Synchronous Logic

1) Sequential Logic
2) Synchronous Design
3) Synchronous Timing Analysis
4) Single Clock Design
5) Finite State Machines
6) Turing Machines
7) What it means to be “Computable”
Road Traveled So Far…

Fets & voltages

Logic gates

Combinational logic circuits

Sequential Logic

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

Acyclic connections

Composable blocks

Design:
- truth tables
- sum-of-products
- muxes, ROMs

Storage & state

Dynamic discipline

Finite-state machines

Throughput & latency

Pipelining

Our motto: Sweat the details once, and then put a box around it!
Something We Can’t Build (Yet)

What if you were given the following 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 circuit so different from those we’ve discussed before?

1. “State” – i.e. the circuit has memory
2. The output was changed by a 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 (from input, current state)
  - the OUTPUTs (from input, current state)
- State changes on LOAD control input

Didn’t we develop some memory devices last time?
Review of Flip Flop Timing

- $t_{PD}$: maximum propagation delay, CLK $\rightarrow$ Q
  - How LONG after clock before outputs (Q) are valid

- $t_{CD}$: minimum contamination delay, CLK $\rightarrow$ Q
  - How SOON after clock outputs (Q) go invalid

- $t_{SETUP}$: setup time
  - How LONG data (D) input must be stable before clock’s rising edge

- $t_{HOLD}$: hold time
  - How LONG data (D) inputs must be held after clock’s rising edge
Synchronous Timing Analysis

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 $t_{CD,reg} > t_{HOLD,reg}$?
- What happens if CLK signal doesn’t arrive at the two registers at exactly the same time (a phenomenon known as “clock skew”)?

$$t_1 = t_{CD,reg1} + t_{CD,L} > t_{HOLD,reg2}$$

$$t_2 = t_{PD,reg1} + t_{PD,L} < t_{CLK} - t_{SETUP,reg2}$$

Minimum Clock Period: $t_{CLK} > t_{PD,reg1} + t_{PD,L} + t_{SETUP,reg2}$
Example: Flip Flop Timing

Questions:

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

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

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

Sequential ≠ Synchronous

However, Synchronous = A recipe for robust sequential circuits:

• No combinational cycles
  (other than those already built into 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"

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

6 different states → encode using 3 bits
Now Do It With Hardware!

ROM
64x4

“start” button
“0” button
“1” button

Current state

Next state

6 inputs → 2^6 locations
each location supplies 4 bits

3 inputs

3 inputs

Q
D

Trigger update periodically ("clock")

Unlock
Abstraction du jour: Finite State Machines

A FINITE STATE MACHINE has

- $k$ STATES $S_1 \ldots S_k$ (one is "initial" state)
- $m$ INPUTS $I_1 \ldots I_m$
- $n$ OUTPUTS $O_1 \ldots O_n$
- Transition Rules $S'(S, i)$ for each state $S$ and input $i$
- Output Rules $Out(S)$ for each state $S$
Discrete State, Time

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

$s$ state bits $\rightarrow 2^s$ possible states

Clock
STATE
NEXT

Clock Period
1
2
3
4
5
A state transition diagram is an abstract “graph” representation of a state machine, 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 causing transition

* = no buttons pressed
Valid State Diagrams

Arrows 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.

**MOORE Machine:** Outputs on States

**MEALY Machine:** Outputs on Transitions
Let's Play State Machine

Let's emulate the behavior specified by the state machine shown below when processing the following string from LSB to MSB.

\[
39_{10} = 0100111_2
\]

It looks to me like this machine outputs a 1 whenever the bit sequence that it has seen thus far is a multiple of 3.
CONVENIENT NOTATION:
When a transition is made on the next input regardless of its value the arc can be labeled with an X or -

AMBIGUOUS TRANSITIONS (Mutual Exclusive property violated):
For each input there can only be one arc leaving a state

UNSPECIFIED TRANSITIONS (Collectively Exhaustive property violated):
There must be an arc leaving a state for all valid inputs (It can, however, loop back to the same state)
FSM Party Games

1. What can you say about the number of states?

   States ≤ $2^k$

2. Same question:

   States ≤ $m \times n$

3. Here's an FSM. Can you discover its rules?
What's My Transition Diagram?

0 = OFF, 1 = ON?

"1111" = Surprise!

• If you know NOTHING about the FSM, you’re never sure!

