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

Current state

Next state

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 $\rightarrow 2^s$ possible states
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
Example State Diagrams

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.
Next Time

Counting state machines
FSMs and Turing Machines

- Ways we know to compute
  - Truth-tables = combinational logic
  - State-transition tables = sequential logic
- Enumerating FSMs
- An even more powerful model: a "Turing Machine"
- What does it mean to compute?
- What can’t be computed
- Universal TMs = programmable TM
**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.

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

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>Next</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>T=0</td>
<td>S0</td>
<td>1</td>
<td>S1</td>
</tr>
<tr>
<td>T=1</td>
<td>S1</td>
<td>1</td>
<td>S0</td>
</tr>
<tr>
<td>T=2</td>
<td>S0</td>
<td>1</td>
<td>S1</td>
</tr>
<tr>
<td>T=3</td>
<td>S1</td>
<td>0</td>
<td>S2</td>
</tr>
<tr>
<td>T=4</td>
<td>S2</td>
<td>0</td>
<td>S1</td>
</tr>
<tr>
<td>T=5</td>
<td>S1</td>
<td>1</td>
<td>S0</td>
</tr>
<tr>
<td>T=6</td>
<td>S0</td>
<td>0</td>
<td>S0</td>
</tr>
</tbody>
</table>

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. (Wow, and FSM can divide by 3!)
1. What can you say about the number of states?

   States $\leq 2^k$

2. Same question:

   States $\leq m \times n$

3. Here's an FSM. Can you discover its rules?

   You Win!
What’s My Transition Diagram?

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 \(< 2^i \times k\) steps.

But ... states may be equivalent!
FSM EQUVALENCE

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?
   - wastes ROM (use gates)
   - meaning?

3. Synchronizing input changes with state update?

4. Choosing encodings for each state?

That symbol is starting to register.

inputs → ROM or gates → outputs

STATE → NEXT

That symbol is starting to register.
2-TYPES OF PROCESSING ELEMENTS

Combinational Logic:
Table look-up, ROM

Recall that there are precisely $2^{2^i}$, i-input combinational functions. A single ROM can store 'o' of them.

Finite State Machines:
ROM with State 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:

$$2^{i+s} \text{ words x (o+s) bits}$$

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

All possible settings of the ROM's contents to: 1 or 0

$$2^{(o+s)2^{i+s}}$$

(some may be equivalent)

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

Recall how we were able to "enumerate" or "name" every 2-input gate?
Can we do the same for FSMs?

### Table Example

<table>
<thead>
<tr>
<th>$i^s$</th>
<th>$s$</th>
<th>$s_{N+1}$</th>
<th>$o$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0...00</td>
<td>0...00</td>
<td>10110</td>
<td>011</td>
</tr>
<tr>
<td>0...01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$$2^{(1+1)4} = 2^8 = 256$$
**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 $i^{th}$ FSM.

<table>
<thead>
<tr>
<th>i</th>
<th>s</th>
<th>o</th>
<th>FSM#</th>
<th>Truth Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>00000000</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>00000001</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>256</td>
<td>11111111</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>257</td>
<td>000000...00000</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>258</td>
<td>000000...00001</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>18,446,744,073,709,551,872</td>
<td>000000...00000</td>
</tr>
<tr>
<td>3.9402 x 10^{115}</td>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3.9402 x 10^{115}</td>
<td>000000...00000</td>
</tr>
</tbody>
</table>

These are the FSMs with 1 input and 1 output and 1 state bit. They have 8-bits in their ROM.
**Some Favorites**

- **FSM**\(_{837}\) modulo 3 state machine
- **FSM**\(_{1077}\) 4-bit counter
- **FSM**\(_{1537}\) Combination lock
- **FSM**\(_{89143}\) Cheap digital watch
- **FSM**\(_{22698469884}\) MIPs processor
- **FSM**\(_{23892749274}\) ARM7 processor
- **FSM**\(_{78436378389}\) Intel I-7 processor (Skylake)
- **FSM**\(_{78436378390}\) Intel I-7 processor (Kaby lake)
Can FSMs Compute Every Binary Function?

Nope!

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

Checking for balanced parentheses

(((())()))  - 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 "bounded" number of events. (Bounded by its number of states)

Is there another form of logic that can solve this problem?
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" that each could compute were analyzed.

