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
- Runt pulse plays HAVOC with register internals!

NASTY HACK: delay LD through negative edge triggered FF to ensure that it won’t change during next positive edge event

- LDx gatedCLK
- CLK skew PLUS LD delayed by half clock cycle ...
- What is the effect on your register transfers?

Do NOT Mess With Clock Signals!
**Why Gating of Clocks is Bad!**

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 Implementation

Assume PLA implementation style
First idea: run ESPRESSO with noise state assignment

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 PLA/PLA

Moore Machine Diagram

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

Moore Machine State Table

<table>
<thead>
<tr>
<th>Reset/Write</th>
<th>RD/CE</th>
<th>PLA</th>
<th>Output</th>
<th>Next State</th>
<th>Register Transfer</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>MAR → Write, Read</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>MAR → Write, Read</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>MAR → Write, Read</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>MAR → Write, Read</td>
</tr>
</tbody>
</table>

Moore Machine State Transition Table

- Observations:
  - Extensive use of Don’t Cares
  - Inputs used only in a small number of states
  - e.g., AGB 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

Moore Machine Implementation

NOVA assignment does better

NOVA State Assignment SUMMARY

Best products = 18
Best, size = 416

Moore Machine Implementation

CS 158 - Fall 2005 - Lab #16: Control Implementation - 13

CS 158 - Fall 2005 - Lab #16: Control Implementation - 14

CS 158 - Fall 2005 - Lab #16: Control Implementation - 15

CS 158 - Fall 2005 - Lab #16: Control Implementation - 16

CS 158 - Fall 2005 - Lab #16: Control Implementation - 17

CS 158 - Fall 2005 - Lab #16: Control Implementation - 18
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

CLK
A
A'
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 I!

Case III: Synchronized Outputs

A asserted during Cycle 0, \( f' \) asserted in next cycle
Effect of \( f \) delayed one cycle

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

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 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 → MAR: T0
MAR → Memory Address Bus: T2 + T6 → (LD + ST + ADD)
Memory Data Bus: → MBR: T2 + T6 → (LD + ST + ADD)
MRR → Memory Data Bus: T6 → ST
MRR → IR: T4
MRR → AC: T7 • LD
AC → MRR: T5 • ST
AC + MRR → AC: T7 • ADD
IR(13:0) + MRR: T5 • (LD + ST + ADD)
AC: T6 • ST
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

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

Jump Counters
Hybrid Jump Counter
Load inputs are function of state and FSM inputs
**Jump Counters**

Implementation Example, Continued

\[ \text{CNT} = (a_0 + a_5 + a_8 + a_{10}) = \text{Wait} \times (a_1 + a_3) = \text{Wait} \times (a_2 + a_6 + a_9 + a_{11}) \]

\[ \text{CLR} = \text{Reset} \times a_7 + a_2 + a_6 + a_9 + a_{11} \]

\[ \text{CL} = \text{Reset} \times \text{S7} = \text{ST2} = \text{ST3} = (S9 + \text{Wait}) \]

\[ \text{LD} = s_4 \]

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 (LDS)</td>
</tr>
<tr>
<td>01</td>
<td>1000 (STD)</td>
</tr>
<tr>
<td>10</td>
<td>1010 (ADD)</td>
</tr>
<tr>
<td>11</td>
<td>1101 (BRN)</td>
</tr>
</tbody>
</table>

**Controller Implementation Summary (Part II)**

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