Overview

- Alternative controller FSM implementation approaches based on:
  - classical Moore and Mealy machines
  - jump counters
  - microprogramming (ROM) based approaches
  - branch sequencers
  - horizontal microcode
  - vertical microcode

Alternative Ways to Implement Processor FSMs

- "Random Logic" based on Moore and Mealy Design
  - 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

Random Logic

- Perhaps poor choice of terms for "classical" FSMs
- Contrast with structured logic: PAL/PLA, PGA, ROM
- Could just as easily construct Moore and Mealy machines with these components

Moore Machine State Diagram

Memory-Register Interface Timing

Moore Machine Diagram

16 states, 4 bit state register
Next State Logic: 9 Inputs, 4 Outputs
Output Logic: 4 Inputs, 16 Outputs
These can be implemented via ROM or PAL/PLA
Output: 16 x 8 bit ROM
**Moore Machine State Table**

<table>
<thead>
<tr>
<th>Next State</th>
<th>Input</th>
<th>Current State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0111</td>
<td>0111</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0110</td>
<td>0110</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0100</td>
<td>0100</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0010</td>
<td>0010</td>
</tr>
</tbody>
</table>

**Moore Machine State Transition Table**

- Observations:
  - Extensive use of Don’t Cares
  - Inputs used only in a small number of state
  - Some outputs are always asserted in a group
  - ROM-based implementations cannot take advantage of don’t cares
  - However, ROM-based implementation can skip state assignment step

**NOVA Machine Implementation**

- Standard Mealy Machine has asynchronous outputs
- These change in response to input changes, independent of clock
- Mealy Machine design can accommodate these changes
- One approach: non-overlapping clocks
Synchronous Mealy Machines

Case I: Synchronizers at Inputs and Outputs

CLK
A
A'
f
f'
A asserted in Cycle 0, f becomes asserted after 2 cycle delay.
This is clearly overkill

Synchronous Mealy Machines

Case II: Synchronizers on Inputs

CLK
A
A'
f
f'
A asserted in Cycle 0, f follows in next cycle
Same as using delayed signal (A') in Cycle 3

Synchronous Mealy Machines

Case III: Synchronized Outputs

CLK
A
f
f'
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>
  - Letter two already come from registers, and are synched to clock
  - Possible to load IR with new instruction in one state
  - Perform multi-way branch on opcode in next state
  - Best solution for Reset and Wait: synchronized inputs
    - Place 0 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 (Divide & Conquer)

Generation of Microoperations

| 0 | PC: Reset                  |
|   |                           |
| 1 | PC = 1: PC + T0           |
| 2 | PC = MAR: T0             |
| 3 | MAR = Memory Address Bus: T2 + T6 (LD + ST + ADD) |
| 4 | Memory Data Bus: MBR: T2 + T6 (LD + ADD) |
| 5 | MBR = Memory Data Bus: T6 + ST |
| 6 | MBR = AC: T7 + LD         |
| 7 | AC = MBR: T5 + ST         |
| 8 | AC = MBR: AC: T7 + ADD    |
| 9 | IRC1: MAR: T5 + (LD + ST + ADD) |
| 10| IRC2: MAR: T6 + BRN       |
| 11| 1: Read/Write: T2 + T6 (LD + ADD) |
| 12| 0: Read/Write: T6 + ST    |
| 13| 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

Hybrid Jump Counter
Multiple “Jump States” = function of current state + inputs

Jump Counters

Pure Jump Counter

Problem with Pure Jump Counter
Difficult to implement multi-way branches

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

Implementation Example
State assignment attempts to take advantage of sequential states
**Jump Counters**

*Implementation Example, Continued*

- CLR = CLRm + Reset
- Active Lo outputs:
  - hi input inverted at output
  - CNT is active fit on counter so invert MUX inputs!

