9.3 Implementing finite-state machine control

(Original section[1])

To implement the control as a finite-state machine, we must first assign a number to each of the 10 states; any state could use any number, but we will use the sequential numbering for simplicity. The figure below shows the finite-state diagram. With 10 states, we will need 4 bits to encode the state number, and we call these state bits S3, S2, S1, and S0. The current-state number will be stored in a state register, as shown in COD Figure C.3.2 (The control unit for MIPS will consist of ...). If the states are assigned sequentially, state *i* is encoded using the state bits as the binary number *i*. For example, state 6 is encoded as 0110_{two} or S3 = 0, S2 = 1, S1 = 1, S0 = 0, which can also be written as

$$\overline{S3} \cdot S2 \cdot S1 \cdot \overline{S0}$$

Figure 9.3.1: The finite-state diagram for multicycle control (COD Figure C.3.1).

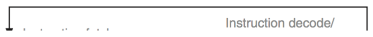
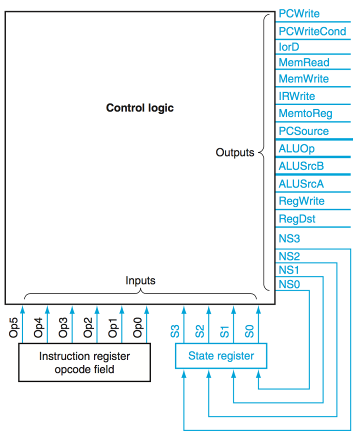


Figure 9.3.2: The control unit for MIPS will consist of some control logic and a register to hold the state (COD Figure C.3.2).

The state register is written at the active clock edge and is stable during the clock cycle.



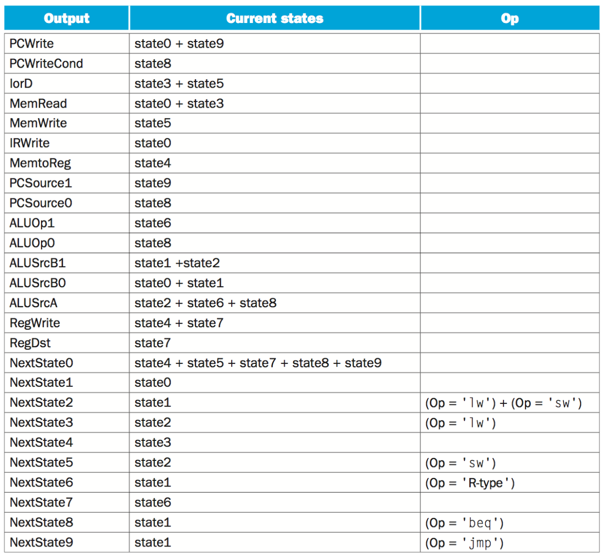
The control unit has outputs that specify the next state. These are written into the state register on the clock edge and become the new state at the beginning of the next clock cycle following the active clock edge. We name these outputs NS3, NS2, NS1, and NS0. Once we have determined the number of inputs, states, and outputs, we know what the basic outline of the control unit will look like, as we show in the figure above.

The block labeled "control logic" in the figure above is combinational logic. We can think of it as a big table giving the value of the outputs in terms of the inputs. The logic in this block implements the two different parts of the finite-state machine. One part is the logic that determines the setting of the datapath control outputs, which depend only on the state bits. The other part of the control logic implements the next-state function; these equations determine the values of the next-state bits based on the current-state bits and the other inputs (the 6-bit opcode).

The figure below shows the logic equations: the top portion shows the outputs, and the bottom portion shows the next-state function. The values in this table were determined from the state diagram in COD Figure C.3.1 (The finite-state diagram for multicycle control). Whenever a control line is active in a state, that state is entered in the second column of the table. Likewise, the next-state entries are made whenever one state is a successor to another.

Figure 9.3.3: The logic equations for the control unit shown in a shorthand form (COD Figure C.3.3).

Remember that "+" stands for OR in logic equations. The state inputs and NextState outputs must be expanded by using the state encoding. Any blank entry is a don't care.