One of these models was the "TURING MACHINE", named after Alan Turing (1912-1954).

A Turing Machine is just an FSM which receives its inputs and writes outputs onto an "infinite tape". This simple addition overcomes the FSM’s limitation that it can only keep track of a "bounded number of events".
A Turing Machine Example

Turing Machine Specification
- Infinite tape
- Discrete symbol positions
- Finite alphabet - say \{0, 1\}
- Control FSM

INPUTS:
- Current symbol on tape

OUTPUTS:
- write 0/1
- move tape Left or Right
- Initial Starting State \{S0\}
- Halt State \{Halt\}

A Turing machine, like an FSM, can be specified via a state-transition table. The following Turing Machine implements a unary (base 1) counter.

<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|1|0…
### Turing Machine Tapes as Integers

**Canonical names for bounded tape configurations:**

<table>
<thead>
<tr>
<th>b_8</th>
<th>b_6</th>
<th>b_4</th>
<th>b_2</th>
<th>b_0</th>
<th>b_1</th>
<th>b_3</th>
<th>b_5</th>
<th>b_7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Look, it's just FSM \( i \) operating on tape \( j \)

**Note:** The FSM part of a Turing Machine is just one of the FSMs in our enumeration. The tape can also be represented as an integer, but this is trickier. It is natural to represent it as a binary fraction, with a binary point just to the left of the starting position. If the binary number is rational, we can alternate bits from each side of the binary point until all that is left is zeros, then we have an integer.
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 halts

I wonder if a TM can compute EVERY integer function…
**Alternative Models of Computation**

Turing Machines [Turing]

```
1 0 0 1 0 0 1 1 0 0

FSM_i
```

Hardware head

Turing

Recursive Functions [Kleene]

\[
\begin{align*}
F(0,x) &= x \\
F(y,0) &= y \\
F(y,x) &= x + y + F(y-1,x-1)
\end{align*}
\]

(\text{define } (\text{fact } n))

(\text{... } (\text{fact } (- n 1)) \text{ ...})

Kleene (1909-1994)

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

\[
\lambda x. \lambda y. x x y
\]

\[
\text{(lambda}(x)(\text{lambda}(y)(x (x y))))
\]

Church (1903-1995)

Turing’s PhD Advisor

Production Systems [Post, Markov]

\[
\begin{align*}
0 &\rightarrow [] \\
0 &\rightarrow [0] \\
0 &\rightarrow 0 \\
0 &\rightarrow i_j
\end{align*}
\]

Post (1897-1954)
The 1st Computer Industry Shakeout

Here's a TM that computes SQUARE ROOT!

| 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |

FSM
AND THE BATTLES RAGED

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

(\lambda (x) \ldots \ldots)

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

(\lambda (x \ n) \ldots \ldots)
A FUNDAMENTAL RESULT

Turing’s amazing proof: Each model is capable of computing exactly the same set of integer functions! None is more powerful than the others.

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.

This means that we know of no mechanisms (including computers) that are more “powerful” than a Turing Machine, in terms of the functions they can compute.
Computable Functions

The “input” to our computable function will be given on the initial tape, and the “output” will be the contents of the tape when the TM halts.

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

\[ f(x) = T_k[x] \equiv f_k(x) \]

Representation tricks: to compute \( f_k(x,y) \) (2 inputs)

\(<x,y> \equiv \text{integer whose even bits come from } x, \]

\text{and whose odd bits come from } y; \text{ whence}

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

\[ f_{12345}(x,y) = x \ast y \]

\[ f_{23456}(x) = 1 \text{ iff } x \text{ is prime, else } 0 \]
TM5, like programs, can misbehave

It is possible that a given Turing Machine may not produce a result for a given input tape. And it may do so by entering an infinite loop!

Consider the given TM.

It scans a tape looking for the first non-zero cell to the right.

What does it do when given a tape that has no 1's to its left?

We say this TM does not halt for that input!

