Synchronous Logic

1) Sequential Logic
2) Synchronous Design
3) Synchronous Timing Analysis
4) Single Clock Design
5) Finite State Machines
6) Mealy and Moore
7) State Transition Diagrams
Road Traveled So Far...

FETs & voltages

Combinational contract:
- Voltage-based "bits"
- 1-bit per wire
- Generate quality outputs, tolerate inferior inputs
- Combinational contract
- Complete in/out/timing spec

Logic gates

Acyclic connections
Composable blocks
Design:
- truth tables
- sum-of-products
- muxes
- ROMs

Combinational logic circuits

ALU, Mult, Etc.

Sequential Logic

Storage & state
Dynamic discipline
Finite-state machines
Throughput & latency
Pipelining

Our motto: Sweat the details once, and then put a box around it!
What if you were given the following system design specification?

When the button is pushed:
1) Turn on the light if it is off
2) Turn off the light if it is on

The light should change state within a second of the button press

What makes this system so different from those we’ve discussed before?

1. "State" - i.e. the circuit has memory
2. The output was changed by an input "event" (pushing a button) rather than an input "value"
"Sequential" = Stateful

Plan: Build a Sequential Circuit with stored digital STATE -

- MEMORY stores CURRENT state
- Combinational Logic computes
  - the NEXT state (Based on inputs & current state)
  - the OUTPUTs (Based on inputs and/or current state)
  - State changes on LOAD control input

Didn't we develop some memory devices last time?
"Synchronous" Single-Clock Logic

Questions for register-based designs:

- How much time for useful work (i.e. for combinational logic delay)?

- Does it help to guarantee a minimum $t_{CD}$? How 'bout designing registers so that $t_{CD,reg} \geq t_{HOLD,reg}$?

- What happens if CLK signal doesn't arrive at the two registers at exactly the same time (a phenomenon known as "clock skew")?

\[
\begin{align*}
 t_1 &= t_{CD,reg1} + t_{CD,L} \geq t_{HOLD,reg2} \\
 t_2 &= t_{PD,reg1} + t_{PD,L} \leq t_{CLK} - t_{SETUP,reg2}
\end{align*}
\]

Minimum Clock Period : $t_{CLK} \geq t_{PD,reg1} + t_{PD,L} + t_{SETUP,reg2}$
Example: Synchronous Timing

Questions:
1. \( t_{\text{CD}} \) for the ROM?
   \[ t_{\text{CD,REG}} + t_{\text{CD,ROM}} > t_{\text{H,REG}} \]
   \[ 1 \text{ ns} + t_{\text{CD,ROM}} > 2 \text{ nS} \]
   \[ t_{\text{CD,ROM}} > 1 \text{ nS} \]

2. Min. clock period?
   \[ t_{\text{CLK}} > t_{\text{PD,REG}} + t_{\text{PD,ROM}} + t_{\text{S,REG}} \]
   \[ t_{\text{CLK}} > 3 \text{ ns} + 5 \text{ ns} + 2 \text{ nS} \]
   \[ t_{\text{CLK}} > 10 \text{ nS} \]

3. Constraints on inputs?
   “start”, “0”, and “1” must be valid
   \[ t_{\text{PD,ROM}} + t_{\text{S,REG}} = 5 + 2 = 7 \text{ ns} \]
   before the clock and held
   \[ t_{\text{H,REG}} - t_{\text{CD,ROM}} = 2 - 1 = 1 \text{ ns} \]
   after it.
Synchronous Single-Clock Design

Sequential ≠ Synchronous

However, Synchronous = A recipe for robust sequential circuits:

- No combinational cycles (other than those already inside the registers)
- Only cares about values of combinational circuits just before rising edge of clock
- Clock period greater than every combinational delay
- Changes state after all logic transitions have stopped!
Designing Sequential Logic

Sequential logic is used when the solution to some design problem involves a sequence of steps:

How to open digital combination lock w/ 3 buttons ("start", "0" and "1"):

- **Step 1:** press "start" button
- **Step 2:** press "0" button
- **Step 3:** press "1" button
- **Step 4:** press "1" button
- **Step 5:** press "0" button

Information remembered between steps is called **state**. Might be just what step we're on, or might include results from earlier steps we'll need to complete a later step.
# Implementing a “State Machine”

This flavor of "truth-table" is called a "state-transition table"

<table>
<thead>
<tr>
<th>Current State “start” “1” “0”</th>
<th>Next State</th>
<th>unlock</th>
</tr>
</thead>
<tbody>
<tr>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>start 000 0 0 1</td>
<td>start 000 0 0 0</td>
<td></td>
</tr>
<tr>
<td>start 000 0 1 0</td>
<td>digit1 001 0 1 0</td>
<td></td>
</tr>
<tr>
<td>start 000 0 0 0</td>
<td>start 000 0 0 0</td>
<td></td>
</tr>
<tr>
<td>digit1 001 0 1 0</td>
<td>digit2 001 0 0 0</td>
<td></td>
</tr>
<tr>
<td>digit1 001 0 0 1</td>
<td>error 001 0 0 0</td>
<td></td>
</tr>
<tr>
<td>digit1 001 0 0 0</td>
<td>digit1 001 0 0 0</td>
<td></td>
</tr>
<tr>
<td>digit2 010 0 1 0</td>
<td>digit3 011 0 0 1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>unlock 100 0 1 0</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>error 100 0 1 0</td>
<td></td>
</tr>
<tr>
<td>unlock 100 0 1 0</td>
<td>error 100 0 1 0</td>
<td></td>
</tr>
<tr>
<td>unlock 100 0 0 0</td>
<td>unlock 100 0 0 0</td>
<td></td>
</tr>
<tr>
<td>error 101 0 0 0</td>
<td>error 101 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

6 different states → encode using 3 bits
Now, we do it with hardware!

6 inputs $\rightarrow 2^6$ locations
Each location supplies 4 bits

“start” button
“0” button
“1” button

ROM
64x4

Current state

Next state

3 inputs $\rightarrow 2^3$ locations
Each location supplies 4 bits

Q D

Trigger update periodically (“clock”)
A Finite State Machine has:

- k States $S_1, S_2, \ldots, S_k$ (one is the “initial” state)
- m inputs $I_1, I_2, \ldots, I_m$
- n outputs $O_1, O_2, \ldots, O_n$
- Transition Rules, $S'(S_i, I_1, I_2, \ldots, I_m)$ for each state and input combination
- Output Rules, $O(S_i)$ for each state
**Discrete State, Discrete Time**

Two design choices:
1. Outputs *only* depend on state (Moore)
2. Outputs depend on inputs + state (Mealy)

While a ROM is shown here in the feedback path any form of combinational logic can be used to construct a state machine.

$s$ state bits → $2^s$ possible states

Clock

State

Next

Clock Period

1

Clock Period

2

Clock Period

3

Clock Period

4

Clock Period

5
A state transition diagram is an abstract "graph" representation of a "state transition table", where each state is represented as a node and each transition is represented as an arc. It represents the machine's behavior not its implementation.

Heavy circle means **INITIAL** state

NAME of state

OUTPUT when in this state (Moore)

INPUT or INPUTs causing transition

★ = no buttons pressed
Arcs leaving a state must be:

1. **mutually exclusive**
   - can only have one choice for any given input value

2. **collectively exhaustive**
   - every state must specify what happens for each possible input combination. "Nothing happens" means arc back to itself.

**Example State Diagrams**

**MOORE Machine:**
Outputs on States

**MEALY Machine:**
Outputs on Transitions
Next Time

Counting state machines