Output	Current states	Op
PCWrite	state0 + state9	
PCWriteCond	state8	
IorD	state3 + state5	
MemRead	state0 + state3	
MemWrite	state5	
IRWrite	state0	
MemtoReg	state4	
PCSource1	state9	
PCSource0	state8	
ALUOp1	state6	
ALUOp0	state8	
ALUSrcB1	state1 +state2	
ALUSrcB0	state0 + state1	
ALUSrcA	state2 + state6 + state8	
RegWrite	state4 + state7	
RegDst	state7	
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

In the figure above, we use the abbreviation *stateN* to stand for current state *N*. Thus, *stateN* is replaced by the term that encodes the state number *N*. We use *NextStateN* to stand for the setting of the next-state outputs to *N*. This output is implemented using the next-state outputs (NS). When *NextStateN* is active, the bits NS[3—0] are set corresponding to the binary version of the value *N*. Of course, since a given next-state bit is activated in multiple next states, the equation for each state bit will be the OR of the terms that activate that signal. Likewise, when we use a term such as (Op = `'lw'`), this corresponds to an AND of the opcode inputs that specifies the encoding of the opcode `lw` in 6 bits, just as we did for the simple control unit in the previous section of this chapter. Translating the entries in the figure above into logic equations for the outputs is straightforward.

Example 9.3.1: Logic equations for next-state outputs.

Give the logic equation for the low-order next-state bit, NS0.

Answer

The next-state bit NS0 should be active whenever the next state has NS0 = 1 in the state encoding. This is true for NextState1, NextState3, NextState5, NextState7, and NextState9. The entries for these states in the figure above supply the conditions when these next-state values should be active. The equation for each of these next states is given below. The first equation states that the next state is 1 if the current state is 0; the current state is 0 if each of the state input bits is 0, which is what the rightmost product term indicates.

$$\mathrm{NextState1} = \mathrm{State0} = \overline{\mathrm{S3}} \cdot \overline{\mathrm{S2}} \cdot \overline{\mathrm{S1}} \cdot \overline{\mathrm{S0}}$$

$$\begin{aligned}\mathrm{NextState3} &= \mathrm{State2} \cdot (\mathrm{Op[5\text{-}0]} = \mathrm{1w}) \\ &= \overline{\mathrm{S3}} \cdot \overline{\mathrm{S2}} \cdot \mathrm{S1} \cdot \overline{\mathrm{S0}} \cdot \mathrm{Op5} \cdot \overline{\mathrm{Op4}} \cdot \overline{\mathrm{Op3}} \cdot \overline{\mathrm{Op2}} \cdot \mathrm{Op1} \cdot \mathrm{Op0}\end{aligned}$$

$$\begin{aligned}\mathrm{NextState5} &= \mathrm{State2} \cdot (\mathrm{Op[5\text{-}0]} = \mathrm{sw}) \\ &= \overline{\mathrm{S3}} \cdot \overline{\mathrm{S2}} \cdot \overline{\mathrm{S1}} \cdot \overline{\mathrm{S0}} \cdot \mathrm{Op5} \cdot \overline{\mathrm{Op4}} \cdot \mathrm{Op3} \cdot \overline{\mathrm{Op2}} \cdot \mathrm{Op1} \cdot \mathrm{Op0}\end{aligned}$$

$$\mathrm{NextState7} = \mathrm{State6} = \mathrm{S3} \cdot \mathrm{S2} \cdot \mathrm{S1} \cdot \mathrm{S0}$$

$$\begin{aligned}\mathrm{NextState9} &= \mathrm{State1} \cdot (\mathrm{Op[5\text{-}0]} = \mathrm{jmp}) \\ &= \overline{\mathrm{S3}} \cdot \overline{\mathrm{S2}} \cdot \overline{\mathrm{S1}} \cdot \mathrm{S0} \cdot \overline{\mathrm{Op5}} \cdot \overline{\mathrm{Op4}} \cdot \overline{\mathrm{Op3}} \cdot \overline{\mathrm{Op2}} \cdot \mathrm{Op1} \cdot \mathrm{Op0}\end{aligned}$$

NS0 is the logical sum of all these terms.

As we have seen, the control function can be expressed as a logic equation for each output. This set of logic equations can be implemented in two ways: corresponding to a complete truth table, or corresponding to a two-level logic structure that allows a sparse encoding of the truth table. Before we look at these implementations, let's look at the truth table for the complete control function.

