of the 4 bits of input to the control output circuit. Eleven bits can encode over 2,000 different combinations instead of the 16 handled by the simple decoder-based circuit we used for the control outputs.

Of course, not all these combinations are used, and although there are 16 instructions there are only 11 states. Also, some instructions use the same sequence of states, and most don't care about the condition flags. Nevertheless, the next state function design requires some thought.

As with the control outputs, a microcode contained in a programmable ROM could also provide the next state function. The current instruction, current state and condition flags could be used to form an address, and the output would be the next state. In fact, the control output and next state function could be combined into a single PROM, with an 11-bit address and 19-bit output (4 bits for the next state, 15 bits for the control outputs).

A more elegant way to encode the next state function is to use the technique of AND-OR arrayed logic. This type of logic is able to encode any function that maps an input to a unique output. With the inclusion of inverters, arrayed logic has the power to perform all the functions of the digital computer. In fact, programmable array logic IC's can be used to create entire microprocessors. We already saw an example of AND-OR array logic in the chapter about building the ALU.

In our case, we will first diagram the next state function as an AND-OR array, and then show how to build it. Keep in mind that the AND-OR array diagram is not a circuit diagram, but only a plan that will allow us to build the actual array.

The diagram consists of two parts. The upper part shows how the inputs are connected to multiple input AND gates. The lower part shows how the outputs of these AND gates are connected to multiple input OR gates. The OR gate outputs are the outputs of the circuit, the next state value.

The first step in creating the next state operation is to map it out in a table. This will show us what we need to do when it comes to building the circuit. We will list all possible combinations of current state, current operation and condition flags, and the

corresponding next state. First, we will make the table with decimal values for the states, using the mnemonics for the operations.

Current state	Operations	Condition	Next state
0	Don't care	Don't care	1
1	ADD, ADC, SUB, SBC, AND, OR, XOR	Don't care	2
1	NOT	Don't care	4
1	LDI	Don't care	5
1	LDM	Don't care	6
1	STM	Don't care	7
1	JMP	Don't care	9
1	Conditional jumps	Condition met	9
1	Conditional jumps	Condition not met	0
1	JPI	Don't care	10
2	ADD, ADC, SUB, SBC	Don't care	3
2	AND, OR, XOR	Don't care	4
3	Don't care	Don't care	0
4	Don't care	Don't care	0
5	Don't care	Don't care	0
6	Don't care	Don't care	0
7	Don't care	Don't care	8
8	Don't care	Don't care	0
9	Don't care	Don't care	0
10	Don't care	Don't care	0

The "don't care" entries show which states are always followed by the corresponding next state, regardless of the operation or condition.

The above table must be made into a circuit. In order to do this, we will need to look at the binary representations of the current state, operation and condition, and figure out how combine them to make a binary representation of the next state using AND-OR array logic.

It is convenient to think backward in making the AND-OR array. In other words, we look at each bit of the next state, and ask what kind of AND-OR operation is needed to get it from the corresponding current state, operation and condition as listed in the table above. Let us label the next state bits NS0 to NS3, (from low bit to high bit.) We will create operations that will cause the next state bit in question to become 1 when the input calls for it. We do not have to intentionally create an operation that will make the bit 0, because by default the bit will be 0 if it is not 1 (such is the elegance of the binary computer).

Let's start with NS0. It will be 1 when the next state is 1, 3, 5, 7, or 9. If we look at the table above, we can see that NS0 will be 1 when:

Current state	Operations	Condition
0	Don't care	Don't care
1	LDI	Don't care
1	STM	Don't care
1	JMP	Don't care
1	Conditional jumps	Condition met
2	ADD, ADC, SUB, SBC	Don't care

Now let's re-write this table using the binary equivalents, including the conditions.

