Controller Implementation--Part I

- Alternative controller FSM implementation approaches based on:
  - Classical Moore and Mealy machines
  - Time state: Divide and Counter
  - Jump counters
  - Microprogramming (ROM) based approaches
    » branch sequencers
    » horizontal microcode
    » vertical microcode

Cascading Edge-triggered Flip-Flops

- Shift register
  - New value goes into first stage
  - While previous value of first stage goes into second stage
  - Consider setup/hold/propagation delays (prop must be > hold)

Clock Skew

- The problem
  - Correct behavior assumes next state of all storage elements determined by all storage elements at the same time
  - Difficult in high-performance systems because time for clock to arrive at flip-flop is comparable to delays through logic (and will soon become greater than logic delay)
  - Effect of skew on cascaded flip-flops:

Why Gating of Clocks is Bad!

- LD generated by FSM shortly after rising edge of CLK
- NASTY HACK: delay LD through negative edge triggered FF to ensure that it won’t change during next positive edge event

Do NOT Mess With Clock Signals!
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 Logic Block 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: PLA, FPGA, ROM-based (latter used in microprogrammed controllers)
- Could just as easily construct Moore and Mealy machines with these components

Moore Machine State Diagram

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
**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; Current State</th>
<th>Next State</th>
<th>Register Transfer Ops</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>LDL (0110)</td>
<td>LDL (0111)</td>
<td>MBR → Mem, Read,</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>LDL (0111)</td>
<td>LDL (0111)</td>
<td>Request, Mem → MBR</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>LDL (0111)</td>
<td>LDL (0110)</td>
<td>IR → MAR</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>LD2 (0000)</td>
<td>IF0 (0001)</td>
<td>MBR → AC</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>LD2 (0000)</td>
<td>ST1 (0010)</td>
<td>IR → MAR, AC → MBR</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>ST1 (0010)</td>
<td>IF0 (0001)</td>
<td>MBR → Mem, Write,</td>
</tr>
<tr>
<td>0</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>ABD (0111)</td>
<td>ABD (0110)</td>
<td>Request, MBR → Mem</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>ABD (0111)</td>
<td>ABD (0110)</td>
<td>IR → MAR</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>ABD2 (0111)</td>
<td>IF0 (0001)</td>
<td>MBR → AC + AC</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>ABD (0110)</td>
<td>ABD (0110)</td>
<td>MBR → Mem, Read,</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>ABD (0101)</td>
<td>ABD (0100)</td>
<td>Request, Mem → MBR,</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>X</td>
<td>X</td>
<td>BR0 (1100)</td>
<td>IF0 (0001)</td>
<td>IR → PC</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>BR0 (1101)</td>
<td>BR1 (1111)</td>
<td>MBR → MBR</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 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
- 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**

<table>
<thead>
<tr>
<th>A</th>
<th>A'</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>f'</td>
</tr>
</tbody>
</table>

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

Synchronous Mealy Machines
Case II: Synchronizers on Inputs

<table>
<thead>
<tr>
<th>Cycle</th>
<th>CLK</th>
<th>A</th>
<th>A'</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

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)

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

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

Hybrid jump counter:
Multiple "Jump States" — function of current state + inputs

Logic blocks implemented via discrete logic, PLAs, ROMs

Jump Counters

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

Load inputs are function of state and FSM inputs

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 (ADD0)</td>
</tr>
<tr>
<td>11</td>
<td>1101 (BR0)</td>
</tr>
</tbody>
</table>
Jump Counters

Implementation Example, continued

Implement CLR using active low

Act

Reset

Wait

S0

AC + MBR = Wait(S8 + S9)


O = 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

Controller Implementation Summary (Part I!)

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

The logic of MUX inputs!