# Microcode

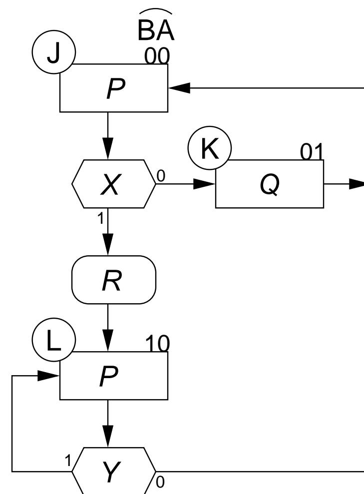
Like any boolean function, and ASM's *next-state* function can be implemented as a table look-up, realized with a memory device. This is standard technique with several advantages.

Several optimization techniques apply. Some of these are illustrated as we refine a simple implementation scheme, below. Since memories grow exponentially with the number of address bits and proportionately with the number of data bits, the primary goal is to reduce the number of address bits.

## Illustration

### Initial Implementation

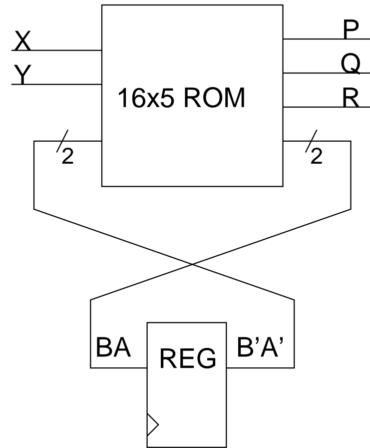
Suppose we begin with the ASM shown below.




A direct, table-lookup implementation of the ASM introduces registers to hold the current control state. It requires a $16 \times 5$ ROM. A word in this memory has two fields, one containing the *next-state* function value and the other generating the *command* bits from the ASM. There must be one address bit for each bit of the encoded state (it would make little sense to use a one-hot state encoding)

Copyright © 2003 Steven D. Johnson 

and one for each *status* bit.

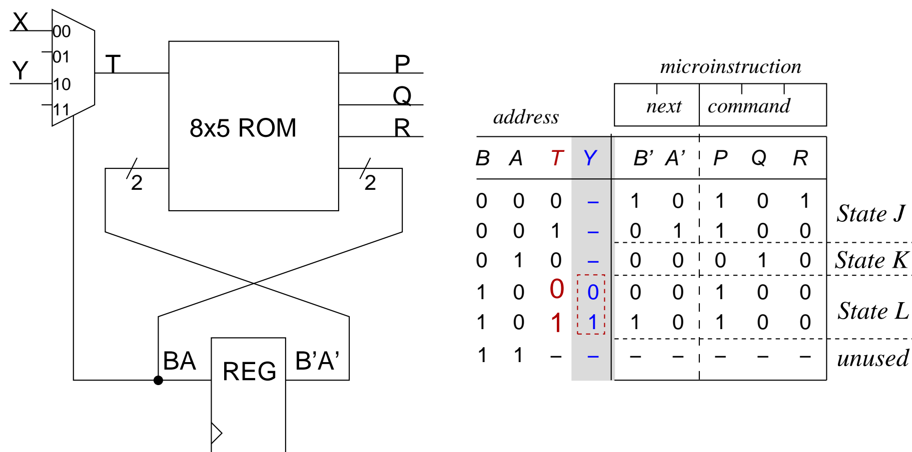



| *address* | | | | *microinstruction: next* | | *microinstruction: command* | | | |
|---|---|---|---|---|---|---|---|---|---|
| *B* | *A* | *X* | *Y* | *B'* | *A'* | *P* | *Q* | *R* | |
| 0 | 0 | 0 | – | 1 | 0 | 1 | 0 | 1 | *State J* |
| 0 | 0 | 1 | – | 0 | 1 | 1 | 0 | 0 | |
| 0 | 1 | – | – | 0 | 0 | 0 | 1 | 0 | *State K* |
| 1 | 0 | – | 0 | 0 | 0 | 1 | 0 | 0 | *State L* |
| 1 | 0 | – | 1 | 1 | 0 | 1 | 0 | 0 | |
| 1 | 1 | – | – | – | – | – | – | – | *unused* |

One-fourth of the memory is unused since there are only three states.