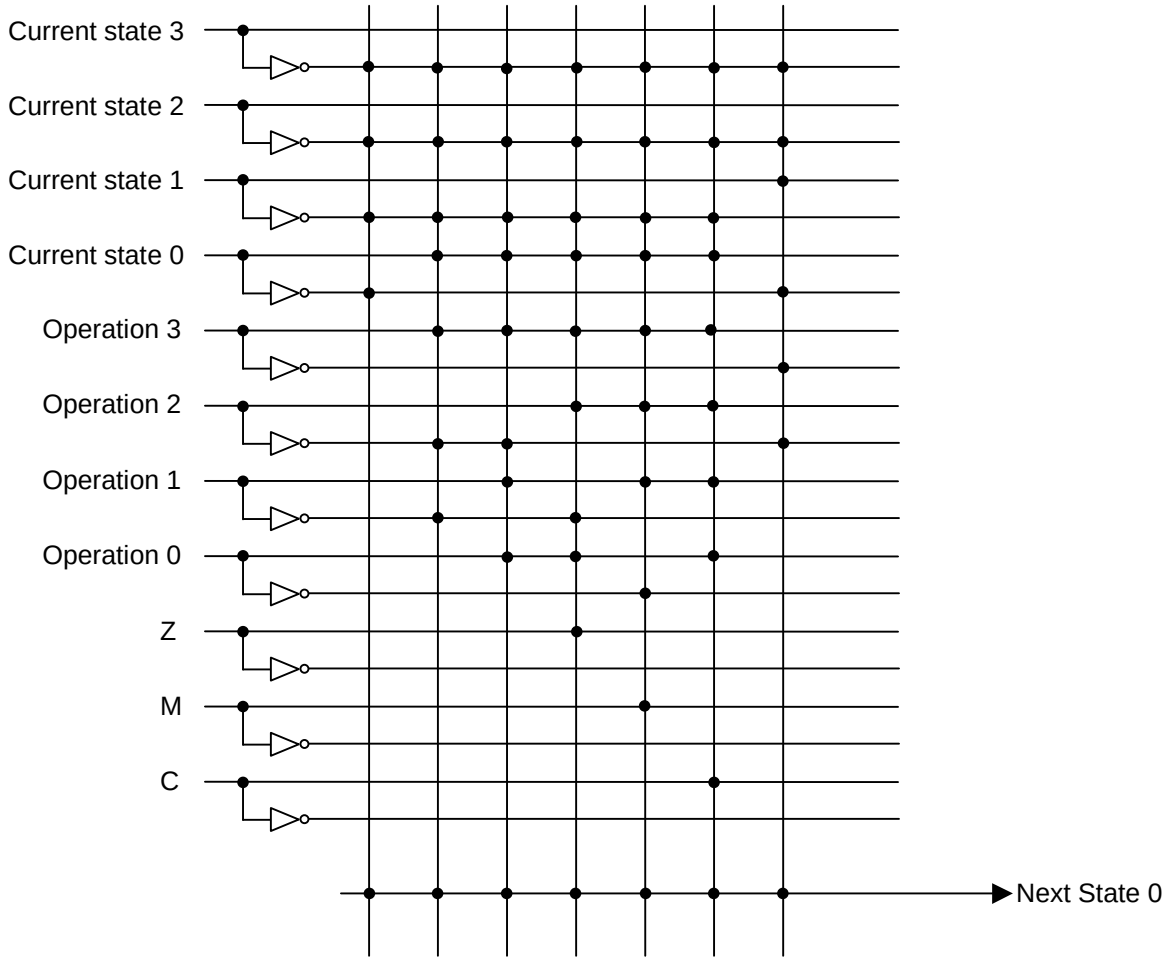
Current state	Operation	Z	M	C
0 0 0 0	X X X X	X	X	X
0 0 0 1	1 0 0 0	X	X	X
0 0 0 1	1 0 0 1	X	X	X
0 0 0 1	1 0 1 1	X	X	X
0 0 0 1	1 1 0 1	1	X	X
0 0 0 1	1 1 1 0	X	1	X
0 0 0 1	1 1 1 1	X	X	1
0 0 1 0	0 0 0 0	X	X	X
0 0 1 0	0 0 0 1	X	X	X
0 0 1 0	0 0 1 0	X	X	X
0 0 1 0	0 0 1 1	X	X	X

The X's stand for "don't care". This table can be simplified a little. Some of the bits can be replaced with "don't care" (X) if the two rows are otherwise identical, and the bit in question is a 1 in one row, and a 0 in the other. For example, rows 2 and 3 can be replaced by a single row in which the low order bit of the operation is a "don't care". Similarly, the last four rows can be combined into a single row with two don't cares in the lower 2 opcode bits. Here is the simplified table:

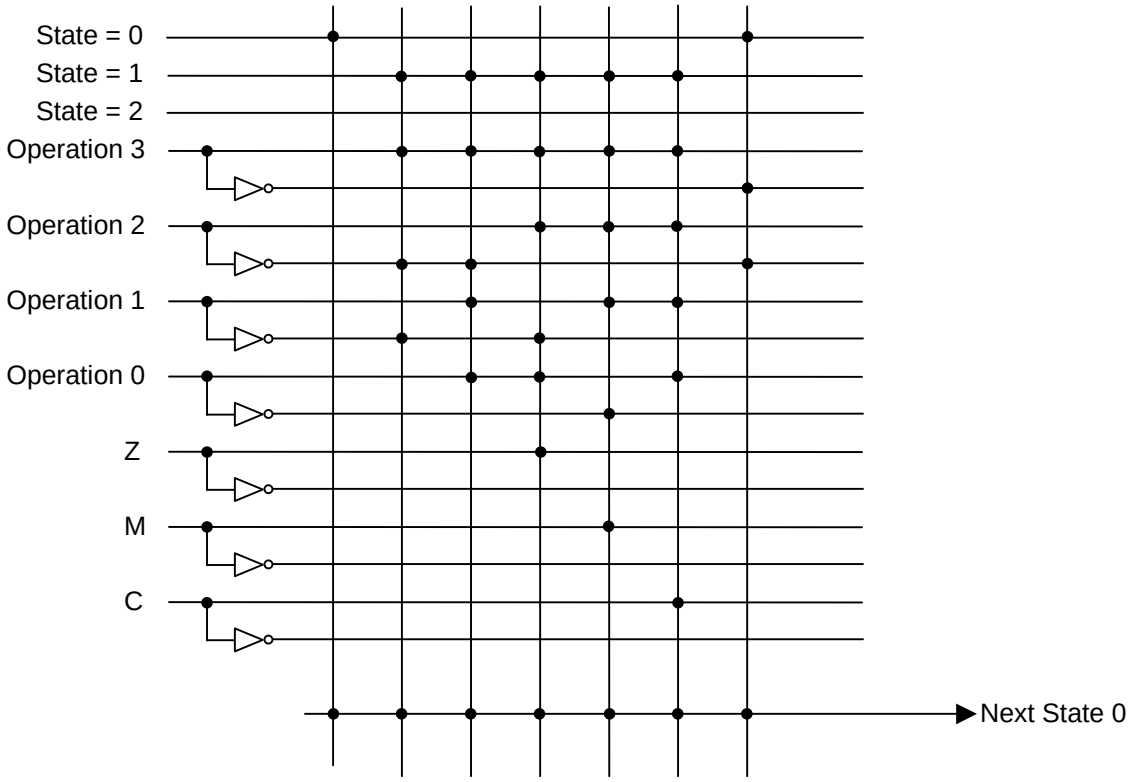
Current state	Operation	Z	M	C
0 0 0 0	X X X X	X	X	X
0 0 0 1	1 0 0 X	X	X	X
0 0 0 1	1 0 1 1	X	X	X
0 0 0 1	1 1 0 1	1	X	X
0 0 0 1	1 1 1 0	X	1	X
0 0 0 1	1 1 1 1	X	X	1
0 0 1 0	0 0 X X	X	X	X

This is the minimal table for the next state zero bit (NS0). Now, we are ready to draw the AND-OR array diagram for this table. All the 11 input bits (4 current state, 4 operation bits, and the 3 condition bits) are used in at least one row, so we will have 11 horizontal lines in the array diagram. We will include 11 inverted inputs to be used when an input bit is 0. There are 7 rows in the above table, each representing one multiple-input AND operation, so there will be 7 vertical lines in the array diagram. The 7 vertical lines will be connected in the lower part of the diagram by a horizontal line that represents the

ORing together of the AND operation outputs. The output of this OR operation is the NS0 bit.



The diagram looks complicated, but the circuit that it represents can be made with surprisingly few components. This is because several of the AND combinations (vertical lines) share patterns. For example, all 7 AND combinations use ((NOT Current State 3) AND (NOT Current State 2)). This can be handled by one two-input AND gate. Depending on how carefully you look for combinations to re-use, this circuit can be built with about 24 two-input AND gates, 8 inverters, and 6 two-input OR gates (the inverters on the condition lines are shown, but not used). This represents 10 14-pin IC's, which is not a difficult job. However, a clever reader may note that the AND patterns of the states and operations are the same as that for a decoder. If we use the outputs from the state decoder used previously in the control output circuit we can save some AND gates. Here is the same AND-OR array diagram using the decoded state lines. I only show the state lines actually used in the array; the others will have no connection in the next-state bit 0 array diagram.



This version uses only 18 AND gates, 4 inverters and 6 OR gates. Since the state decoder is already part of the logic circuitry, this is a reasonable way to proceed.

Here are the binary tables for the other next state output bits.