<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>L</td>
<td>Halt S0</td>
</tr>
<tr>
<td>S0</td>
<td>0</td>
<td>0</td>
<td>R</td>
<td>S0</td>
</tr>
</tbody>
</table>

\[
tape_{256} = \cdots 0|0|0|0|0|0|0|1|0|0 \cdots \\
\]

\[
tape_{8} = \cdots 0|1|0|0|0|0|0|0|0|0 \cdots \\
\]
### 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 it never halts.

| Turing Machine Tapes | f\(_i(0)\) | f\(_i(1)\) | f\(_i(2)\) | ... | f\(_i(j)\) | ...
|----------------------|------------|------------|------------|-----|------------|-----
| f\(_0\)              | X1         | X1         | X0         | ... | ...        |     |
| f\(_1\)              | X1         | X0         | 000        | ... | ...        |     |
| ...                  | ...        | ...        | ...        | ... | ...        |     |
| f\(_k\)              | ...        | ...        | ...        | ... | f\(_k(j)\) |     |
| ...                  | ...        | ...        | ...        | ... | ...        |     |

**The Halting Problem:** Given j, k: Does TM\(_k\) Halt with input j?

Every computable function is in this table, since everything that we know how to compute can be computed by a TM.

Do there exist well-specified integer functions that a TM can't compute?
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 \( T_k[j] \) 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 \( T_{Nasty} \)

We only run \( T_h \) on a subset of inputs, those on the diagonal of the table given on the previous slide.
**What does $T_{\text{nasty}}$ do?**

Answer:

$T_{\text{nasty}}$ loops if $T_{\text{nasty}}$ halts
$T_{\text{nasty}}$ halts if $T_{\text{nasty}}$ loops

That's a contradiction.

Thus, $T_H$ is not computable by a Turing Machine!

**Net Result**: There are some integer functions 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 well-specified problems that defy computation.
Limits of Turing Machines

A Turing machine is a formal abstraction that addresses

- Fundamental Limits of Computability -
  What is means to compute.
  The existence of uncomputable 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)
Recall Church’s thesis:

“Any discrete function computable by ANY realizable machine is computable by some Turing Machine”

We’ve defined what it means to COMPUTE (whatever a TM can compute), but, a Turing machine is nothing more that an FSM that receives inputs from, and outputs onto, an infinite tape.

So far, we’ve been designing a new FSM for each new Turing machine that we encounter.

Wouldn’t it be nice if we could design a more general-purpose Turing machine?
Programs as Data

What if we encoded the description of the FSM on our tape, and then wrote a general purpose FSM to read the tape and EMULATE the behavior of the encoded machine? We could just store the state-transition table for our TM on the tape and then design a new TM that makes reference to it as often as it likes. It seems possible that such a machine could be built.

"It is possible to invent a single machine which can be used to compute any computable sequence. If this machine U is supplied with a tape on the beginning of which is written the S.D ["standard description" of an action table] of some computing machine M, then U will compute the same sequence as M.”
**Fundamental Result: Universality**

Define "Universal Function": $u(x,y) = T_x(y)$ for every $x, y$ ...

**Surprise!** $u(x,y)$ IS COMPUTABLE, hence $u(x,y) = T_u(<x,y>)$ for some $u$.

**Universal Turing Machine (UTM):**

$T_U[<y, z>] = T_y[z]$  

- Tape = "data"  
- TM = "program"  
- "interpreter"

**PARADIGM** for General-Purpose Computer!

---

**INFINITELY many UTMs ...**
Any one of them can evaluate any computable function by simulating/emulating/interpreting the actions of Turing machine given to it as an input.

**UNIVERSALITY:**
Basic requirement for a general purpose computer.
Demonstrating Universality

Suppose you’ve designed Turing Machine $T_K$ and want to show that it is universal.

**Approach:**

1. Find some known universal machine, say $T_U$.
2. Devise a program, $P$, to simulate $T_U$ on $T_K$: $T_K[<P,x>] = T_U[x]$ for all $x$.
3. Since $T_U[<y,z>] = T_y[z]$, it follows that, for all $y$ and $z$,

\[
T_K[<P,<y,z>>] = T_U[<y,z>] = T_y[z]
\]

**Conclusion:** Armed with program $P$, machine $T_K$ can mimic the behavior of an arbitrary machine $T_y$ operating on an arbitrary input tape $z$.

**Hence $T_K$** can compute any function that can be computed by any Turing Machine.
Next Time

Enough theory already, let's build something!

Build SOMETHING

Build SOMETHING GOOD

Keep Calm and Build Something