Memory I

CSCI 2050U - Computer Architecture

Randy J. Fortier
@randy_fortier

Outline

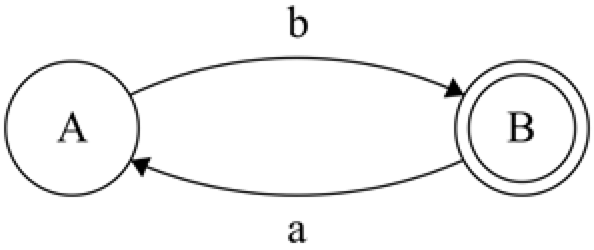
- Finite state machines
- Oscillators (clock)
- Latches

Finite State Machines

CSCI 2050U - Computer Architecture

Finite State Machines (FSMs)

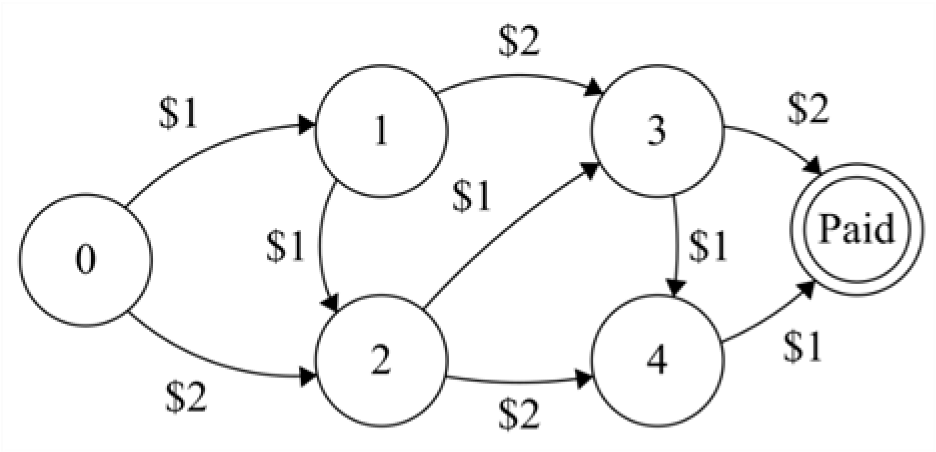
- Also called finite state automata (singular: finite state automaton)
- A model which as a finite number of states
  - Inputs may cause transitions between those states
- They are used to model situations where we care about *what happened before*
- This is a state machine that differentiates between strings that end with `a` and strings that end with `b`:



Drawn using http://madebyevan.com/fsm/

Finite State Machines (FSMs)

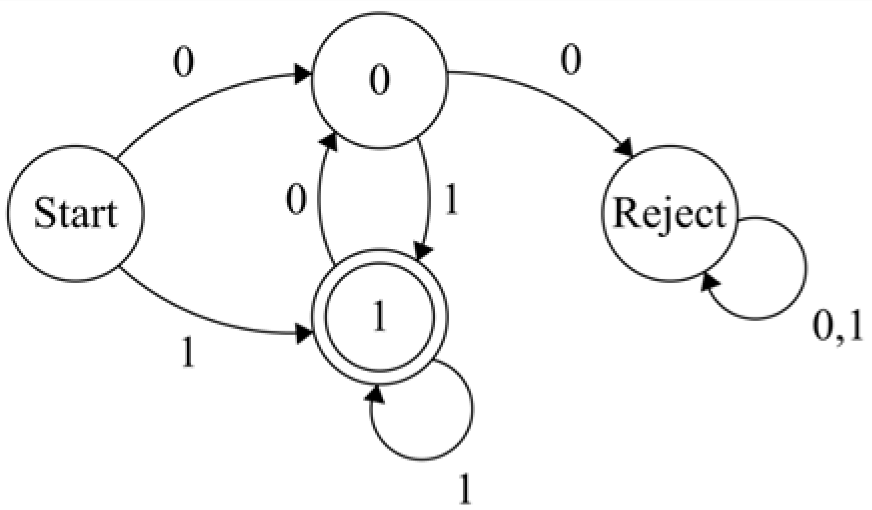
- Let's say that we want to create a device for our parking lot that will let our customers pay the flat rate of $5 using $1 or $2 coins:



Drawn using http://madebyevan.com/fsm/

Finite State Machines (FSMs)

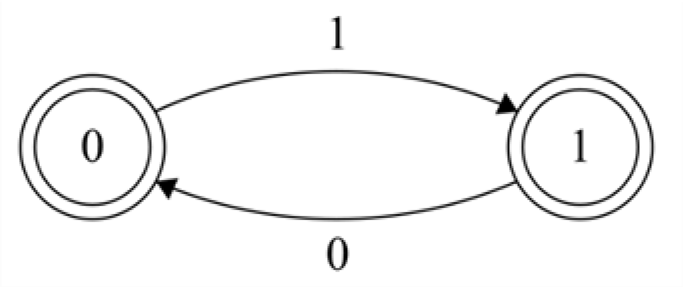
- Let's say that we want to create a device for recognizing a sequence of bits
  - The sequence must end in `1`, but cannot have `00` anywhere
  - Note the *self transitions* in this state diagram



