

## Finite-State Machines as Hardware

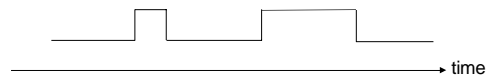
## Implementing Finite-State Machines using Logic+Memory

- 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.

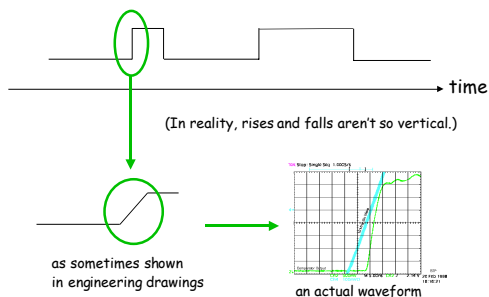
## Representing a Discrete Sequence in Continuous Time

- From our viewpoint, time appears to be a continuous variable.
- For a digital sequence, we want discrete values  $[x_0, x_1, x_2, x_3, \dots]$ , 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".

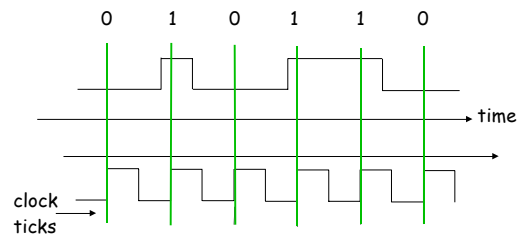
## Sampling a Signal



## Sampling a Signal



## Sampling a Signal



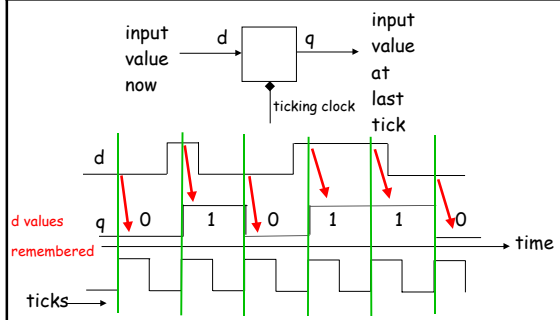
## Clock Rate

- 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

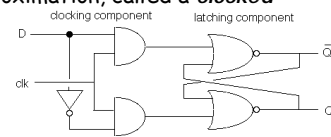
- A flip-flop remembers one bit, either a 0 or 1.
- The presence of a synchronizing clock is assumed.
- The bit is held from one clock-tick to the next.
- Each time the clock ticks, whatever value (0 or 1) exists at the flip-flop's input is remembered; the old value is lost.

## Flip-Flop Behavior

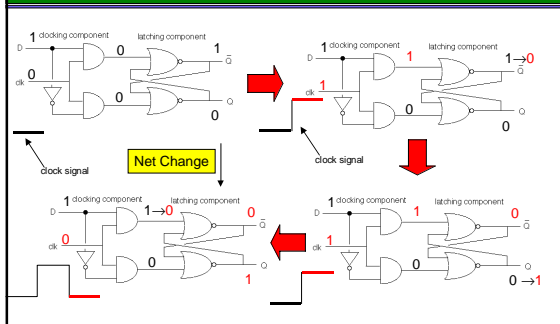


## Inside a Flip-Flop

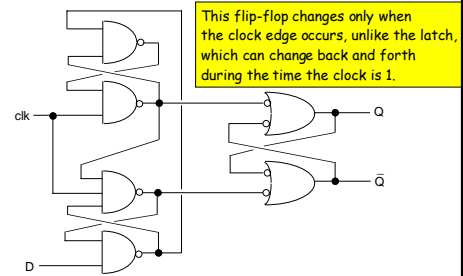
- A flip-flop can be constructed from ordinary gates (which have some associated switching *delay*) and feed-back connections.
- A first approximation, called a *clocked latch*, is:



## Clocked Latch Behavior



## Edge-Triggered Flip-Flop

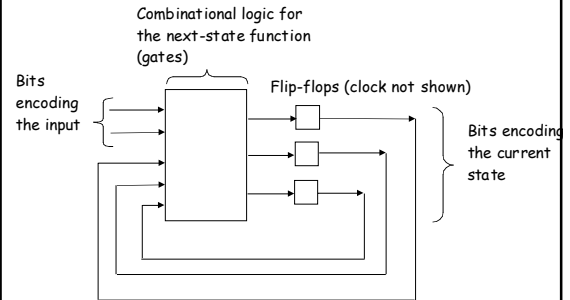


Analysis is left to the reader.

