Implement of Processor FSMs

- Classical Finite State Machine Design
- Divide and Conquer Approach: Time-State Method
  - Partition FSM into multiple communicating FSMs
- Exploit MSI Functionality: Jump Counters
  - Counters, Multiplexers, Decoders
- Microprogramming: ROM-based methods
  - Direct encoding of next states and outputs
Problem:
- The processor and memory often do not share the same clock.

Solution:
- Use appropriate handshaking

Moore

Note capture of MBR in these states
Memory-Register Interface Timing

Valid data latched on IF2 to IF3 transition because data must be valid before Wait can go low.

Processor Signal Flow

[Diagram showing processor signal flow with interconnections between components such as Memory Address Bus, Result Bus, Address Bus, Memory Data Bus, etc.]

EECS150 - Fall 2001
Moore Machine Diagram

16 states, 4 bit state register
Next State Logic: 9 Inputs, 4 Outputs
Output Logic: 4 Inputs, 18 Outputs

These can be implemented via ROM or PAL/PLA

Next State: 512 x 4 bit ROM
Output: 16 x 18 bit ROM

Moore Machine State Table

<table>
<thead>
<tr>
<th>Reset</th>
<th>Wait</th>
<th>IR&lt;15&gt;</th>
<th>IR&lt;14&gt;</th>
<th>AC&lt;15&gt;</th>
<th>Current State</th>
<th>Next State</th>
<th>Register Transfer Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>RES (0000)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>RES (0000)</td>
<td>IF0 (0001)</td>
<td>0 → PC</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>IF0 (0001)</td>
<td>IF1 (0010)</td>
<td>PC → MAR, PC + 1 → PC</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>IF1 (0010)</td>
<td>IF1 (0010)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>IF1 (0010)</td>
<td>IF2 (0011)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>IF2 (0011)</td>
<td>IF2 (0011)</td>
<td>MAR → Mem, Read,</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>IF2 (0011)</td>
<td>IF3 (0100)</td>
<td>Request, Mem → MBR</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>IF3 (0100)</td>
<td>IF3 (0100)</td>
<td>MBR → IR</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>IF3 (0100)</td>
<td>OD (0101)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>OD (0101)</td>
<td>LD0 (0110)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>X</td>
<td>OD (0101)</td>
<td>BR0 (1110)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>LD0 (0110)</td>
<td>LD1 (0111)</td>
<td>IR → MAR</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>LD1 (0111)</td>
<td>LD1 (0111)</td>
<td>MAR → Mem, Read,</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>LD2 (1000)</td>
<td>IF0 (0001)</td>
<td>Request, Mem → MBR</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>LD2 (1000)</td>
<td>MBR → AC</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>ST0 (1001)</td>
<td>ST1 (1010)</td>
<td>IR → MAR, AC → MBR</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>ST1 (1010)</td>
<td>ST1 (1010)</td>
<td>MAR → Mem, Write,</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>ST1 (1010)</td>
<td>IF0 (0001)</td>
<td>Request, MBR → Mem</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>AD0 (1011)</td>
<td>AD1 (1100)</td>
<td>IR → MAR</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>AD1 (1100)</td>
<td>AD1 (1100)</td>
<td>MAR → Mem, Read,</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>AD1 (1100)</td>
<td>AD2 (1101)</td>
<td>Request, Mem → MBR</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>AD2 (1101)</td>
<td>IF0 (0001)</td>
<td>MBR + AC → AC</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>0</td>
<td>BR0 (1110)</td>
<td>IF0 (0001)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>1</td>
<td>BR0 (1110)</td>
<td>BR1 (1111)</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>BR1 (1111)</td>
<td>IF0 (0001)</td>
<td>IR → PC</td>
</tr>
</tbody>
</table>
State Transition Table

- Observations:
  - Extensive use of Don't Cares
  - Inputs used only in a small number of state
    e.g., AC<15> examined only in BR0 state
    IR<15:14> examined only in OD state
  - Some outputs always asserted in a group
  - ROM-based implementations cannot take
    advantage of don't cares
  - However, ROM-based implementation can
    skip state assignment step

Synchronous Mealy Machines

- Standard Mealy Machine has asynchronous outputs
- These change in response to input changes, independent
  of clock
- Revise Mealy Machine design so outputs change only on
  clock edges
- One approach: non-overlapping clocks
Synchronous Mealy Machines

Case I: Synchronizers at Inputs and Outputs

A asserted in Cycle 0, \( f \) becomes asserted after 2 cycle delay!
This is clearly overkill!