Drawn using http://madebyevan.com/fsm/

Finite State Machines (FSMs)

- An FSM representing a 1-bit storage:



Drawn using http://madebyevan.com/fsm/

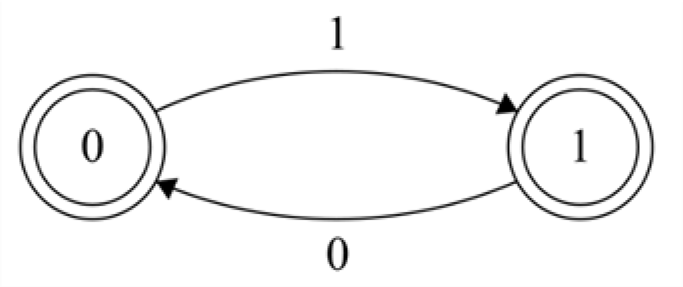
Finite State Machines (FSMs)

- Let's do an example
  - Let's create an FSM to recognize our locker combination: `4 6 4`

Drawn using http://madebyevan.com/fsm/

FSM → Circuit

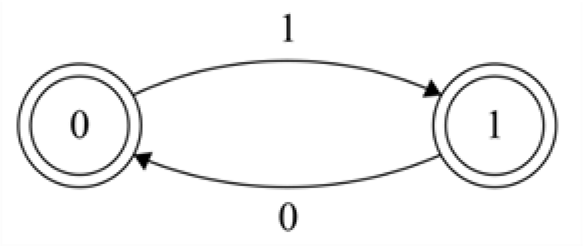
- Let's try to convert this simple FSM into a circuit:



Drawn using *http://madebyevan.com/fsm/*

FSM → Circuit

- What if we did this?
    - Does this work?

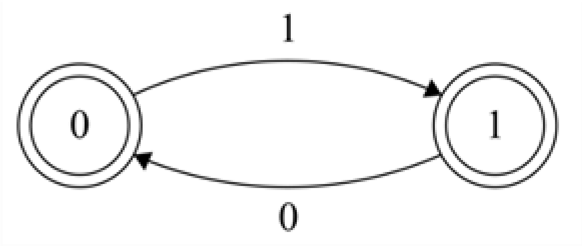


input	Q_i	Q_{i+1}
0	0	0
0	1	0
1	0	1
1	1	1

Drawn using http://madebyevan.com/fsm/

FSM → Circuit

- What if we did this?
    - Does this work?
    - It works once, but we need a continuing storage
    - Let's forge ahead, anyway



input	***Q***	***Q'***
0	0	0
0	1	0
1	0	1
1	1	1

FSM → Circuit

- Simplifying this circuit (e.g. with a K-map) will be left as an exercise

`Q' = input`

`Q' ———————— input`

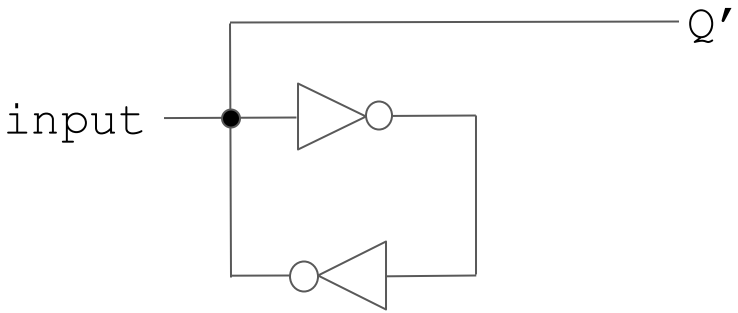
input	*Q*	*Q'*
0	0	0
0	1	0
1	0	1
1	1	1

FSM → Circuit

- This won't work repeatedly
  - There are no logic gates to refresh the signal
  - Attenuation and noise will eventually destroy the signal

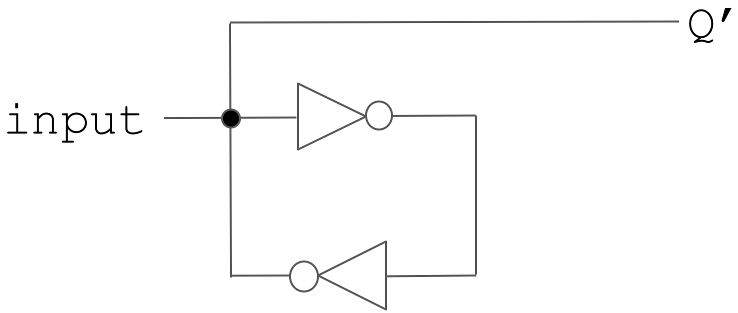
FSM → Circuit

- This is a bit better
  - The NOT gates will refresh the signal
  - With two NOT gates, the value will not change
- The problem is that any change to `input` will change the value
  - If `input` always had the correct value, we would not need memory
  - What if `input` comes from an ALU circuit?



