9.4 Implementing the next-state function with a sequencer

(Original section[1])

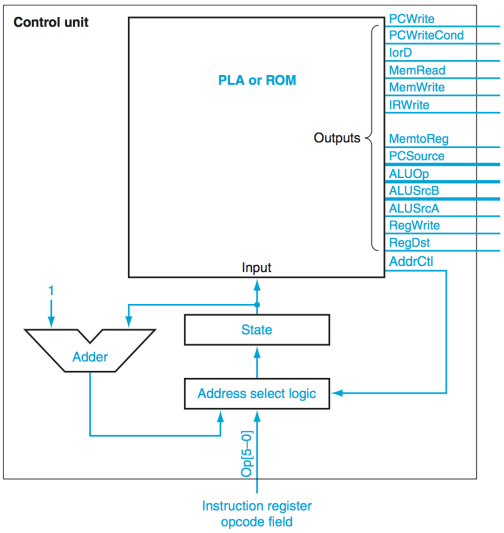
Let's look carefully at the control unit we built in the last section. If you examine the ROMs that implement the control in COD Figures C.3.7 (The contents of the upper 16 bits of the ROM depend only on the state inputs) and C.3.8 (This table contains the lower 4 bits of the control word ...), you can see that much of the logic is used to specify the next-state function. In fact, for the implementation using two separate ROMs, 4096 out of the 4368 bits (94%) correspond to the next-state function! Furthermore, imagine what the control logic would look like if the instruction set had many more different instruction types, some of which required many clocks to implement. There would be many more states in the finite-state machine. In some states, we might be branching to a large number of different states depending on the instruction type (as we did in state 1 of the finite-state machine in COD Figure C.3.1 (The finite-state diagram for multicycle control)). However, many of the states would proceed in a sequential fashion, just as states 3 and 4 do in COD Figure C.3.1 (The finite-state diagram for multicycle control).

For example, if we included floating point, we would see a sequence of many states in a row that implement a multicycle floating-point instruction. Alternatively, consider how the control might look for a machine that can have multiple memory operands per instruction. It would require many more states to fetch multiple memory operands. The result of this would be that the control logic will be dominated by the encoding of the next-state function. Furthermore, much of the logic will be devoted to sequences of states with only one path through them that look like states 2 through 4 in COD Figure C.3.1 (The finite-state diagram for multicycle control). With more instructions, these sequences will consist of many more sequentially numbered states than for our simple subset.

To encode these more complex control functions efficiently, we can use a control unit that has a counter to supply the sequential next state. This counter often eliminates the need to encode the next-state function explicitly in the control unit. As shown in the figure below, an adder is used to increment the state, essentially turning it into a counter. The incremented state is always the state that follows in numerical order. However, the finite-state machine sometimes "branches." For example, in state 1 of the finite-state machine (see COD Figure C.3.1 (The finite-state diagram for multicycle control)), there are four possible next states, only one of which is the sequential next state. Thus, we need to be able to choose between the incremented state and a new state based on the inputs from the Instruction register and the current state. Each control word will include control lines that will determine how the next state is chosen.

Figure 9.4.1: The control unit using an explicit counter to compute the next state (COD Figure C.4.1).

In this control unit, the next state is computed using a counter (at least in some states). By comparison, COD Figure C.3.2 (The control unit for MIPS ...) encodes the next state in the control logic for every state. In this control unit, the signals labeled *AddrCtl* control how the next state is determined.



It is easy to implement the control output signal portion of the control word, since, if we use the same state numbers, this portion of the control word will look exactly like the ROM contents shown in COD Figure C.3.7 (The contents of the upper 16 bits of the ROM depend only on the state inputs). However, the method for selecting the next state differs from the next-state function in the finite-state machine.

With an explicit counter providing the sequential next state, the control unit logic need only specify how to choose the state when it is not the sequentially following state. There are two methods for doing this. The first is a method we have already seen: namely, the control unit explicitly encodes the next-state function. The difference is that the control unit need only set the next-state lines when the designated next state is not the state that the counter indicates. If the number of states is large and the next-state function that we need to encode is mostly empty, this may not be a good choice, since the resulting control unit will have lots of empty or redundant space. An alternative approach is to use separate external logic to specify the next state when the counter does not specify the state. Many control units, especially those that implement large instruction sets, use this approach, and we will focus on specifying the control externally.