### Encoded tests

With a bit of external hardware, we can reduce the memory size by half (in this case). A selector is added to choose which of the test inputs, $X$ or $Y$, is in effect for a given state. This refinement is valid because the ASM has at most one decision block in each state. A similar optimization would apply in more general cases by more selectors, as dictated by the most complex branch in the ASM.



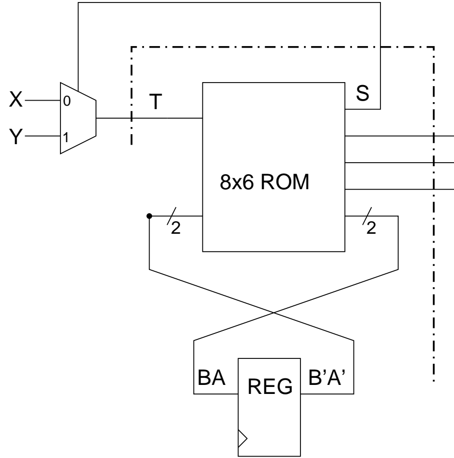

| address | | | | microinstruction: next | | command | | | |
|---|---|---|---|---|---|---|---|---|---|
| B | A | T | Y | B' | A' | P | Q | R | |
| 0 | 0 | 0 | – | 1 | 0 | 1 | 0 | 1 | State J |
| 0 | 0 | 1 | – | 0 | 1 | 1 | 0 | 0 | |
| 0 | 1 | 0 | – | 0 | 0 | 0 | 1 | 0 | State K |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | State L |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | |
| 1 | 1 | – | – | – | – | – | – | – | unused |

An alternative is to restrict our ASMs to have at most have at most one decision branch per state. The choice depends on several factors, and whether the goal is to do custom design or develop supporting tools.

Copyright © 2003 Steven D. Johnson 

### Test selection

Instead of using the current-state value (BA) for selecting a condition to test, we gain some generality by adding a *test selection* bit to the command word. At the expense of a slightly larger ROM, we get a more orthogonal separation of control flow—the business of the controller—and architcture status.




| *address* | | | *ts* | *next* | | *command* | | | |
|---|---|---|---|---|---|---|---|---|---|
| *B* | *A* | *T* | *S* | *B'* | *A'* | *P* | *Q* | *R* | |
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | *State J* |
| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | |
| 0 | 1 | – | 0 | 0 | 0 | 0 | 1 | 0 | *State K* |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | *State L* |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | |
| 1 | 1 | – | – | – | – | – | – | – | *unused* |

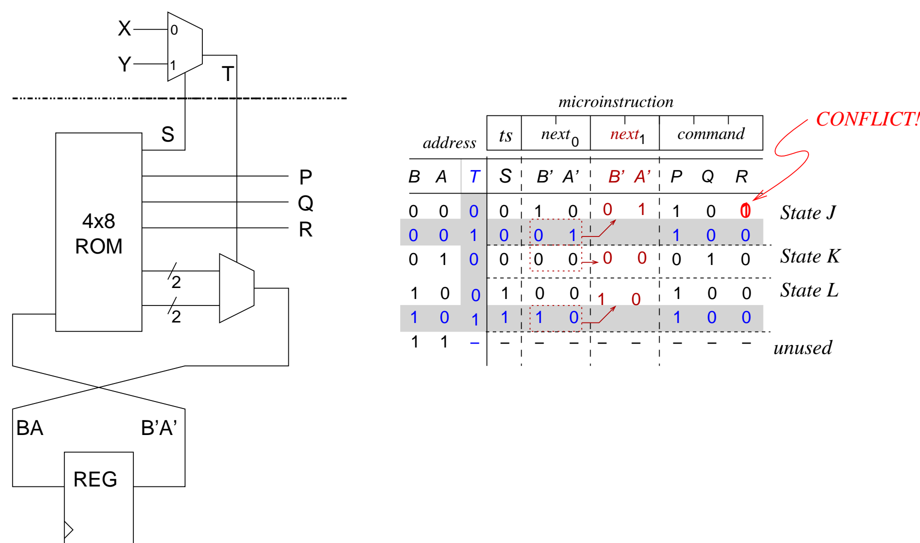
(*microinstruction* = *ts*, *next*, *command*)

The real price of this "orthogonality" is to restrict the form of the ASM; we are exchanging expressiveness for regularity in the implementation.

