Finite-State Machines as Hardware
Implementing Finite-State Machines using Logic+Memory

- When we implemented logic functions, we used only gates and no memory; those are called *combinational* logic circuits.

- To implement a finite-state machine, some kind of *memory* is generally necessary to remember the previous state. These are called *sequential* logic circuits.
From our viewpoint, time appears to be a continuous variable.

For a digital sequence, we want discrete values \([x_0, x_1, x_2, x_3, \ldots]\), not a continuous function \(x(t)\).

The typical way to handle this is to use a clock.

The continuous sequence is “sampled” at regularly-spaced times, when the clock “ticks”.
Sampling a Signal

\[ \text{time} \]
Sampling a Signal

(In reality, rises and falls aren’t so vertical.)

as sometimes shown in engineering drawings

an actual waveform
Sampling a Signal

0 1 0 1 1 0

clock ticks

time
Clock Rate

- The clock is analogous to the conductor of a symphony orchestra: it keeps all of the players in synch.

- The rate at which the clock ticks is the quoted rate of the processor, e.g. 500 MHz (500,000,000 ticks per second).

- It is possible to design systems that don’t have clocks (“asynchronous systems“) but these are rare.
The Basic Unit of Memory is the Flip-Flop

- A flip-flop remembers **one bit**, either a 0 or 1.
- The presence of a synchronizing clock is assumed.
- The bit is held from one clock-tick to the next.
- Each time the clock ticks, whatever value (0 or 1) exists at the flip-flop’s input is remembered; the old value is lost.
Flip-Flop Behavior

Input value now

Input value at last tick

Ticking clock

D values remembered

Ticks

Time

\(d\)

\(q\)

\(0\)

\(1\)

\(0\)

\(1\)

\(1\)

\(0\)
Inside a Flip-Flop

- A flip-flop can be constructed from ordinary gates (which have some associated switching delay) and feed-back connections.
- A first approximation, called a clocked latch, is:
Clocked Latch Behavior
This flip-flop changes only when the clock edge occurs, unlike the latch, which can change back and forth during the time the clock is 1.

Analysis is left to the reader.
We can encode an arbitrary state set just as we encode any set, in terms of some number of bits.

When the encoding has been chosen, we need one flip-flop per bit.

We implement the next-state function using combinational logic: given an encoded version of the current state, produce the encoding of the next state.
Next-State Sequential Logic for a Finite-State Machine

- Bits encoding the current state
- Flip-flops (clock not shown)
- Combinational logic for the next-state function (gates)
- Bits encoding the input

Diagram:

- Flowchart showing the relationship between the current state, input, and next-state logic.
Example: Implement a multiples-of-3 machine

Suppose we encode the state set using 2 bits, thus:
Example: Implement a multiples-of-3 machine

The next state is summarized by the following table:

<table>
<thead>
<tr>
<th>next state</th>
<th>input</th>
</tr>
</thead>
<tbody>
<tr>
<td>current state</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
</tr>
</tbody>
</table>

But we already know how to implement such a table in logic!
Logic for Multiple-of-3

<table>
<thead>
<tr>
<th>current state (vw)</th>
<th>input (x)</th>
<th>next vw</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>00</td>
<td>00</td>
<td>0</td>
</tr>
<tr>
<td>01</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>01</td>
<td>1</td>
</tr>
</tbody>
</table>

next v:

What are the simplified logic functions?
Logic for Multiple-of-3

next vw
current state (vw) | input (x)
--- | --- | ---
00 | 00 | 01
01 | 10 | 00
10 | 01 | 10

next v: 

next w:

\[ \text{next } v = wx' + vx \]

\[ \text{next } w = vx' + v'w'x \]
\[ \text{next } v = wx' + vx \]

\[ \text{next } w = vx' + v'w'x \]
Logic Diagram in Context

Flip-Flops

initial state \( v \ w = 0\ 0 \)
Output Considerations

- We need to drive the output from the encoded state.
- The output is coded, just like the state and input.
- Let's say that the output is $z$ which has value 1 for accepting, 0 for rejecting.
- Since the only accepting state is 00, the output function is

$$z = v' \cdot w'$$
Final Circuit, with Output

Flip-Flops
initial state v w = 0 0
We can check our result by “reverse engineering”, that is construct the state diagram from the logic itself.