Although the nonsequential next state will come from an external table, the control unit needs to specify when this should occur and how to find that next state. There are two kinds of "branching" that we must implement in the address select logic. First, we must be able to jump to one of a number of states based on the opcode portion of the Instruction register. This operation, called a *dispatch*, is usually implemented by using a set of special ROMs or PLAs included as part of the address selection logic. An additional set of control outputs, which we call AddrCtl, indicates when a dispatch should be done. Looking at the finite-state diagram (COD Figure C.3.1 (The finite-state diagram for multicycle control)), we see that there are two states in which we do a branch based on a portion of the opcode. Thus we will need two small dispatch tables. (Alternatively, we could also use a single dispatch table and use the control bits that select the table as address bits that choose from which portion of the dispatch table to select the address.)

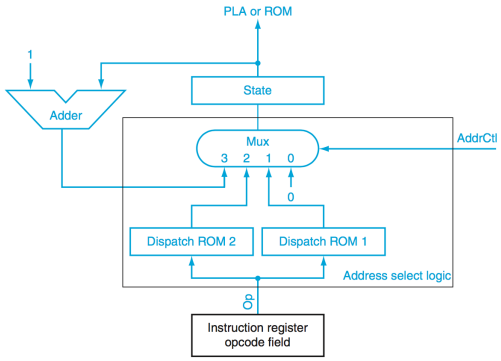
The second type of branching that we must implement consists of branching back to state 0, which initiates the execution of the next MIPS instruction. Thus there are four possible ways to choose the next state (three types of branches, plus incrementing the current-state number), which can be encoded in 2 bits. Let's assume that the encoding is as follows:

AddrCtl value	Action

0	Set state to 0
1	Dispatch with ROM 1
2	Dispatch with ROM 2
3	Use the incremented state

If we use this encoding, the address select logic for this control unit can be implemented as shown in the figure below.

Figure 9.4.2: This is the address select logic for the control unit of the figure above (COD Figure C.4.2).



To complete the control unit, we need only specify the contents of the dispatch ROMs and the values of the address-control lines for each state. We have already specified the datapath control portion of the control word using the ROM contents of COD Figure C.3.7 (The contents of the upper 16 bits of the ROM depend only on the state inputs) (or the corresponding portions of the PLA in COD Figure C.3.9 (This PLA implements the control function logic for the multicycle implementation)). The next-state counter and dispatch ROMs take the place of the portion of the control unit that was computing the next state, which was shown in COD Figure C.3.8 (This table contains the lower 4 bits of the control word ...). We are only implementing a portion of the instruction set, so the dispatch ROMs will be largely empty. The figure below shows the entries that must be assigned for this subset.

Figure 9.4.3: The dispatch ROMs each have 2^6 = 64 entries that are 4 bits wide, since that is the number of bits in the state encoding (COD Figure C.4.3).

This figure only shows the entries in the ROM that are of interest for this subset. The first column in each table indicates the value of Op, which is the address used to access the dispatch ROM. The second column shows the symbolic name of the opcode. The third column indicates the value at that address in the ROM.

Dispatch ROM 1		
Op	**Opcode name**	**Value**
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	**Opcode name**	**Value**
100011	lw	0011
101011	sw	0101

Now we can determine the setting of the address selection lines (AddrCtl) in each control word. The table in the figure below shows how the address control must be set for every state. This information will be used to specify the setting of the AddrCtl field in the control word associated with that state.

Figure 9.4.4: The values of the address-control lines are set in the control word that corresponds to each state (COD Figure C.4.4).

State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

The contents of the entire control ROM are shown in the figure below. The total storage required for the control is quite small. There are 10 control words, each 18 bits wide, for a total of 180 bits. In addition, the two dispatch tables are 4 bits wide and each has 64 entries, for a total of 512 additional bits. This total of 692 bits beats the implementation that uses two ROMs with the next-state function encoded in the ROMs (which requires 4.3 Kbits).

Figure 9.4.5: The contents of the control memory for an implementation using an explicit counter (COD Figure C.4.5).

The first column shows the state, while the second shows the datapath control bits, and the last column shows the address-control bits in each control word. Bits 17—2 are identical to those in COD Figure C.3.7 (The contents of the upper 16 bits of the ROM depend only on the state inputs).

State number	Control word bits 17–2	Control word bits 1–0
0	1001010000001000	11
1	0000000000011000	01
2	0000000000010100	10
3	0011000000000000	11
4	0000001000000010	00
5	0010100000000000	00
6	0000000001000100	11
7	0000000000000011	00
8	0100000010100100	00
9	1000000100000000	00

Of course, the dispatch tables are sparse and could be more efficiently implemented with two small PLAs. The control ROM could also be replaced with a PLA.