**Jump codes**

If you look at the ROM above, redundancy in the control outputs, P, Q, and R begins to be apparent. It requires two words to represent each state in the ASM. If we could merge these two words into one, we could again halve the size of the ROM.

The idea is to put both the alternative next-state values in a single word and externalize the selection.



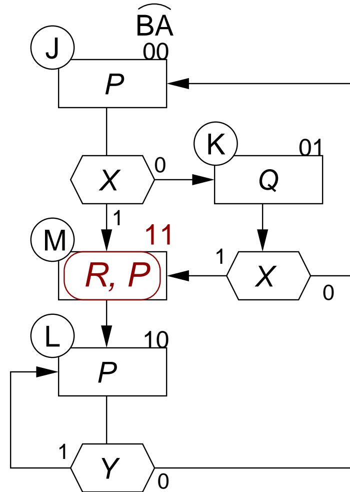

| B | A | T | S | B' | A' | B' | A' | P | Q | R | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | State J |
| 0 | 0 | 1 | 0 | 0 | 1 | | | 1 | 0 | 0 | |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | State K |
| 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | State L |
| 1 | 0 | 1 | 1 | 1 | 0 | | | 1 | 0 | 0 | |
| 1 | 1 | – | – | – | – | – | – | – | – | – | unused |

This merging results in a conflict in the conditional output, R. One way to handle this problem is to externalize the selection of conditional outputs, but we shall not pursue that avenue.

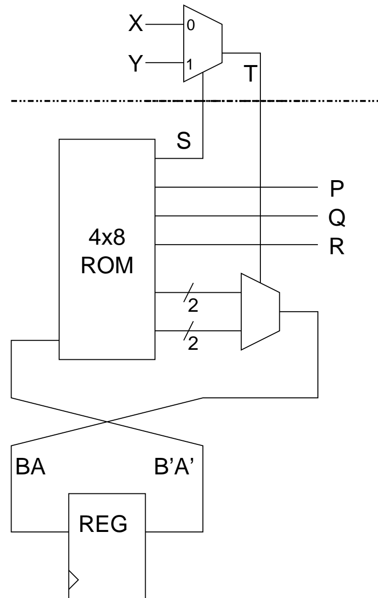
An alternative solution is, simply, to eliminate conditional outputs. This is a big sacrifice in expressiveness, but not necessarily in functionality. Usually, it is possible to write down an equivalent ASM, or one that is adequate if we make adjustments to the architecture. The modified ASM, below, is pretty close in

Copyright © 2003 Steven D. Johnson 

behavior to the initial ASM.




Fortunately, we have a spare encoding to assign to the newly introduced state, M.

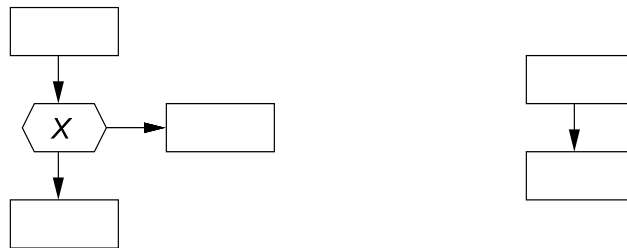



| *address* | | *microinstruction* | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | *ts* | *next*$_0$ | | *next*$_1$ | | *command* | | | |
| *B* | *A* | *S* | *B'* | *A'* | *B'* | *A'* | *P* | *Q* | *R* | |
| 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | *State J* |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | *State K* |
| 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | *State L* |
| 1 | 1 | – | 1 | 1 | 1 | 1 | 1 | 0 | 1 | *State M* |

Now we end up with a $4 \times 8$ microcode memory:

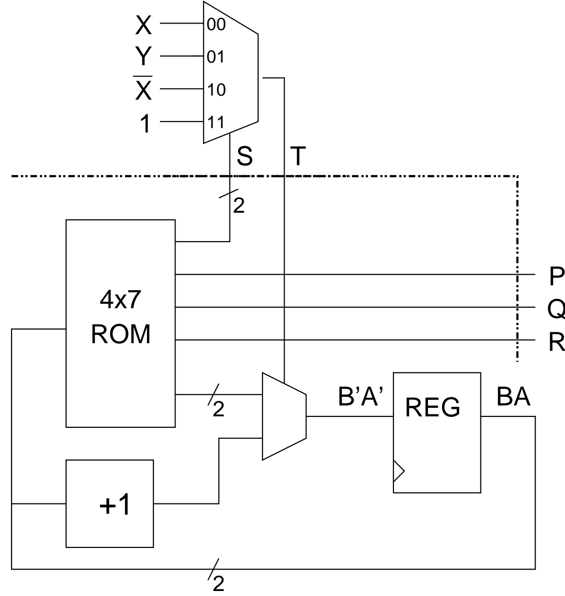
**Jump Bits**

For an ASM with a large number of states, storing two alternative next-state values is costly. The restrictions imposed on ASMs—single test, no conditional outputs—leave us with just two kinds of state transitions,




Since we have made these restrictions, perhaps there is a way to exploit them further. The refinement below reflects that fact that in such restricted ASMs, one can assign state encodings in such a way that, in most cases, one of the alternatives is the the incremented current-state value. An incrementer is added

to the sequencer to compute the next-state value accordingly.




| *address* | | *ts* | | *next*$_1$ | | *command* | | |
|---|---|---|---|---|---|---|---|---|
| $B$ | $A$ | $S_1$ | $S_0$ | $B'$ | $A'$ | $P$ | $Q$ | $R$ |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 |

```
J: assert P, if X jump to M
K: assert Q, if not X jump to J
M: assert R,P;
L: assert P; if Y jump to L
```

This implementation tactic opens the way for a more linear form of specification, looking something like sequential code. For our running illustration we need an additional test selection bits to enable us to reverse the sense of a test and provide a *unconditional branch* capability.

## The AM2910 microcode sequencer

Fig. 1 shows the architecture of the AM2910 microcode sequencer and the Logic Engine's generic implementation environment. The 2910 provides four sources for microinstruction addressing:

1. *Direct addressing* from an external source
2. *Initialized addressing* using an internal register
3. A *The micro-PC*, and internal counter
4. A five-deep *stack*

The generic sequencing architecture contains a *micro-pipeline* register which holds the current micro-instruction. It holds both the next-state value for microcontrol and the current commands presented to the design architecture. This register cleans up the asychronous and glitchy outputs from the *micro-control memory*.

The Logic Engine used a static RAM memory to hold the microcode, rather than a ROM. A micro-instruction contains three fields.

- The *sequencing operation* specifying how the next-instruction is to be addressed. The sequencing operation contains
  - A four-bit *operation code*, OP.
  - A four-bit *condition code*, X containing
    - CIN (active high), which increments the 2910's internal $\mu$PC.
    - CCEN (active low), enables testing of the design architecture's *test* signal, T.
    - CC.INV (active high), complements the truth value of the design architecture's *test* signal.
    - CC.FAIL overrides the design architecture's *test* signal, forcing the condition code to fail.
  - A 12-bit *direct address*, D, for the next microinstruction.
- A *command word* of up to 64 bits (the word capacity of the Logic Engine's control store). The command word typically has two fields,
  - A *test index* used to select a status condition from the architecture
  - A *command* presented to the architecture.

Copyright © 2003 Steven D. Johnson 

The 4-bit OP specifies one of the sixteen AM2910 instructions:

| *code* | OP | *meaning* |
|---|---|---|
| 0000 | JZ | Jump to location 0, clear the stack |
| 0001 | CJS | Conditional jump to subroutine at location $\mu$PC |
| 0010 | JMAP | Jump to the map address, provided externally |
| 0011 | CJP | Conditional jump to location $\mu$PC |
| 0100 | PUSH | Push with conditional load of $\mu$PC |
| 0101 | JSRP | Subroutine jump to R or $\mu$PC |
| 0110 | CJV | Conditional jump to the externally provided *vector* address |
| 0111 | JRP | Jump to R or $\mu$PC |
| 1000 | RFCT | Repeat loop at R if R$\neq$ 0 and decrement R |
| 1001 | RPCT | Jump to $\mu$PC if R $\neq$ 0 |
| 1010 | CRTN | Conditional return from subroutine. |
| 1011 | LDCT | Load $\mu$PC and continue |
| 1100 | CJPP | Conditional jump to $\mu$PC and stack pop |
| 1101 | LOOP | Conditionally jump to stack-top and pop the stack |
| 1110 | CONT | Continue |
| 1111 | TWB | Three-way branch to stack-top (CC true), D with stack-pop (CC false), or with stack-pop (CC false), or $\mu$PC (CC.EN false) |