NS1: (when next state = 2,3,6,7,10)

Current state	Operation	Z	M	C
0 0 0 1	0 0 X X	X	X	X
0 0 0 1	0 1 0 X	X	X	X
0 0 1 0	0 0 X X	X	X	X
0 0 0 1	1 0 0 1	X	X	X
0 0 0 1	1 0 1 0	X	X	X
0 0 0 1	1 1 0 0	X	X	X

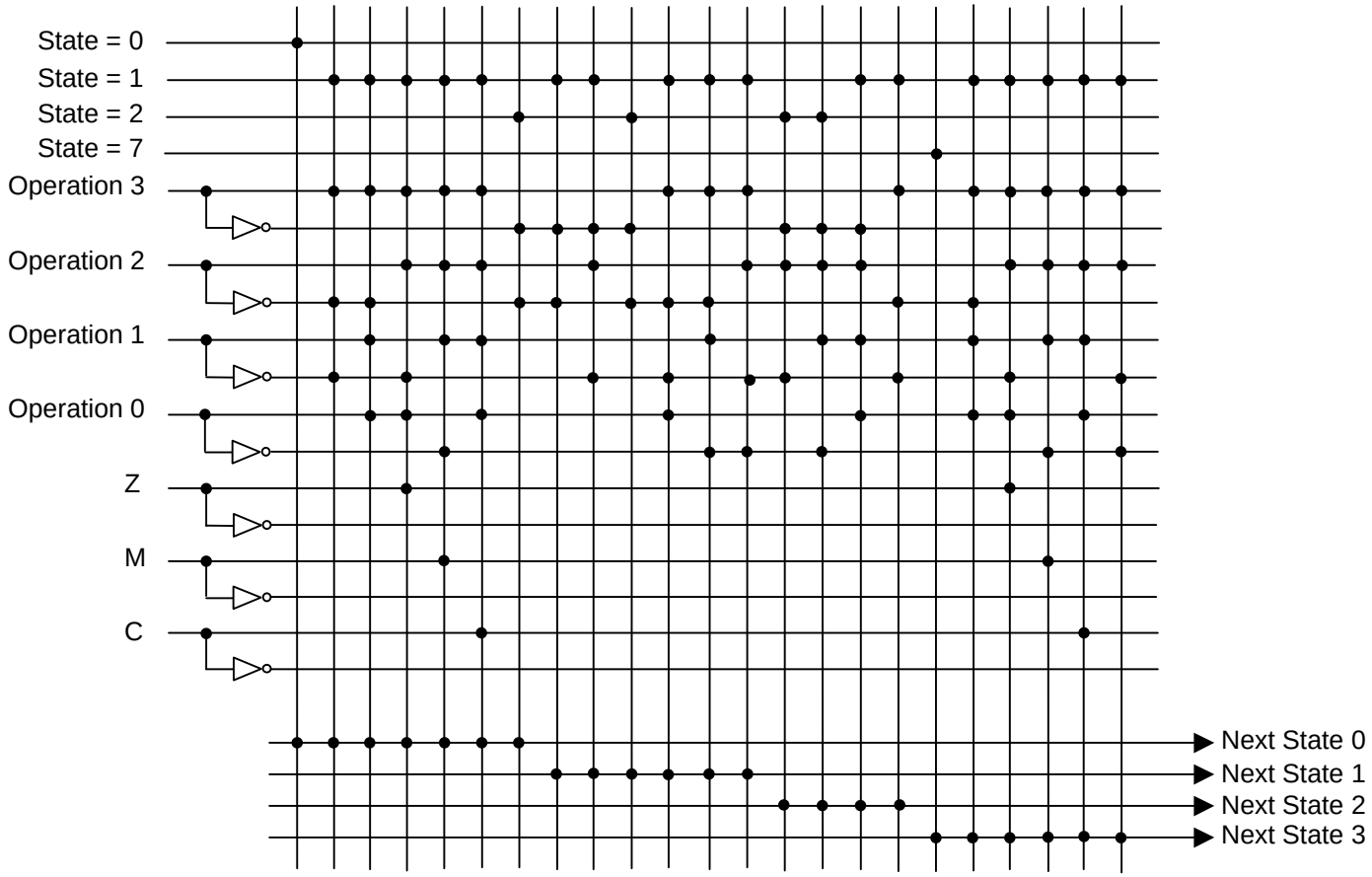
NS2: (when next state = 4,5,6,7)

Current state	Operation	Z	M	C
0 0 1 0	0 1 0 X	X	X	X
0 0 1 0	0 1 1 0	X	X	X
0 0 0 1	0 1 1 1	X	X	X
0 0 0 1	1 0 0 X	X	X	X