From the drawing, we have:

- next $v = vx + wx'$
- next $w = vx' + v'w'x$
- initial $v w = 0 0$
- output $z = v'w'$
Reverse Engineering

- From the drawing, we have:
  - next \( v = vx + wx' \)
  - next \( w = vx' + v'w'x \)
  - initial \( v \ w = 0 \ 0 \)
  - output \( z = v'w' \)

- Construct a diagram with states = \( v \ w \), starting with initial state \( 0 \ 0 \):
  - \( 0 \ 0 \) with input 0 \( \rightarrow 0 \ 0 \)
  - \( 0 \ 1 \) with input 0 \( \rightarrow \)
  - \( 1 \ 0 \) with input 0 \( \rightarrow \)
  - 0 0 with input 1 \( \rightarrow \)
  - 0 1 with input 1 \( \rightarrow \)
  - 1 0 with input 1 \( \rightarrow \)
Reverse Engineering

- From the drawing, we have:
  - next $v = vx + wx'$
  - next $w = vx' + v'w'x$
  - initial $v, w = 0, 0$
  - output $z = v'w'$

- Construct a diagram with states = $v, w$, starting with initial state 0 0:

  0 0 with input 0 $\rightarrow$ 0 0
  0 0 with input 1 $\rightarrow$ 0 1
  0 1 with input 0 $\rightarrow$ 1 0
  0 1 with input 1 $\rightarrow$ 0 0
  1 0 with input 0 $\rightarrow$ 0 1
  1 0 with input 1 $\rightarrow$ 1 0
Reverse Engineering

- The resulting diagram:

```
  00  01  10
|   1   |
  0   0  1
```

which is what we started with.

- In general, reverse engineering will start from the logic.
Physical Computers

(as distinguished from “virtual” computers, such as Turing machines)
Computer Components

- Finite-state machines
- Combinational logic
- Busses
A register is a finite-state machine that remembers values as bit vectors.

A register may perform other functions as well:
- Clearing
- Incrementing, decrementing
- Shifting
Simplest register

Remembers one bit = Flip-Flop
Two-Bit Register
Inputs to a Register

- Data (e.g. value to be remembered)
- “Strobe”: function to be performed
- Simplest register has no strobe inputs
Register with Strobe Input

- a strobe
- load: 1 remember input, 0 leave as is
- input
- contents
Register with Two Strobe Inputs
1-bit register with load & clear

- \{clear, load\} is 1-hot (never both 1 simultaneously)
Strobe Possibilities

- load
- clear
- increment
- decrement
- complement
- left-shift
- right-shift
Register Transfer

Equivalent Java: \[ A = B; \]
Selective Register Transfer

Equivalent Java: \( A = P \ ? \ B : C; \)

See also: Boole/Shannon Expansion
4-way selection
Selection Using a Bus

For this to make sense, we need another register output value separate from 0,1.
Implementing Bus Connection

- We can’t simply use AND-gates; the output of an AND will always be 0 or 1.
- Connecting together wires with 0 and 1 simultaneously would be fatal.
- For the bus, use a third possible output value:
  - “high impedance”, “high Z”, or NC (no connection)
3-State Buffer

\[ \text{output} = \text{control} \ ? \ \text{data} : \text{NC} ; \]

No Connection
Selection using Bus

\{b, c, d\} is one-hot
The bus-type connection allows selection from a large number of inputs without requiring a multiplexor tree or other complex logic.
Computing using Combinational Functions

Java: \[ A = B + C; \]

The actual computation takes place "between" clock ticks.

The clock simply strobes the result into the register.