Optimizing the control implementation

We can further reduce the amount of logic in the control unit by two different techniques. The first is *logic minimization*, which uses the structure of the logic equations, including the don't-care terms, to reduce the amount of hardware required. The success of this process depends on how many entries exist in the truth table, and how those entries are related. For example, in this subset, only the `lw` and `sw` opcodes have an active value for the signal Op5, so we can replace the two truth table entries that test whether the input is `lw` or `sw` by a single test on this bit; similarly, we can eliminate several bits used to index the dispatch ROM because this single bit can be used to find `lw` and `sw` in the first dispatch ROM. Of course, if the opcode space were less sparse, opportunities for this optimization would be more difficult to locate. However, in choosing the opcodes, the architect can provide additional opportunities by choosing related opcodes for instructions that are likely to share states in the control.

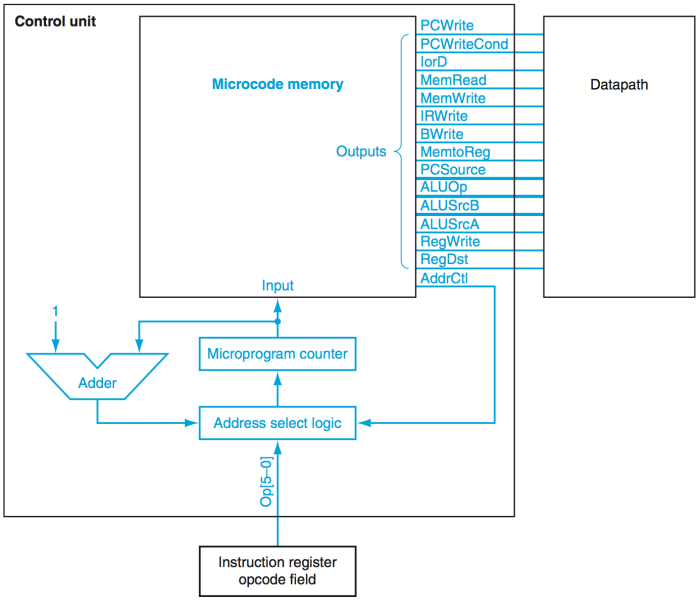
A different sort of optimization can be done by assigning the state numbers in a finite-state or microcode implementation to minimize the logic. This optimization, called *state assignment*, tries to choose the state numbers such that the resulting logic equations contain more redundancy and can thus be simplified. Let's consider the case of a finite-state machine with an encoded next-state control first, since it allows states to be assigned arbitrarily. For example, notice that in the finite-state machine, the signal RegWrite is active only in states 4 and 7. If we encoded those states as 8 and 9, rather than 4 and 7, we could rewrite the equation for RegWrite as simply a test on bit S3 (which is only on for states 8 and 9). This renumbering allows us to combine the two truth table entries in part (o) of COD Figure C.3.4 (The truth tables are shown for the 16 datapath control signals …) and replace them with a single entry, eliminating one term in the control unit. Of course, we would have to renumber the existing states 8 and 9, perhaps as 4 and 7.

The same optimization can be applied in an implementation that uses an explicit program counter, though we are more restricted. Because the next-state number is often computed by incrementing the current-state number, we cannot arbitrarily assign the states. However, if we keep the states where the incremented state is used as the next state in the same order, we can reassign the consecutive states as a block. In an implementation with an explicit next-state counter, state assignment may allow us to simplify the contents of the dispatch ROMs.

If we look again at the control unit in COD Figure C.4.1 (The control unit using an explicit counter to compute the next state), it looks remarkably like a computer in its own right. The ROM or PLA can be thought of as memory supplying instructions for the datapath. The state can be thought of as an instruction address. Hence the origin of the name *microcode* or *microprogrammed control*. The control words are thought of as *microinstructions* that control the datapath, and the State register is called the *microprogram counter*. The figure below shows a view of the control unit as microcode. The next section describes how we map from a microprogram to microcode.

Figure 9.4.6: The control unit as a microcode (COD Figure C.4.6).

The use of the word "micro" serves to distinguish between the program counter in the datapath and the microprogram counter, and between the microcode memory and the instruction memory.



(*1) This section is in original form.

Provide feedback on this section