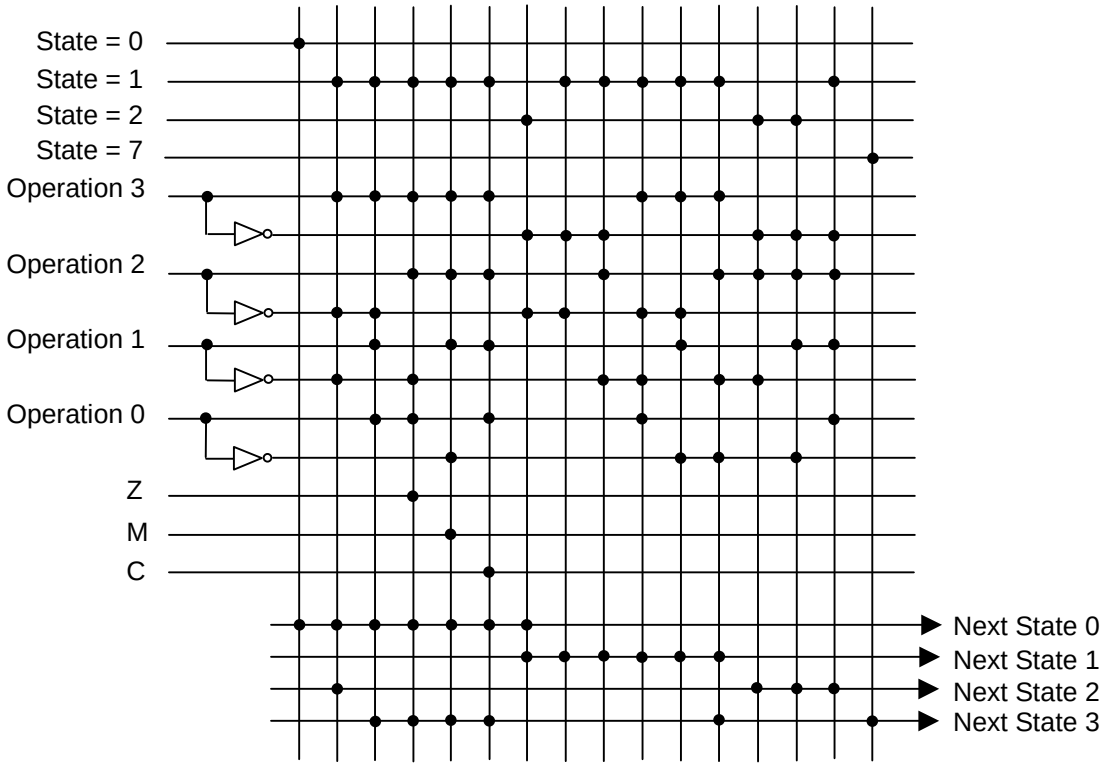
NS3 (when next state = 8,9,10)

Current state	Operation	Z	M	C
0 1 1 1	X X X X	X	X	X
0 0 0 1	1 0 1 1	X	X	X
0 0 0 1	1 1 0 1	1	X	X
0 0 0 1	1 1 1 0	X	1	X
0 0 0 1	1 1 1 1	X	X	1
0 0 0 1	1 1 0 0	X	X	X

Here is the AND-OR array diagram for the entire next-state operation.



The diagram suggests a lot of AND gates will be needed. But wait--more simplification is possible. First, we can eliminate the unused inverted condition flag lines. More importantly, we notice that some of the vertical lines have exactly the same patterns of dots. This means one or more can be eliminated, and the output of a single AND operation sent to the OR gates that received the output of the AND operations we eliminate. You can also see this if you look carefully at the next state bit tables. For example, the second row in the NS0 table is exactly the same as the fourth row in the NS2 table. We can use a single vertical line representing this particular AND operation, and connect the output to the OR gate inputs for the NS0 and NS2 bits. In all, there are 7 AND operations that can be shared by different NS output bits. Here is the final, simplified next state AND-OR array diagram.



The circuit described by this diagram can be built with 34 two-input AND gates, four inverters, and 20 2-input OR gates. These gates can be purchased in 14-pin IC's with 4 logic gates or 6 inverters each. Fewer IC's can be used if one purchases the available three or four input AND and OR gates. The entire control logic described in this chapter can be built of 15 next-state IC's, plus three IC's for the control logic, plus one 4-bit register IC, or 19 IC's total. The cost would be about $8.00 for these chips, but again you have to buy the board and sockets for a lot more money.