It is simplest if we break the control function defined in the figure above into two parts: the next-state outputs, which may depend on all the inputs, and the control signal outputs, which depend only on the current-state bits. The figure below shows the truth tables for all the datapath control signals. Because these signals actually depend only on the state bits (and not the opcode), each of the entries in a table in the figure below actually represents 64 (= 2^6) entries, with the 6 bits named Op having all possible values; that is, the Op bits are don't-care bits in determining the data path control outputs. COD Figure C.3.5 (The four truth tables for the four next-state output ...) shows the truth table for the next-state bits NS[3—0], which depend on the state input bits and the instruction bits, which supply the opcode.

Figure 9.3.4: The truth tables are shown for the 16 datapath control signals that depend only on the current-state input bits, which are shown for each table (COD Figure C.3.4).

Each truth table row corresponds to 64 entries: one for each possible value of the six Op bits. Notice that some of the outputs are active under nearly the same circumstances. For example, in the case of PCWriteCond, PCSource0, and ALUOp0, these signals are active only in state 8 (see b, i, and k). These three signals could be replaced by one signal. There are other opportunities for reducing the logic needed to implement the control function by taking advantage of further similarities in the truth tables.

s3	s2	s1	s0
0	0	0	0
1	0	0	1

a. Truth table for PCWrite

s3	s2	s1	s0
1	0	0	0

b. Truth table for PCWriteCond

s3	s2	s1	s0
0	0	1	1
0	1	0	1

c. Truth table for IorD

s3	s2	s1	s0
0	0	0	0
0	0	1	1

d. Truth table for MemRead

s3	s2	s1	s0
0	1	0	1

e. Truth table for MemWrite

s3	s2	s1	s0
0	0	0	0

f. Truth table for IRWrite

s3	s2	s1	s0
0	1	0	0

g. Truth table for MemtoReg

s3	s2	s1	s0
1	0	0	1

h. Truth table for PCSource1

s3	s2	s1	s0
1	0	0	0

i. Truth table for PCSource0

s3	s2	s1	s0
0	1	1	0

j. Truth table for ALUOp1

s3	s2	s1	s0
1	0	0	0

k. Truth table for ALUOp0

s3	s2	s1	s0
0	0	0	1
0	0	1	0

l. Truth table for ALUSrcB1

s3	s2	s1	s0
0	0	0	0
0	0	0	1

m. Truth table for ALUSrcB0

s3	s2	s1	s0
0	0	1	0
0	1	1	0
1	0	0	0

n. Truth table for ALUSrcA

s3	s2	s1	s0
0	1	0	0
0	1	1	1

o. Truth table for RegWrite

s3	s2	s1	s0
0	1	1	1

p. Truth table for RegDst

Figure 9.3.5: The four truth tables for the four next-state output bits (NS[3-0]) (COD Figure C.3.5).

The next-state outputs depend on the value of Op[5—0], which is the opcode field, and the current state, given by S[3—0]. The entries with X are don't-care terms. Each entry with a don't-care term corresponds to two entries, one with that input at 0 and one with that input at 1. Thus an entry with n don't-care terms actually corresponds to 2^n truth table entries.

Op5	Op4	Op3	Op2	Op1	Op0	S3	S2	S1	S0

Elaboration

There are many opportunities to simplify the control function by observing similarities among two or more control signals and by using the semantics of the implementation. For example, *the signals PCWriteCond, PCSource0, and ALUOp0 are all asserted in exactly one state, state 8. These three control signals can be replaced by a single signal.*

A ROM implementation

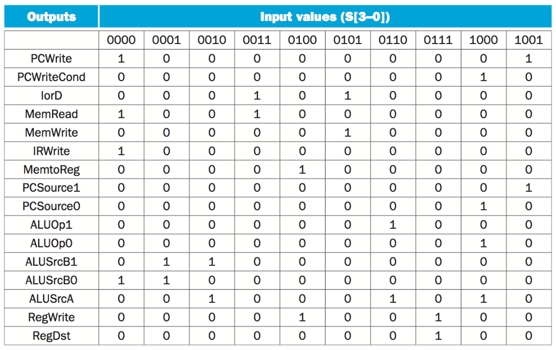
Probably the simplest way to implement the control function is to encode the truth tables in a read-only memory (ROM). The number of entries in the memory for the truth tables of COD Figures C.3.4 (The truth tables are shown for the 16 datapath control signals ...) and C.3.5 (The four truth tables for the four next-state output ...) is equal to all possible values of the inputs (the 6 opcode bits plus the 4 state bits), which is $2^{\#\text{ inputs}} = 2^{10} = 1024$. The inputs to the control unit become the address lines for the ROM, which implements the control logic block that was shown in COD Figure C.3.2 (The control unit for MIPS ...). The width of each entry (or word in the memory) is 20 bits, since there are 16 datapath control outputs and 4 next-state bits. This means the total size of the ROM is $2^{10} \times 20 = 20$ Kbits.