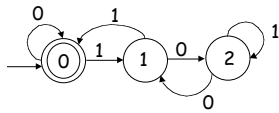
## Encoding An Arbitrary State Set for a Finite-State Machine

- 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.

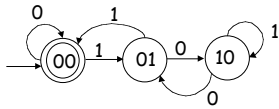
## Next-State Sequential Logic for a Finite-State Machine



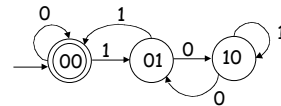
## Example: Implement a multiples-of-3 machine



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:

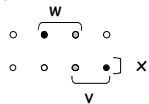
next state current state	input	
	0	1
00	00	01
01	10	00
10	01	10

But we already know how to implement such a table in logic!

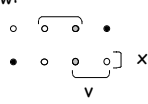
## Logic for Multiple-of-3

next vw current state (vw)	input (x)	
	0	1
00	00	01
01	10	00
10	01	10

next v:



next w:

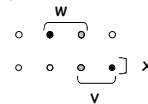


What are the simplified logic functions?

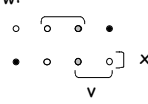
## Logic for Multiple-of-3

next vw current state (vw)	input (x)	
	0	1
00	00	01
01	10	00
10	01	10

next v:



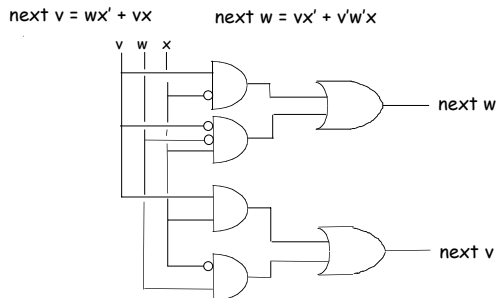
next w:



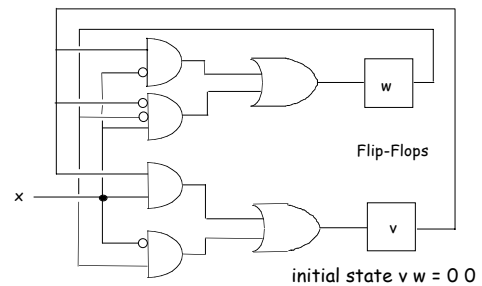
$$\text{next } v = wx' + vx$$

$$\text{next } w = vx' + v'w'x$$

## Logic Diagram



## Logic Diagram in Context

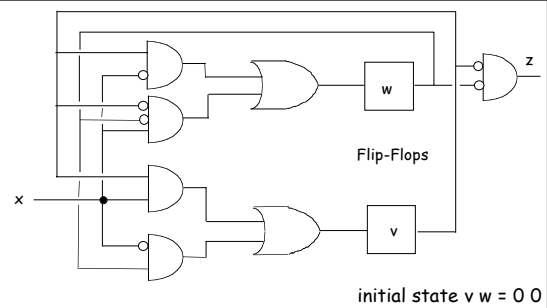


## Output Considerations

- 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

$$z = v'w'$$

## Final Circuit, with Output



## Reverse Engineering

- 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 → 0 0	0 0 with input 1 →
0 1 with input 0 →	0 1 with input 1 →
1 0 with input 0 →	1 0 with input 1 →

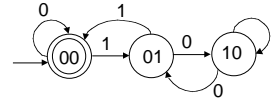
## Reverse Engineering

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



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

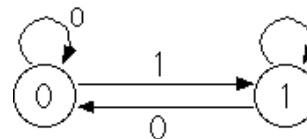
- Finite-state machines
- Combinational logic
- Busses