**Logic Engine Assembly Language**

Copyright © 2003 Steven D. Johnson DRAFT MATERIAL December 4, 2006

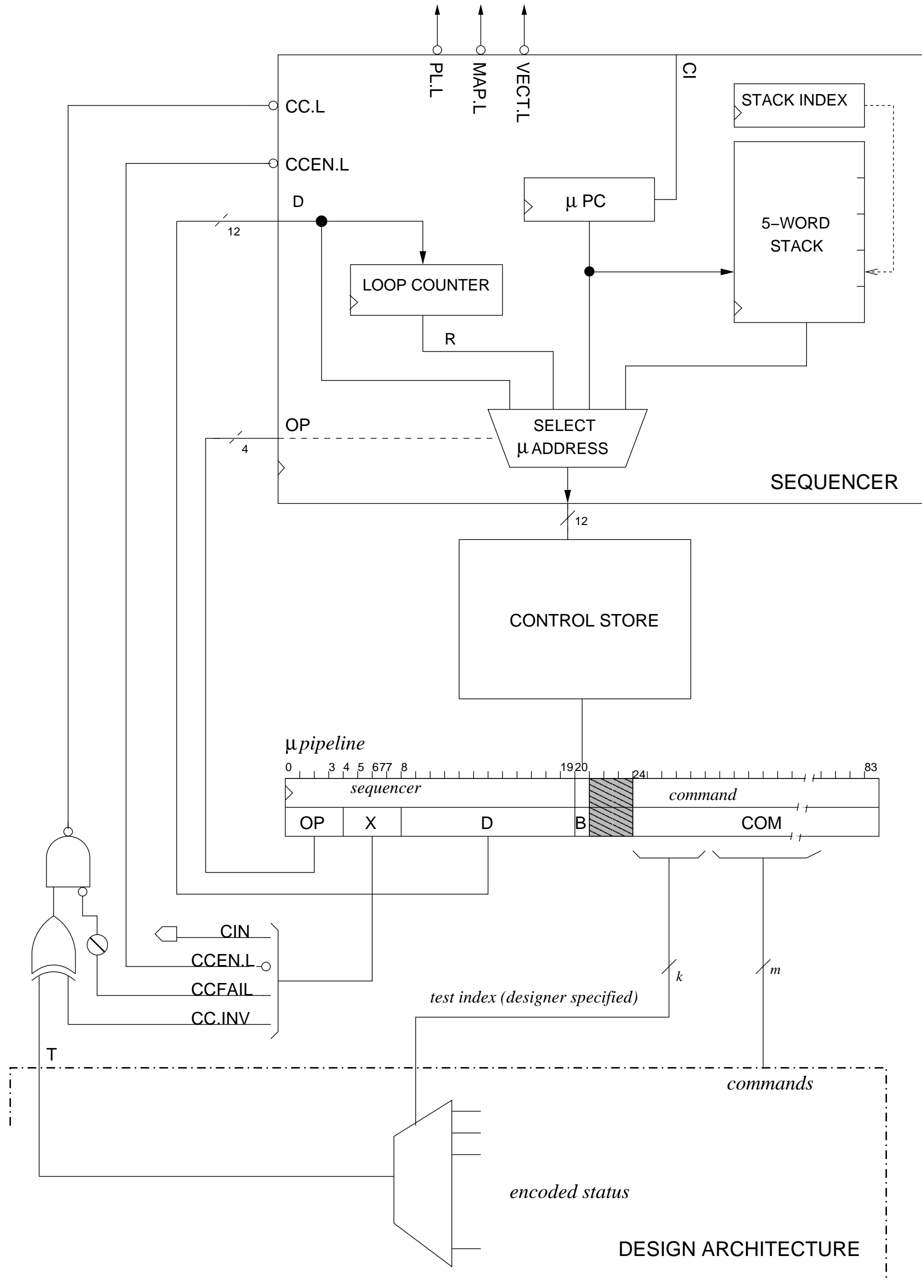



Figure 1: AM2910 Architecture

Copyright © 2003 Steven D. Johnson 