The setting of the bits in a word in the ROM depends on which outputs are active in that word. Before we look at the control words, we need to order the bits within the control input (the address) and output words (the contents), respectively. We will number the bits using the order in COD Figure C.3.2 (The control unit for MIPS ...), with the next-state bits being the low-order bits of the control *word* and the current-state input bits being the low-order bits of the *address*. This means that the PCWrite output will be the high-order bit (bit 19) of each memory word, and NS0 will be the low-order bit. The high-order address bit will be given by Op5, which is the high-order bit of the instruction, and the low-order address bit will be given by S0.

We can construct the ROM contents by building the entire truth table in a form where each row corresponds to one of the 2^n unique input combinations, and a set of columns indicates which outputs are active for that input combination. We don't have the space here to show all 1024 entries in the truth table. However, by separating the datapath control and next-state outputs, we do, since the datapath control outputs depend only on the current state. The truth table for the datapath control outputs is shown in the figure below. We include only the encodings of the state inputs that are in use (that is, values 0 through 9 corresponding to the 10 states of the state machine).

Figure 9.3.6: The truth table for the 16 datapath control outputs, which depend only on the state inputs (COD Figure C.3.6).

The values are determined from COD Figure C.3.4 (The truth tables are shown for the 16 datapath control signals ...). Although there are 16 possible values for the 4-bit state field, only 10 of these are used and are shown here. The 10 possible values are shown at the top; each column shows the setting of the datapath control outputs for the state input value that appears at the top of the column. For example, when the state inputs are 0011 (state 3), the active datapath control outputs are IorD or MemRead.



Outputs	Input values (S[3–0])									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	0	0	0	0	1	0	0	0	0	0
PCSource1	0	0	0	0	0	0	0	0	0	1
PCSource0	0	0	0	0	0	0	0	0	1	0
ALUOp1	0	0	0	0	0	0	1	0	0	0
ALUOp0	0	0	0	0	0	0	0	0	1	0
ALUSrcB1	0	1	1	0	0	0	0	0	0	0
ALUSrcB0	1	1	0	0	0	0	0	0	0	0
ALUSrcA	0	0	1	0	0	0	1	0	1	0
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	0	0	0	0	0	0	0	1	0	0

The truth table in the figure above directly gives the contents of the upper 16 bits of each word in the ROM. The 4-bit input field gives the low-order 4 address bits of each word, and the column gives the contents of the word at that address.

If we did show a full truth table for the datapath control bits with both the state number and the opcode bits as inputs, the opcode inputs would all be don't cares. When we construct the ROM, we cannot have any don't cares, since the addresses into the ROM must be complete. Thus, the same datapath control outputs will occur many times in the ROM, since this part of the ROM is the same whenever the state bits are identical, independent of the value of the opcode inputs.

Example 9.3.2: Control ROM entries.

For what ROM addresses will the bit corresponding to PCWrite, the high bit of the control word, be 1?

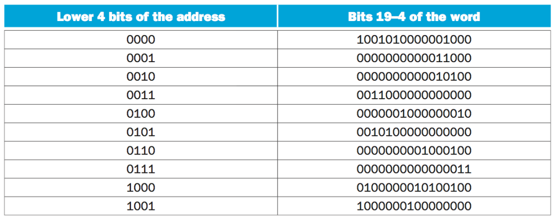
Answer

PCWrite is high in states 0 and 9; this corresponds to addresses with the 4 low-order bits being either 0000 or 1001. The bit will be high in the memory word independent of the inputs Op[5—0], so the addresses with the bit high are 000000000, 0000001001, 0000010000, 0000011001, ..., 1111110000, 1111111001. The general form of this is XXXXXX0000 or XXXXXX1001, where XXXXXX is any combination of bits, and corresponds to the 6-bit opcode on which this output does not depend.

We will show the entire contents of the ROM in two parts to make it easier to show. The figure below shows the upper 16 bits of the control word; this comes directly from the figure above. These datapath control outputs depend only on the state inputs, and this set of words would be duplicated 64 times in the full ROM, as we discussed above. The entries corresponding to input values 1010 through 1111 are not used, so we do not care what they contain.