Case II: Synchronizers on Inputs

A asserted in Cycle 0, \( f \) follows in next cycle
Same as using delayed signal (\( A' \)) in Cycle 1!
Synchronous Mealy Machines

Case III: Synchronized Outputs

A asserted during Cycle 0, $f'$ asserted in next cycle
Effect of $f$ delayed one cycle

Synchronous Mealy Machines

- Implications for Processor FSM Already Derived
- Consider inputs: Reset, Wait, IR<15:14>, AC<15>
  - Latter two already come from registers, and are sync'd to clock
  - Possible to load IR with new instruction in one state & perform multiway branch on opcode in next state
  - Best solution for Reset and Wait: synchronized inputs
    - Place D flipflops between these external signals and the control inputs to the processor FSM
    - Sync'd versions of Reset and Wait delayed by one clock cycle
Time State Divide and Conquer

- Overview
  - Classical Approach: Monolithic Implementations
  - Alternative "Divide & Conquer" Approach:
    - Decompose FSM into several simpler communicating FSMs
    - Time state FSM (e.g., IFetch, Decode, Execute)
    - Instruction state FSM (e.g., LD, ST, ADD, BRN)
    - Condition state FSM (e.g., AC < 0, AC ≠ 0)

Time State (Divide & Conquer)

**Time State FSM**
Most instructions follow same basic sequence
Differ only in detailed execution sequence
Time State FSM can be parameterized by opcode and AC states

- Instruction State: stored in IR<15:14>
- Condition State: stored in AC<15>

Diagram showing states and transitions for time state FSM.
### Time State (Divide & Conquer)

#### Generation of Microoperations

0 → PC: Reset
PC + 1 → PC: T0
PC → MAR: T0
MAR → Memory Address Bus: T2 + T6 • (LD + ST + ADD)
Memory Data Bus → MBR: T2 + T6 • (LD + ADD)
MBR → Memory Data Bus: T6 • ST
MBR → IR: T4
MBR → AC: T7 • LD
AC → MBR: T5 • ST
AC + MBR → AC: T7 • ADD
IR<13:0> → MAR: T5 • (LD + ST + ADD)
IR<13:0> → PC: T6 • BRN
1 → Read/Write: T2 + T6 • (LD + ADD)
0 → Read/Write: T6 • ST
1 → Request: T2 + T6 • (LD + ST + ADD)

---

### Jump Counter

#### Concept

Implement FSM using MSI functionality: counters, mux, decoders

**Pure jump counter:** only one of four possible next states

![Jump Counter Diagram]

**Hybrid jump counter:**

Multiple "Jump States" — function of current state + inputs
Jump Counters

Pure Jump Counter

Inputs
Count, Load, Clear Logic
Jump State Logic
Clear
Load
Count
CLOCK

Logic blocks implemented via discrete logic, PALs/PLAs, ROMs

NOTE: No inputs to jump state logic

Problem with Pure Jump Counter

Difficult to implement multi-way branches

Logical State Diagram

Extra States:

Pure Jump Counter State Diagram
Jump Counters

Hybrid Jump Counter

Load inputs are function of state and FSM inputs

Synchronous Counter
State Register

Implementation Example

State assignment attempts to take advantage of sequential states
Jump Counters

Implementation Example, Continued

\[
\text{CNT} = (s_0 + s_5 + s_8 + s_{10}) + \text{Wait} \cdot (s_1 + s_3) + \text{Wait} \cdot (s_2 + s_6 + s_9 + s_{11})
\]

\[
\text{CLR} = \text{Reset} \cdot s_7 \cdot s_{12} \cdot s_{13} \cdot (s_9 \cdot \text{Wait})
\]

\[
\text{LD} = s_4
\]

Contents of Jump State ROM

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents (Symbolic State)</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0101 (LD0)</td>
</tr>
<tr>
<td>01</td>
<td>1000 (ST0)</td>
</tr>
<tr>
<td>10</td>
<td>1010 (AD0)</td>
</tr>
<tr>
<td>11</td>
<td>1101 (BR0)</td>
</tr>
</tbody>
</table>

Jump Counters

Implementation Example, continued

Implement CNT using active low PAL

Implement CLR

NOTE: Active low outputs from decoder
Jump Counter

CLR, CNT, LD implemented via Mux Logic

CLR = CLRm + Reset
CLR = CLRm + Reset

Active Lo outputs: hi input inverted at the output

Note that CNT is active hi on counter so invert MUX inputs!