• If you have a BOUND on the number of states, you can discover its behavior:

  k-state FSM: Every (reachable) state can be reached in < k steps.

BUT ... states may be equivalent!
FSM Equivalence

ARE THEY DIFFERENT?
NOT in any practical sense! They are EXTERNALLY INDISTINGUISHABLE, hence interchangeable.

FSMs are EQUIVALENT iff every input sequence yields identical output sequences.

ENGINEERING GOAL:
• HAVE an FSM which works...
• WANT simplest (ergo cheapest) equivalent FSM.
Housekeeping issues...

1. Initialization? Clear the memory?

2. Unused state encodings?
   - waste ROM (use PLA or gates)
   - meaning?

3. Synchronizing input changes with state update?

4. Choosing encoding for state?

That symbol is starting to register
2-Flavors of Processing Elements

Combinational Logic:
- Table look-up, ROM

Finite State Machines:
- ROM with Feedback Memory

Thus far, we know of nothing more powerful than an FSM

Fundamentally, everything that we’ve learned so far can be done with a ROM and registers.
FSMs as Programmable Machines

ROM-based FSM sketch:

Given i, s, and o, we need a ROM organized as:

\[ 2^{i+s} \text{ words } \times (o+s) \text{ bits} \]

So how many possible i-input, o-output, FSMs with s-state bits exist?

\[ \frac{2^{(o+s)}2^{i+s}}{2^{i+s}} \] (some may be equivalent)

An FSM’s behavior is completely determined by its ROM contents.

All possible settings of the ROM’s contents to: 1 or 0
FSM Enumeration

GOAL: List all possible FSMs in some canonical order.

- INFINITE list, but
- Every FSM has an entry in and an associated index.

Every possible FSM can be associated with a unique number. This requires a few wasteful simplifications. First, given an i-input, s-state-bit, and o-output FSM, we'll replace it with its equivalent n-input, n-state-bit and n-output FSM, where n is the greatest of i, s, and o. We can always ignore the extra input-bits, and set the extra output bits to 0. This allows us to discuss the i\textsuperscript{th} FSM.
Some Perennial Favorites...

- $\text{FSM}_{837}$: modulo 3 counter
- $\text{FSM}_{1077}$: 4-bit counter
- $\text{FSM}_{1537}$: Combination lock
- $\text{FSM}_{89143}$: Cheap digital watch
- $\text{FSM}_{22698469884}$: Intel Pentium CPU – rev 1
- $\text{FSM}_{784362783}$: Intel Pentium CPU – rev 2
- $\text{FSM}_{784363783}$: Intel Pentium II CPU
Are FSMs the Ultimate Computation Device?

Nope!
There exist many simple problems that cannot be computed by FSMs. For instance:

Checking for balanced parenthesis

(()(()(()))) - Okay
(()()) - No good!

PROBLEM: Requires ARBITRARILY many states, depending on input. Must "COUNT" unmatched LEFT parens. But, an FSM can only keep track of a finite number of objects.

Is there a machine that can solve this problem?
Unbounded-Space Computation

DURING 1920s & 1930s, much of the “science” part of computer science was being developed (long before actual electronic computers existed). Many different “Models of Computation” were proposed, and the classes of “functions” which could be computed by each were analyzed.

One of these models was the TURING MACHINE, named after Alan Turing.

A Turing Machine is just an FSM which receives its inputs and writes outputs onto an infinite tape...

This simple addition solves "FINITE" problem of FSMs.
A Turing Machine Example

Turing Machine Specification

• Doubly-infinite tape
• Discrete symbol positions
• Finite alphabet – say \{0, 1\}
• Control FSM

  INPUTS:
  Current symbol

  OUTPUTS:
  write 0/1
  move Left/Right

• Initial Starting State \{S0\}
• Halt State \{Halt\}

A Turing machine, like an FSM, can be specified with a truth table. The following Turing Machine implements a unary (base 1) incrementer.