Figure 9.3.7: The contents of the upper 16 bits of the ROM depend only on the state inputs (COD Figure C.3.7).

These values are the same as those in the figure above, simply rotated 90°. This set of control words would be duplicated 64 times for every possible value of the upper six bits of the address.

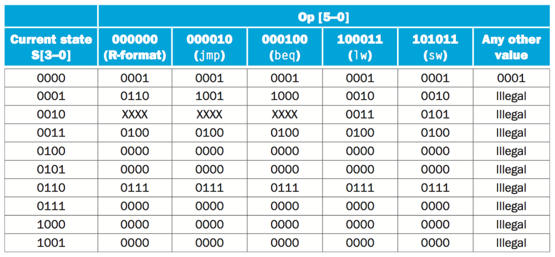


Lower 4 bits of the address	Bits 19–4 of the word
0000	1001010000001000
0001	0000000000011000
0010	0000000000010100
0011	0011000000000000
0100	0000001000000010
0101	0010100000000000
0110	0000000001000100
0111	0000000000000011
1000	0100000010100100
1001	1000000100000000

The figure below shows the lower four bits of the control word corresponding to the next-state outputs. The last column of the table in the figure below corresponds to all the possible values of the opcode that do not match the specified opcodes. In state 0, the next state is always state 1, since the instruction was still being fetched. After state 1, the opcode field must be valid. The table indicates this by the entries marked illegal; we discuss how to deal with these exceptions and interrupts opcodes in COD Section 4.9 (Exceptions).

Figure 9.3.8: This table contains the lower 4 bits of the control word (the NS outputs), which depend on both the state inputs, S[3—0], and the opcode, Op[5—0], which correspond to the instruction opcode (COD Figure C.3.8).

These values can be determined from COD Figure C.3.5 (The four truth tables for the four next-state output bits ...). The opcode name is shown under the encoding in the heading. The four bits of the control word whose address is given by the current-state bits and Op bits are shown in each entry. For example, when the state input bits are 0000, the output is always 0001, independent of the other inputs; when the state is two, the next state is don't care for three of the inputs, three for `lw`, and five for `sw`. Together with the entries in the figure above, this table specifies the contents of the control unit ROM. For example, the word at address 1000110001 is obtained by finding the upper 16 bits in the table in the figure above using only the state input bits (0001) and concatenating the lower four bits found by using the entire address (0001 to find the row and 100011 to find the column). The entry from the figure above yields 0000000000011000, while the appropriate entry in the table immediately above is 0010. Thus the control word at address 1000110001 is 00000000000110000010. The column labeled "Any other value" applies only when the Op bits do not match one of the specified opcodes.



	Op [5–0]					
Current state S[3–0]	000000 (R-format)	000010 (`jmp`)	000100 (`beq`)	100011 (`lw`)	101011 (`sw`)	Any other value
0000	0001	0001	0001	0001	0001	0001
0001	0110	1001	1000	0010	0010	Illegal
0010	XXXX	XXXX	XXXX	0011	0101	Illegal
0011	0100	0100	0100	0100	0100	Illegal
0100	0000	0000	0000	0000	0000	Illegal
0101	0000	0000	0000	0000	0000	Illegal
0110	0111	0111	0111	0111	0111	Illegal
0111	0000	0000	0000	0000	0000	Illegal
1000	0000	0000	0000	0000	0000	Illegal
1001	0000	0000	0000	0000	0000	Illegal

Not only is this representation as two separate tables a more compact way to show the ROM contents; it is also a more efficient way to implement the ROM. The majority of the outputs (16 of 20 bits) depends only on four of the 10 inputs. The number of bits in total when the control is implemented as two separate ROMs is $2^4 \times 16 + 2^{10} \times 4 = 256 + 4096 = 4.3$ Kbits, which is about one-fifth of the size of a single ROM, which requires $2^{10} \times 20 = 20$ Kbits. There is some overhead associated with any structured-logic block, but in this case the additional overhead of an extra ROM would be much smaller than the savings from splitting the single ROM.