Jump Counters

Microoperation implementation

0 → PC = Reset
PC + 1 → PC = S0
PC → MAR = S0
MAR → Memory Address Bus = Wait(S1 + S2 + S5 + S6 + S8 + S9 + S11 + S12)
Memory Data Bus → MBR = Wait(S2 + S6 + S11)
MBR → Memory Data Bus = Wait(S8 + S9)
MBR → IR = Wait+S3
MBR → AC = Wait+S7
AC → MBR = IR15+IR14+S4
AC + MBR → AC = Wait+S12
IR<13:0> → MAR = (IR15+IR14 + IR15+IR14 + IR15+IR14)+S4
IR<13:0> → PC = AC15+S13
1 → Read/Write = Wait(S1 + S2 + S5 + S6 + S11 + S12)
0 → Read/Write = Wait(S8 + S9)
1 → Request = Wait(S1 + S2 + S5 + S6 + S8 + S9 + S11 + S12)

Jump Counters: CNT, CLR, LD function of current state + Wait
Why not store these as outputs of the Jump State ROM?
Make Wait and Current State part of ROM address
32 x as many words, 7 bits wide
Branch Sequencers

Concept
Implement Next State Logic via ROM
Address ROM with current state and inputs
Problem: ROM doubles in size for each additional input
Note: Jump counter trades off ROM size vs. external logic
Even in hybrid approach, state + input subset form ROM address

Branch Sequencer: between the extremes
Next State stored in ROM
Each state limited to small number of next states
Always a power of 2

Observe: only a small set of inputs are examined in any state

Branch Sequencers

4 Way Branch Sequencer

Current State selects two inputs to form part of ROM address
These select one of four possible next states (and output sets)
Every state has exactly four possible next states

EECS150 - Fall 2001
Branch Sequenter

**Processor CPU Design Example**

Alpha, Beta multiplexer input setup

---

**Example Processor FSM**

<table>
<thead>
<tr>
<th>ROM ADDRESS (Reset, Current State, a, b)</th>
<th>ROM CONTENTS</th>
<th>Next State</th>
<th>Register Transfer Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>RES 0 0000 X X</td>
<td>0001 (IF0)</td>
<td>PC → MAR, PC + 1 → PC</td>
<td></td>
</tr>
<tr>
<td>IF0 0 0001 0 0</td>
<td>0001 (IF0)</td>
<td></td>
<td>MAR → Mem, Read, Request</td>
</tr>
<tr>
<td>IF0 0 0010 1 1</td>
<td>0010 (IF1)</td>
<td></td>
<td>Mem → MBR</td>
</tr>
<tr>
<td>IF1 0 0010 0 0</td>
<td>0011 (IF2)</td>
<td>MAR → Mem, Read, Request</td>
<td></td>
</tr>
<tr>
<td>IF2 0 0011 0 0</td>
<td>0011 (IF2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OD 0 0100 0 0</td>
<td>0101 (LD0)</td>
<td>IR → MAR</td>
<td></td>
</tr>
<tr>
<td>OD 0 0100 0 1</td>
<td>1000 (ST0)</td>
<td>IR → MAR, AC → MBR</td>
<td></td>
</tr>
<tr>
<td>OD 0 0100 1 0</td>
<td>1001 (AD0)</td>
<td>IR → MAR</td>
<td></td>
</tr>
<tr>
<td>OD 0 0100 1 1</td>
<td>1101 (BR0)</td>
<td>IR → MAR</td>
<td></td>
</tr>
<tr>
<td>LD0 0 0111 X X</td>
<td>0110 (LD1)</td>
<td>MAR → Mem, Read, Request</td>
<td></td>
</tr>
<tr>
<td>LD1 0 0111 0 0</td>
<td>0111 (LD2)</td>
<td>Mem → MBR</td>
<td></td>
</tr>
<tr>
<td>LD2 0 0111 1 1</td>
<td>0110 (LD1)</td>
<td>MAR → Mem, Read, Request</td>
<td></td>
</tr>
<tr>
<td>LD2 0 0111 1 1</td>
<td>0110 (LD1)</td>
<td></td>
<td>MBR + AC → AC</td>
</tr>
<tr>
<td>ST0 0 1000 X X</td>
<td>1001 (ST1)</td>
<td>MAR → Mem, Write, Request, MBR → Mem</td>
<td></td>
</tr>
<tr>
<td>ST1 0 1001 0 0</td>
<td>0000 (RES)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ST1 0 1001 0 1</td>
<td>1001 (ST1)</td>
<td>MAR → Mem, Write, Request, MBR → Mem</td>
<td></td>
</tr>
<tr>
<td>AD0 0 1010 X X</td>
<td>1011 (AD1)</td>
<td>MAR → Mem, Read, Request</td>
<td></td>
</tr>
<tr>
<td>AD1 0 1011 0 0</td>
<td>1100 (AD2)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AD2 0 1011 1 1</td>
<td>1011 (AD1)</td>
<td>MAR → Mem, Read, Request</td>
<td></td>
</tr>
<tr>
<td>BR0 0 1100 X X</td>
<td>0000 (RES)</td>
<td>MBR + AC → AC</td>
<td></td>
</tr>
<tr>
<td>BR1 0 1101 0 0</td>
<td>0000 (RES)</td>
<td>IR → AC</td>
<td></td>
</tr>
<tr>
<td>BR2 0 1101 1 1</td>
<td>0000 (RES)</td>
<td>IR → AC</td>
<td></td>
</tr>
</tbody>
</table>
**Branch Sequencers**

