**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

**Processor / Memory Interface**

**Problem:**
- The processor and memory often do not share the same clock.

**Solution:**
- Use appropriate handshaking

**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 of processor signal flow]
Moore Machine Diagram

Moore Machine State Table

<table>
<thead>
<tr>
<th>Current State</th>
<th>Next State</th>
<th>Register Transfer Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 X X X X</td>
<td>0000</td>
<td>RES (0000)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>0001</td>
<td>IF0 (0001)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>0010</td>
<td>IF0 (0001) + IF1 (0010)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>0011</td>
<td>IF0 (0001) + IF1 (0010)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>0100</td>
<td>IF0 (0001) + IF2 (0011)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>0101</td>
<td>IF0 (0001) + IF2 (0011)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>0110</td>
<td>IF0 (0001) + IF3 (0100)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>0111</td>
<td>IF0 (0001) + IF3 (0100)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>1000</td>
<td>LD0 (0110)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>1001</td>
<td>LD0 (0110)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>1010</td>
<td>LD0 (0110)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>1011</td>
<td>LD0 (0110)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>1100</td>
<td>LD0 (0110)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>1101</td>
<td>LD0 (0110)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>1110</td>
<td>LD0 (0110)</td>
</tr>
<tr>
<td>0 X X X X</td>
<td>1111</td>
<td>LD0 (0110)</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

- Case I: Synchronizers at Inputs and Outputs
  - A asserted in Cycle 0, f becomes asserted after 2 cycle delay
  - This is clearly overkill!

Synchronous Mealy Machine

- Case II: Synchronizers on Inputs
  - A asserted in Cycle 0, f follows in next cycle
  - Same as using delayed signal (A') in Cycle II
### Synchronous Mealy Machines

**Case III: Synchronized Outputs**
- A asserted during Cycle 0.
- $f'$ asserted in next cycle.
- Effect of $f$ delayed one cycle.

### Time State (Divide & Conquer)

**Time State FSM**
- Most instructions follow some 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>

#### Generation of Microoperations

- **0 → PC:** Reset
- **PC + 1 → PC:** T0
- **PC + 0 → PC:** T1
- **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:** T5 • LD
- **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)

### Time State Divide and Conquer

- **Overview**
  - Classical Approach: Monolithic Implementations
  - Alternative "Divide & Conquer" Approach:
    - Decompose FSM into several simpler communicating FSMS.
    - Instruction state FSM (e.g., LD, ST, ADD, BRN)
    - Condition state FSM (e.g., AC < 0, AC ≥ 0)

### Jump Counter

- **Concept**
  - Implement FSM using MSI functionality: counters, mux, decoders.
- **Pure jump counter:** only one of four possible next states.
- **Hybrid jump counter:** Multiple "Jump States" — function of current state + inputs.
Jump Counters

Pure Jump Counter

- Logic implemented via discrete logic, PALs/PLAs, ROMs
- No inputs to jump state logic

Jump Counters

Implementation Example

State assignment attempts to take advantage of sequential states

Jump Counters

Problem with Pure Jump Counter

Difficult to implement multi-way branches

Jump Counters

Hybrid Jump Counter

- Load inputs are functions of state and FSM inputs

Jump Counters

Implementation Example, Continued

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

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

\[
\text{LD} = s_4
\]

Jump Counters

Implementation Example, continued

- Implement CNT using active low PAL

- Implement CLR using active low decoder

Jump Counters

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

**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

Only jump states kept in ROM

Even in hybrid approach, state + input subset form ROM address

Jump Counter:
- **CLR, CNT, LD** implemented via Mux Logic
- CLR = CLRm + Reset
- CNT = CLRm + Reset

Active Lo outputs:
- hi input inverted at the output

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

### 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

Only jump states kept in ROM

Even in hybrid approach, state + input subset form ROM address

Branch Sequencers 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

### Example Processor FSM

<table>
<thead>
<tr>
<th>ROM ADDRESS</th>
<th>ROM CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(State, Current State, a, b)</td>
<td>Next State, Register Transfer Operations</td>
</tr>
<tr>
<td>RES</td>
<td>00000</td>
</tr>
<tr>
<td>IPS</td>
<td>00001</td>
</tr>
<tr>
<td>IF1</td>
<td>00011</td>
</tr>
<tr>
<td>IF2</td>
<td>00101</td>
</tr>
<tr>
<td>CD</td>
<td>00111</td>
</tr>
<tr>
<td>OD</td>
<td>01000</td>
</tr>
<tr>
<td>LD</td>
<td>01001</td>
</tr>
<tr>
<td>LD1</td>
<td>01010</td>
</tr>
<tr>
<td>LD2</td>
<td>01011</td>
</tr>
<tr>
<td>ST0</td>
<td>01100</td>
</tr>
<tr>
<td>ST1</td>
<td>01100</td>
</tr>
<tr>
<td>AD0</td>
<td>10000</td>
</tr>
<tr>
<td>AD1</td>
<td>10100</td>
</tr>
<tr>
<td>AD2</td>
<td>10010</td>
</tr>
<tr>
<td>BRS</td>
<td>00100</td>
</tr>
<tr>
<td>BRS1</td>
<td>01101</td>
</tr>
</tbody>
</table>

**Register Transfer Operations**
- PC ← MAR, PC + 1 ← PC
- MAR ← Mem, Read, Request
- Mem ← Mem, Read, Request
- IR ← MAR, AC ← MBR
- IR ← MAR
- IR ← Mem, Read, Request
- IR ← Mem, Read, Request
- IR ← Mem, Write, Request, MBR ← Mem
- IR ← Mem, Write, Request, MBR ← Mem
- MAR ← Mem, Read, Request
- MAR ← Mem, Read, Request
- MAR ← Mem, Write, Request, MBR ← Mem
- MAR ← Mem, Read, Request
- MAR ← Mem, Write, Request, MBR ← Mem
- MAR ← Mem, Write, Request, MBR ← Mem
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 x 4) x 64 = 256 ROM bits

Horizontal format: (14 x 4 x 2) x 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

14 Register Transfer operations become 22 Microoperations:

PC -> ABUS
IR -> ABUS
MBR -> AC
MBR -> ALU B
ALU ADV
ALU ADD
MBR -> Adder Bus
ABUS -> IR
ABUS -> MAR
Data Bus -> MBR
RBUS -> MBR
RBUS -> RBUS
α -> PC
PC -> PC
PC -> PC
Read/Write Request
AC -> RBUS
ALU Result -> RBUS

Microprogramming

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

<table>
<thead>
<tr>
<th>Current State</th>
<th>O</th>
<th>A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>Next States</th>
<th>PC</th>
<th>IR</th>
<th>ALU B</th>
<th>MAR</th>
<th>Data Bus</th>
<th>MBR</th>
<th>RBUS</th>
<th>Request</th>
<th>ALU</th>
<th>Result</th>
<th>Address Bus</th>
<th>PC + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0001</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Horizontal Microprogramming

Advantages:
- most flexibility -- complete parallel access to datapath control points

Disadvantages:
- vary 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 → PC, PC + 1 → PC, ABUS → PC mutually exclusive
- Save ROM bit with external 2:4 Decoder

EECS150 - Fall 2001
**Horizontal Microprogramming**

- Partially Encoded Control Outputs
  - Typical use multiple microword formats:
    - More extensive encoding to reduce ROM word length
    - Separate formats for control outputs and "branch jumps"
    - In the extreme, very much like assembly language programming.

**Vertical Microprogramming**

- 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"
  - In the extreme, very much like assembly language programming.

**Vertical Programming**

- Controller Block Diagram
Vertical Microprogramming

- 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