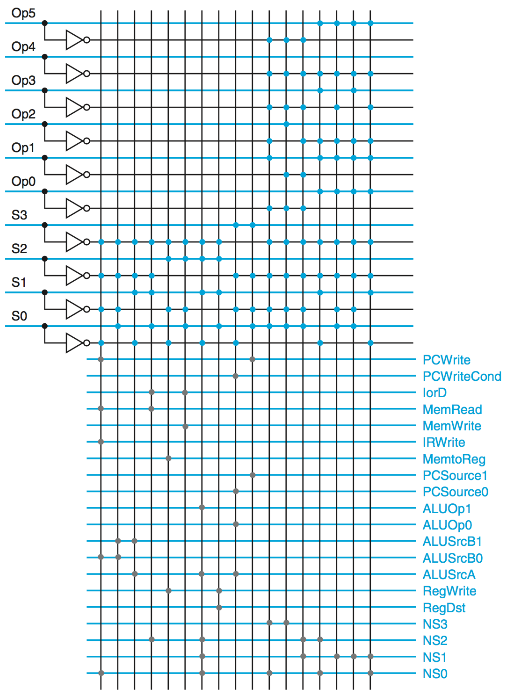
Although this ROM encoding of the control function is simple, it is wasteful, even when divided into two pieces. For example, the values of the Instruction register inputs are often not needed to determine the next state. Thus, the nextstate ROM has many entries that are either duplicated or are don't care. Consider the case when the machine is in state 0: there are 2^6 entries in the ROM (since the opcode field can have any value), and these entries will all have the same contents (namely, the control word 0001). The reason that so much of the ROM is wasted is that the ROM implements the complete truth table, providing the opportunity to have a different output for every combination of the inputs. But most combinations of the inputs either never happen or are redundant!

A PLA Implementation

We can reduce the amount of control storage required at the cost of using more complex address decoding for the control inputs, which will encode only the input combinations that are needed. The logic structure most often used to do this is a programmed logic array (PLA), which we mentioned earlier and illustrated in COD Figure C.2.5 (The structured implementation of the control function ...). In a PLA, each output is the logical OR of one or more minterms. A *minterm*, also called a *product term*, is simply a logical AND of one or more inputs. The inputs can be thought of as the address for indexing the PLA, while the minterms select which of all possible address combinations are interesting. A minterm corresponds to a single entry in a truth table, such as those in COD Figure C.3.4 (The truth tables are shown for the 16 datapath control signals ...), including possible don't-care terms. Each output consists of an OR of these minterms, which exactly corresponds to a complete truth table. However, unlike a ROM, only those truth table entries that produce an active output are needed, and only one copy of each minterm is required, even if the minterm contains don't cares. The figure below shows the PLA that implements this control function.

Figure 9.3.9: This PLA implements the control function logic for the multicycle implementation (COD Figure C.3.9).

The inputs to the control appear on the left and the outputs on the right. The top half of the figure is the AND plane that computes all the minterms. The minterms are carried to the OR plane on the vertical lines. Each colored dot corresponds to a signal that makes up the minterm carried on that line. The sum terms are computed from these minterms, with each gray dot representing the presence of the intersecting minterm in that sum term. Each output consists of a single sum term.



As we can see from the PLA in the figure above, there are 17 unique minterms—10 that depend only on the current state and seven others that depend on a combination of the Op field and the current-state bits. The total size of the PLA is proportional to (#inputs × #product terms) + (#outputs × #product terms), as we can see symbolically from the figure. This means the total size of the PLA in the figure above is proportional to (10 × 17) + (20 × 17) = 510. By comparison, the size of a single ROM is proportional to 20 Kb, and even the two-part ROM has a total of 4.3 Kb. Because the size of a PLA cell will be only slightly larger than the size of a bit in a ROM, a PLA will be a much more efficient implementation for this control unit.

Of course, just as we split the ROM in two, we could split the PLA into two PLAs: one with four inputs and 10 minterms that generates the 16 control outputs, and one with 10 inputs and seven minterms that generates the four next-state outputs. The first PLA would have a size proportional to (4 × 10) + (10 × 16) = 200, and the second PLA would have a size proportional to (10 × 7) + (4 × 7) = 98. This would yield a total size proportional to 298 PLA cells, about 55% of the size of a single PLA. These two PLAs will be considerably smaller than an implementation using two ROMs. For more details on PLAs and their implementation, as well as the references for books on logic design, see COD Appendix A (The Basics of Logic Design).

(*1) This section is in original form.

Provide feedback on this section