Alternative Horizontal Implementation

Input MUX controlled by encoded signals, not state
Much fewer inputs than unique states!
In example FSM, input MUX can be 2:1!

Adding length to ROM word saves on bits vs. doubling words
Vertical format: \((14 + 4) \times 64 = 1152\) ROM bits
Horizontal format: \((14 + 4 \times 4 + 2) \times 16 = 512\) ROM bits

**Microprogramming**

How to organize the control signals
Implement control signals by storing 1’s and 0’s in a ROM

*Horizontal vs. vertical microprogramming*

Horizontal: 1 ROM output for each control signal
Vertical: encoded control signals in ROM, decoded externally
some mutually exclusive signals can be combined helps reduce ROM length
Microprogramming

Register Transfer/Microoperations

14 Register Transfer operations become 22 Microoperations:

PC → ABUS
IR → ABUS
MBR → ABUS
RBUS → AC
AC → ALU A
MBUS → ALU B
ALU ADD
ALU PASS B
MAR → Address Bus
MBR → Data Bus
ABUS → IR

ABUS → MAR
Data Bus → MBR
MBR → MBR
MBR → MBUS
0 → PC
PC + 1 → PC
ABUS → PC
Read/Write
Request
AC → RBUS
ALU Result → RBUS

Horizontal Microprogramming

Horizontal Branch Sequencer

α, β Mux bits
4 x 4 Next State bits
22 Control operation bits
40 bits total
## Moore Processor ROM

| Current State (Address) | BR1 (1111) | BR0 (1110) | AD2 (1111) | AD1 (1100) | AD0 (1100) | ST1 (1010) | ST0 (1001) | LD2 (1000) | LD1 (0111) | LD0 (0110) | IF3 (000) | IF2 (0011) | IF1 (0010) | IF0 (0001) | RES (0000) |
|------------------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|-----------|-----------|-----------|-----------|-----------|-----------|
|                        | 0001       | 0010       | 0010       | 1100       | 1110       | 0001       | 0010       | 1100       | 0001       | 0010       | 1100       | 1110      | 0001      | 0010      | 1100      | 0001      |

### Next States

<table>
<thead>
<tr>
<th>Current State (Address)</th>
<th>BR1 (1111)</th>
<th>BR0 (1110)</th>
<th>AD2 (1111)</th>
<th>AD1 (1100)</th>
<th>AD0 (1100)</th>
<th>ST1 (1010)</th>
<th>ST0 (1001)</th>
<th>LD2 (1000)</th>
<th>LD1 (0111)</th>
<th>LD0 (0110)</th>
<th>IF3 (000)</th>
<th>IF2 (0011)</th>
<th>IF1 (0010)</th>
<th>IF0 (0001)</th>
<th>RES (0000)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0001</td>
<td>0010</td>
<td>0010</td>
<td>1100</td>
<td>1110</td>
<td>0001</td>
<td>0010</td>
<td>1100</td>
<td>0001</td>
<td>0010</td>
<td>1100</td>
<td>1110</td>
<td>0001</td>
<td>0010</td>
<td>1100</td>
</tr>
</tbody>
</table>