**Jump Counter**

- CLR, CNT, LD
- Implemented via Mux Logic

**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
  - Example in hybrid approach: state + input subset form ROM address

- **Branch Sequences 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

**Jump Counters**

*Implementation Example, continued*

- Contents of Jump State ROM
  - **Address** | **Contents (Symbolic State)**
    - 00 | 0101 (LD0)
    - 01 | 1000 (ST10)
    - 10 | 1010 (ADD)
    - 11 | 1101 (BR0)

- CLR = Reset + s7 + s12 + s13 + (s9 + Wait)
- CNT = Wait•(s1 + s3) + Wait•(s2 + s6 + s9 + s11)

- **Microoperation implementation**
  - 0: PC = Reset
  - 1: PC = S0
  - 2: MAR = S0
  - 3: Memory Address Bus = Wait(S1 + s2 + s5 + s6 + s8 + s9 + s11 + s12)
  - 4: Memory Data Bus = MBR = Wait(S3 + s5 + s6 + s8 + s9)
  - 5: MBR = IR = Wait-S3
  - 6: MBR = AC = Wait-S7
  - 7: AC = MBR = IR15-IR14-S4
  - 8: AC = MBR = AC = Wait-S12
  - 9: IR = 13:0 MAR = IR15-IR14 + IR15-IR14 + IR15-IR14-S4
  - 10: Read/Write = Wait(S1 + s2 + s5 + s6 + s8 + s9 + s11 + s12)
  - 11: Read/Write = Wait(S3 + s5 + s6 + S8 + S9)
  - 12: Request = Wait(S1 + s2 + s5 + s6 + s8 + s9 + s11 + s12)

- Jump Counters: CNT, CLR, LS, etc are functions 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 32 bit wide: 7 bits wide

- **Branch Sequencers**

- **4 Way Branch Sequencer**
  - 64 Word ROM
  - Current State selects two inputs to form part of ROM address
    - These select one of four possible next state (and output sets)
    - Every state has exactly four possible next states
Horizontal Microprogramming

**Horizontal Branch Sequencer**

- Max bits
- 4 x 4 Next state bits
- 2 x Control operation bits
- 40 bits total

---

**Moore Processor ROM**

- Current State: 0000 - 0111
- Next States: 0000 - 1111
- Bits: 0 = RD, 1 = WR, 2 = ALU ADD, 3 = ALU ADD

---

**Partialy Encoded Control Outputs**

- AC = RD
- MBR = MBR
- ALU ADD = ALU ADD
- Data Bus = Data Bus

---

**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 format for control outputs and ‘branch jumps’
  - May require several microwords in a sequence to implement some function as single horizontal word
- In the extreme, very much like assembly language programming

---

**Register Transfer Format**

- Source, Destination, Operation
- 10 ROM Bits

---

**Register Transfer Format**

- Source, Destination, Operation
- 10 ROM Bits

---

**Register Transfer Format**

- Source, Destination, Operation
- 10 ROM Bits
Vertical Microprogramming

```
ROM ADDRESS  SYMBOLIC CONTENTS  BINARY CONTENTS
00000000  RES  RT  PC  MAR  PC+1  RT  00  00  0000  00
00000001  IFD  RT  MAR  M  Read  00  0000  00
00000011  IF1  RT  MAR  M  MBB  Read  00  0000  00
00000100  IFT  RT  MAR  M  MBB  Read  00  0000  00
00000000  IFT  RT  MAR  M  MBB  Read  00  0000  00
00000000  IFT  RT  MAR  M  MBB  Read  00  0000  00
00000000  IFT  RT  MAR  M  MBB  Read  00  0000  00
00000000  IFT  RT  MAR  M  MBB  Read  00  0000  00
00000000  IFT  RT  MAR  M  MBB  Read  00  0000  00
```

Vertical Microprogramming

```
ROM ADDRESS  SYMBOLIC CONTENTS  BINARY CONTENTS
00000000  STO  RT  AC  MBB  00  0000  00000000
00000000  RT  MAR  M  MBB  Write  00  0000  0000  00
00000000  ST  RT  MAR  M  MBB  Write  00  0000  0000  00
00000000  ST  RT  MAR  M  MBB  Write  00  0000  0000  00
00000000  ST  RT  MAR  M  MBB  Write  00  0000  0000  00
00000000  ST  RT  MAR  M  MBB  Write  00  0000  0000  00
00000000  ST  RT  MAR  M  MBB  Write  00  0000  0000  00
00000000  ST  RT  MAR  M  MBB  Write  00  0000  0000  00
00000000  ST  RT  MAR  M  MBB  Write  00  0000  0000  00
```

Vertical Programming

```
Controller Block Diagram
```

Controller Implementation Summary

```
- Control Unit Organization
  - Register transfer operation
  - Classical Moore and Mealy machines
  - Time State Approach
  - Jump Counter
  - Branch Sequencers
  - Horizontal and Vertical Microprogramming
```

Writeable Control Store

- Part of control store addresses map into RAM
  - Allows assembly language programmer to implement own instructions
  - Extends "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