<table>
<thead>
<tr>
<th>Current State</th>
<th>Tape Input</th>
<th>Write Tape</th>
<th>Move</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>1</td>
<td>1</td>
<td>R</td>
<td>S0</td>
</tr>
<tr>
<td>S0</td>
<td>0</td>
<td>1</td>
<td>L</td>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
<td>1</td>
<td>1</td>
<td>L</td>
<td>S1</td>
</tr>
<tr>
<td>S1</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>Halt</td>
</tr>
</tbody>
</table>

0 0 0 0 1 1 1 1 0
Turing Machine Tapes as Integers

**Canonical names for bounded tape configurations:**

```
  b8  b6  b4  b2  b0  b1  b3  b5  b7
  0  0  1  0  0  1  1  0  0
```

**Look, it's just FSM \( i \) operating on tape \( j \)**
TMs as Integer Functions

Turing Machine $T_i$ operating on Tape $x$, where $x = \ldots b_8 b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$

$y = T_i [x]$

- $x$: input tape configuration
- $y$: output tape when TM \textit{halts}

I wonder if a TM can compute \textit{EVERY} integer function...
Alternative Models of Computation

Turing Machines [Turing]

FSM

Hardware

head

Recursive Functions [Kleene]

F(0,x) ≡ x

F(1+y,x) ≡ 1+F(x,y)

(define (fact n)
  (... (fact (- n 1)) ...) )

Kleene

Production Systems [Post, Markov]

α → β

IF pulse=0 THEN patient=dead

Post

Church

Lambda calculus [Church, Curry, Rosser...]

λx.λy.xxy

(lambda(x)(lambda(y)(x (x y))))

Church
Here’s a TM that computes SQUARE ROOT!
And the Battles Raged

Here's a Lambda Expression that does the same thing...

\((\lambda(x) \ldots)\)

... and here's one that computes the \(n^{th}\) root for ANY \(n\)!

\((\lambda(x\ n) \ldots)\)
Fundamental Result: Computable Functions

Each model is capable of computing exactly the same set of integer functions!

Proof Technique: Constructions that translate between models

BIG IDEA: Computability, independent of computation scheme chosen

Church's Thesis:
Every discrete function computable by ANY realizable machine is computable by some Turing machine.

Does this mean that we know of no computer that is more "powerful" than a Turing machine?
Computable Functions

\[
f(x) \text{ computable } \iff \text{ for some } k, \text{ all } x:\]
\[
f(x) = T_k[x] \equiv f_k(x)
\]

Representation tricks: to compute \( f_k(x,y) \)

\(<x,y> \equiv \text{integer whose even bits come from } x, \text{ and whose odd bits come from } y;\)

whence

\[
f_k(x, y) \equiv T_k[<x, y>] 
\]

\( f_{12345}(x,y) = x \times y \)

\( f_{23456}(x) = 1 \text{ iff } x \text{ is prime, else } 0 \)
### Enumeration of Computable functions

Conceptual table of TM behaviors...

**VERTICAL AXIS:** Enumeration of TMs.

**HORIZONTAL AXIS:** Enumeration of input tapes.

(j, k) entry = result of TM_k[j] -- integer, or * if never halts.

| j | f_0 | f_1 | f_2 | ... | f_k | ...
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>*1</td>
<td>*1</td>
<td>*0</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>21</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
| ... | ... | ... | ... | ... | ... | ...

The Halting Problem: Given j, k: Does TM_k Halt with input j?

Is every Integer Function that I can precisely specify computable?
The Halting Problem

The Halting Function: $T_H[k, j] = 1$ iff $T_M[k][j]$ halts, else $0$

Can a Turing machine compute this function?

Suppose, for a moment, $T_H$ exists:

1 iff $T_x[y]$ HALTS
0 otherwise

Then we can build a $T_{Nasty}$:

$T_{Nasty}[k]$  LOOP if $T_k[k] = 1$ (halts)
  HALT if $T_k[k] = 0$ (loops)

If $T_H$ is computable then so is $T_{Nasty}$.
What does $T_{Nasty}[Nasty]$ do?

Answer:

$T_{Nasty}[Nasty]$ loops if $T_{Nasty}[Nasty]$ halts

$T_{Nasty}[Nasty]$ halts if $T_{Nasty}[Nasty]$ loops

That's a contradiction.

Thus, $T_H$ is uncomputable by a Turing Machine!

Net Result: There are some questions that Turing Machines simply cannot answer. Since, we know of no better model of computation than a Turing machine, this implies that there are some questions that defy computation.
Reality: Limits of Turing Machines

A Turing machine is formal abstraction that addresses

- Fundamental Limits of Computability –
  What is means to compute.
  The existence of incomputable functions.

- We know of no machine more powerful than a Turing machine in terms of the functions that it can compute.

But they ignore

- Practical coding of programs
- Performance
- Implementability
- Programmability

... these latter issues are the primary focus of contemporary computer science (Remainder of Comp 411)