### Alpha Inputs

<table>
<thead>
<tr>
<th>Alpha Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Wait</td>
</tr>
<tr>
<td>1</td>
<td>IR&lt;15&gt;</td>
</tr>
</tbody>
</table>

### Beta Inputs

<table>
<thead>
<tr>
<th>Beta Inputs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AC&lt;15&gt;</td>
</tr>
<tr>
<td>1</td>
<td>IR&lt;14&gt;</td>
</tr>
</tbody>
</table>

---

### Horizontal Microprogramming

**Advantages:**

- Most flexibility -- complete parallel access to datapath control points

**Disadvantages:**

- Very long control words -- 100+ bits for real processors

NOTE: Not all microoperation combinations make sense!

### Output Encodings:

- Group mutually exclusive signals
- Use external logic to decode

### Example:

- \( 0 \rightarrow \text{PC}, \text{PC} + 1 \rightarrow \text{PC}, \text{ABUS} \rightarrow \text{PC} \) mutually exclusive

- Save ROM bit with external 2:4 Decoder

---
Horizontal Microprogramming

Partially Encoded Control Outputs

<table>
<thead>
<tr>
<th>ALU ADD</th>
<th>ALU PASS B</th>
<th>MAR → Address Bus</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBR → Data Bus</td>
<td>ABUS → MAR</td>
<td>Request</td>
</tr>
<tr>
<td>RBUS → MBR</td>
<td>AC → RBUS</td>
<td></td>
</tr>
<tr>
<td>MBUS → ALU B</td>
<td>MBR → MBUS</td>
<td></td>
</tr>
<tr>
<td>AC → ALU A</td>
<td>ALU Result → RBUS</td>
<td></td>
</tr>
<tr>
<td>MBR → ABUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABUS → IR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 → PC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC → ABUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>IR → ABUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data Bus → MBR</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

More extensive encoding to reduce ROM word length

- Typically use multiple microword formats:
  - Horizontal microcode -- next state + control bits in same word
  - Separate formats for control outputs and "branch jumps"
  - may require several microwords in a sequence to implement same function as single horizontal word
- In the extreme, very much like assembly language programming
Vertical Microprogramming

**Branch Jump Format**

<table>
<thead>
<tr>
<th>Type</th>
<th>Condition Select</th>
<th>Condition Compare</th>
<th>Next Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>6</td>
</tr>
</tbody>
</table>

00 = Wait
01 = AC<15>
10 = IR<15>
11 = IR<14>

**Register Transfer Format**

<table>
<thead>
<tr>
<th>Source</th>
<th>Destination</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

000: NO OP
001: RBUS → AC
010: MBUS → IR
011: ABUS → MAR
100: M → MBR
101: AC → RBUS
110: ALU Res → RBUS
111: MBR → M

**ROM Address Symbolic Contents**

<table>
<thead>
<tr>
<th>ROM Address</th>
<th>Symbolic Contents</th>
<th>Binary Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>RES RT PC → MAR, PC +1 → PC</td>
<td>0 001 011 100</td>
</tr>
<tr>
<td>000001</td>
<td>IF0 RT MAR → M, Read</td>
<td>0 100 000 101</td>
</tr>
<tr>
<td>000010</td>
<td>BJ Wait=0, IF0</td>
<td>1 000 000 001</td>
</tr>
<tr>
<td>000011</td>
<td>IF1 RT MAR → M, M → MBR, Read</td>
<td>0 100 100 101</td>
</tr>
<tr>
<td>000100</td>
<td>BJ Wait=1, IF1</td>
<td>1 001 000 011</td>
</tr>
<tr>
<td>000101</td>
<td>IF2 RT MBR → IR</td>
<td>0 011 010 000</td>
</tr>
<tr>
<td>000110</td>
<td>BJ Wait=0, IF2</td>
<td>1 000 000 101</td>
</tr>
<tr>
<td>000111</td>
<td>RT IR → MAR</td>
<td>0 010 011 000</td>
</tr>
<tr>
<td>001000</td>
<td>OD BJ IR&lt;15&gt;=1, OD1</td>
<td>1 101 010 101</td>
</tr>
<tr>
<td>001001</td>
<td>BJ IR&lt;14&gt;=1, ST0</td>
<td>1 101 010 000</td>
</tr>
<tr>
<td>001010</td>
<td>LD0 RT MAR → M, Read</td>
<td>0 100 000 101</td>
</tr>
<tr>
<td>001011</td>
<td>LD1 RT MAR → M, M → MBR, Read</td>
<td>0 100 100 101</td>
</tr>
<tr>
<td>001100</td>
<td>BJ Wait=1, LD1</td>
<td>1 001 001 011</td>
</tr>
<tr>
<td>001101</td>
<td>LD2 RT MBR → AC</td>
<td>0 110 001 010</td>
</tr>
<tr>
<td>001110</td>
<td>BJ Wait=0, RES</td>
<td>1 000 000 000</td>
</tr>
<tr>
<td>001111</td>
<td>BJ Wait=1, RES</td>
<td>1 001 000 000</td>
</tr>
</tbody>
</table>

