State Machine Timing

- **Retiming**
  - Slosh logic between registers to balance latencies and improve clock timings
  - Accelerate or retard cycle in which outputs are asserted

- **Pipelining**
  - Splitting computations into overlapped, smaller time steps

Recall: Synchronous Mealy Machine Discussion

- Placement of flipflops before and after the output logic changes the timing of when the output signals are asserted ...
Recall: Synchronous Mealy Machine with Synchronizers Following Outputs

Case III: Synchronized Outputs

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

A asserted during Cycle 0, f' asserted in next cycle

Effect of f delayed one cycle

---

Vending Machine State Machine

- **Moore machine**
  - outputs associated with state
- **Mealy machine**
  - outputs associated with transitions
State Machine Retiming

- Moore vs. (Async) Mealy Machine
  - Vending Machine Example

Open asserted only when in state 15

State Machine Retiming

- Retiming the Moore Machine: Faster generation of outputs
- Synchronizing the Mealy Machine: Add a FF, delaying the output
- These two implementations have identical timing behavior

Push the AND gate through the State FFs and synchronize with an output FF
Like computing open in the prior state and delaying it one state time
State Machine Retiming

- **Effect on timing of Open Signal (Moore Case)**

**Clk**

**State**

**Open**

**Retimed Open**

**Open Calculation**

*NOTE: overlaps with Next State calculation*

---

State Machine Retiming

- Timing behavior is the same, but are the implementations really identical?

**K-map for Open**

- FF input in retimed Moore implementation

**K-map for Open**

- FF input in synchronous Mealy implementation

*Only difference in don't care case of nickel and dime at the same time*
Pipelining Principle

- Pipelining review from CS61C:
  Analog to washing clothes:
  
  step 1: wash  (20 minutes)
  step 2: dry   (20 minutes)
  step 3: fold  (20 minutes)
  
  60 minutes × 4 loads ⇒ 4 hours

<table>
<thead>
<tr>
<th>wash</th>
<th>load1</th>
<th>load2</th>
<th>load3</th>
<th>load4</th>
</tr>
</thead>
<tbody>
<tr>
<td>dry</td>
<td>load1</td>
<td>load2</td>
<td>load3</td>
<td>load4</td>
</tr>
<tr>
<td>fold</td>
<td>load1</td>
<td>load2</td>
<td>load3</td>
<td>load4</td>
</tr>
</tbody>
</table>

   20 min

overlapped ⇒ 2 hours

- Increase number of loads, average time per load approaches 20 minutes

- Latency (time from start to end) for one load = 60 min
- Throughput = 3 loads/hour

- Pipelined throughput = # of pipe stages x un-pipelined throughput.
Pipelining

- General principle:
  \[ T' = 4 \text{ ns} + 1 \text{ ns} + 4 \text{ ns} + 1 \text{ ns} = 10 \text{ ns} \]
  \[ F = 1/(4 \text{ ns} + 1 \text{ ns}) = 200 \text{ MHz} \]
  Assume \( T = 8 \text{ ns} \)
  \( T_{\text{FF}}(\text{setup } + \text{clk} \rightarrow q) = 1 \text{ ns} \)
  \( F = 1/9 \text{ ns} = 111 \text{ MHz} \)

- Cut the CL block into pieces (stages) and separate with registers:
  \[ T' = 4 \text{ ns} + 1 \text{ ns} + 4 \text{ ns} + 1 \text{ ns} = 10 \text{ ns} \]
  \[ F = 1/(4 \text{ ns} + 1 \text{ ns}) = 200 \text{ MHz} \]
  Assume \( T_1 = T_2 = 4 \text{ ns} \)

- CL block produces a new result every 5 ns instead of every 9 ns

Limits on Pipelining

- Without FF overhead, throughput improvement proportional to # of stages
  - After many stages are added, FF overhead begins to dominate:
    \[ FF \text{ "overhead" is the setup and clk to Q times.} \]

- Other limiters to effective pipelining:
  - Clock skew contributes to clock overhead
  - Unequal stages
  - FFs dominate cost
  - Clock distribution power consumption
  - Feedback (dependencies between loop iterations)
**Pipelining Example**

- \( F(x) = y = a \cdot x^2 + b \cdot x + c \)

\[ \begin{align*}
  x & \rightarrow F(x) & \rightarrow y \\
\end{align*} \]

- \( x \) and \( y \) are assumed to be "streams"
- Divide into 3 (nearly) equal stages.
- Insert pipeline registers at dashed lines.
- Can we pipeline basic operators?

**Example: Pipelined Adder**

- Possible, but usually not done …
  (arithmetic units can often be made sufficiently fast without internal pipelining)
State Machine Retiming Summary

- **Retiming**
  - Vending Machine Example
    - Very simple output function in this particular case
  - But if output takes a long time to compute vs. the next state computation time -- can use retiming to "balance" these calculations and reduce the cycle time

- **Pipelining**
  - Introduce registers to split computation to reduce cycle time and allow parallel computation
  - Trade latency (number of stage delays) for cycle time reduction