## Register

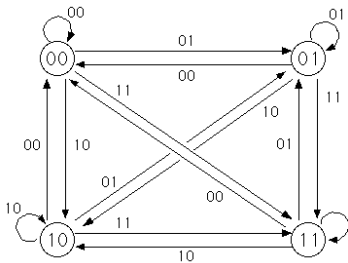
- A register is a finite-state machine that remembers values as bit vectors.
- A register may perform other functions as well:
  - Clearing
  - Incrementing, decrementing
  - Shifting

## Simplest register

Remembers one bit = Flip-Flop

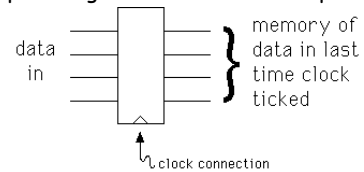


## Two-Bit Register

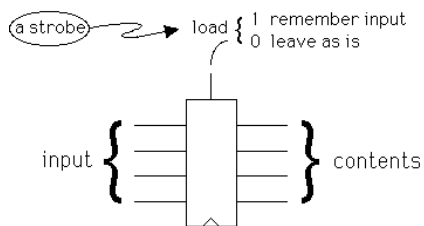


## Inputs to a Register

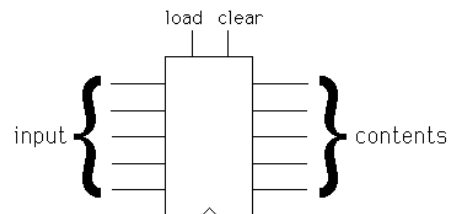
- Data (e.g. value to be remembered)
- "Strobe": function to be performed
- Simplest register has no strobe inputs



## Register with Strobe Input

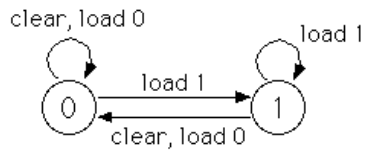


## Register with Two Strobe Inputs



## 1-bit register with load & clear

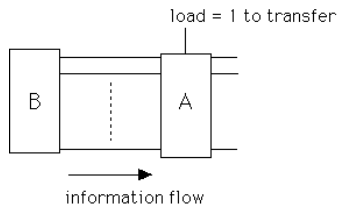
- {clear, load} is 1-hot (never both 1 simultaneously)



## Strobe Possibilities

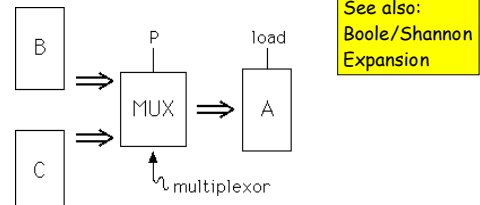
- load
- clear
- increment
- decrement
- complement
- left-shift
- right-shift

## Register Transfer



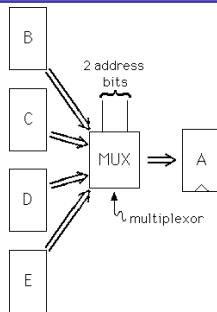
Equivalent Java: `A = B;`

## Selective Register Transfer

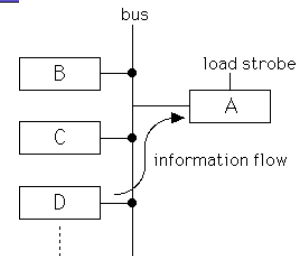


Equivalent Java: `A = P ? B : C;`

## 4-way selection



## Selection Using a Bus

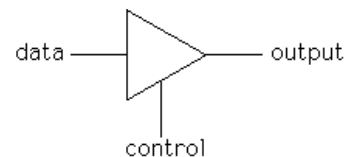


For this to make sense, we need another register output value separate from 0,1.

## 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)

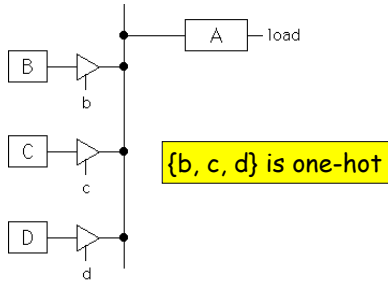
## 3-State Buffer



output = control ? data : NC;

No Connection

### Selection using Bus



### Bus vs. Multiplexor

- 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