EECS150 - Fall 2001
**Vertical Microprogramming**

<table>
<thead>
<tr>
<th>ROM ADDRESS</th>
<th>SYMBOLIC CONTENTS</th>
<th>BINARY CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>ST0 RT AC → MBR</td>
<td>0 101 101 000</td>
</tr>
<tr>
<td>010001</td>
<td>RT MAR → M, MBR → M, Write</td>
<td>0 100 111 110</td>
</tr>
<tr>
<td>010010</td>
<td>ST1 RT MAR → M, MBR → M, Write</td>
<td>0 100 111 110</td>
</tr>
<tr>
<td>010011</td>
<td>BJ Wait=0, RES</td>
<td>1 000 000 000</td>
</tr>
<tr>
<td>010100</td>
<td>BJ Wait=1, ST1</td>
<td>1 001 010 010</td>
</tr>
<tr>
<td>010101</td>
<td>OD1 BJ IR&lt;14&gt;=1, BR0</td>
<td>1 111 011 101</td>
</tr>
<tr>
<td>010110</td>
<td>AD0 RT MAR → M, Read</td>
<td>0 100 000 101</td>
</tr>
<tr>
<td>010111</td>
<td>AD1 RT MAR → M, M → MBR, Read</td>
<td>0 100 100 101</td>
</tr>
<tr>
<td>011000</td>
<td>BJ Wait=1, AD1</td>
<td>1 001 010 111</td>
</tr>
<tr>
<td>011001</td>
<td>AD2 RT AC + MBR → AC</td>
<td>0 110 001 001</td>
</tr>
<tr>
<td>011010</td>
<td>BJ Wait=0, RES</td>
<td>1 000 000 000</td>
</tr>
<tr>
<td>011011</td>
<td>BJ Wait=1, RES</td>
<td>1 000 000 000</td>
</tr>
<tr>
<td>011100</td>
<td>BR0 BJ AC&lt;15&gt;=0, RES</td>
<td>1 010 000 000</td>
</tr>
<tr>
<td>011101</td>
<td>RT IR → PC</td>
<td>0 010 110 000</td>
</tr>
<tr>
<td>011110</td>
<td>BJ AC&lt;15&gt;=1, RES</td>
<td>1 011 000 000</td>
</tr>
</tbody>
</table>

31 words x 10 ROM bits = 310 bits total versus 16 x 38 = 608 bits horizontal

**Vertical Programming**

Controller Block Diagram

*Controller Block Diagram*
Vertical Microprogramming

**Condition Logic**

- **Condition Selector**
- **Condition Comparator**
- **Microinstruction Type**

Diagram:

```
        Condition Selector
          ↓                ↓
         4:1 MUX          ⊕
   Wait AC<15>  IR<15> IR<14>  Microinstruction Type

Microinstruction Type
         ⊕
        ↓                ↓
         LD              CNT
```

**Writeable Control Store**
- Part of control store addresses map into RAM
  - Allows assembly language programmer to implement own instructions
  - Extend "native" instruction set with application specific instructions
  - Requires considerable sophistication to write microcode
  - Not a popular approach with today's processors
- Make the native instruction set simple and fast
- Write "higher level" functions as assembly language sequences