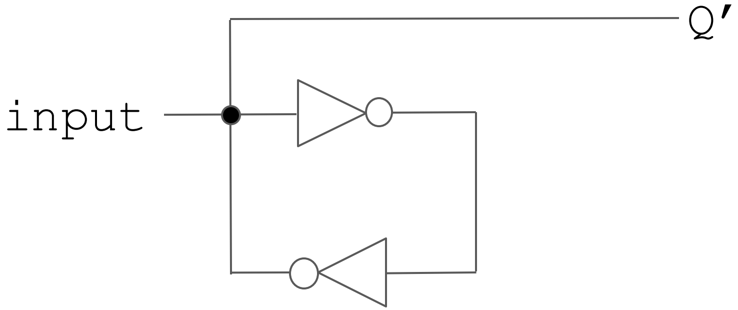
FSM → Circuit

- The circuit below is our first sequential logic circuit
  - You can always identify a sequential logic circuit by the *feedback* (cycles, circular patterns) in the graph
- Imagine the value continuously travelling around the circular path
  - How fast?



FSM → Circuit

- The circuit below is our first sequential logic circuit
  - You can always identify a sequential logic circuit by the *feedback* (cycles, circular patterns) in the graph
- Imagine the value continuously travelling around the circular path
  - How fast?
  - The signal would be slower than the speed of light, due to the wire's resistance
  - This is called the circuit's *propagation delay*



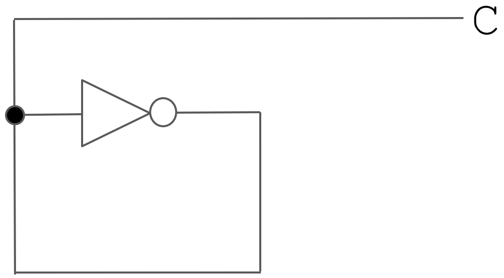
Oscillators and the Clock

CSCI 2050U - Computer Architecture

Ontario Tech University

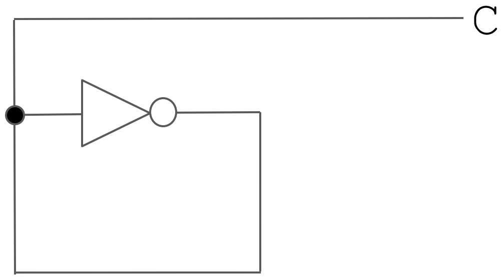
Oscillator

- The circuit below is a bit surprising
  - It may seem like a logical contradiction
  - The input to the NOT gate is the same as its output, yet a NOT gate inverts its input
- Remembering that every circuit has a *propagation delay*, though, it should become obvious that this circuit oscillates between 0 and 1:



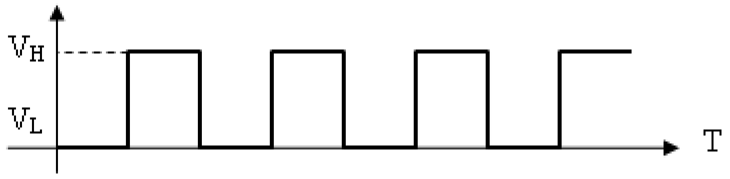
The Clock

- Modern computers use something like an oscillator to choreograph the operations of its circuits
  - The signal generated by the oscillator is called the *clock signal*
- The oscillators used in moderns computers looks nothing like this
  - We expect to be able to tune the clock frequency to our needs
    - Factory clocking
    - Overclocking



The Clock

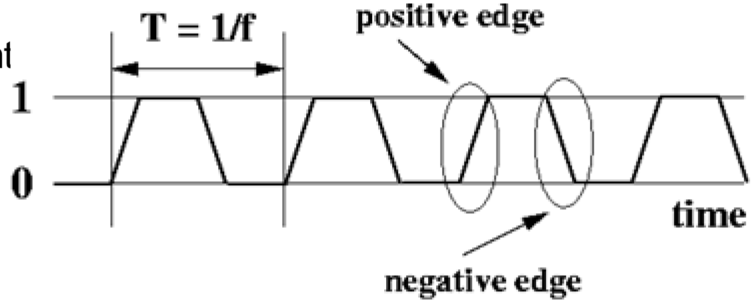
- The purpose of the clock is to make sure everything happens at the right time
    - Load the first value into register A from memory
    - Load the second value into register B from memory
    - Add register A to register B, putting the result into register A
    - Store register A into memory
- The clock signal is a predictable pattern, with regular timing



$$f = 1/t$$

The Clock

- The purpose of the clock is to make sure everything happens at the right time
  - Load the first value into register A from memory:
    `Positive edge`
  - Load the second value into register B from memory
    `Negative edge`
  - Add register A to register B, putting the result into register A `Positive edge`
  - Store register A int



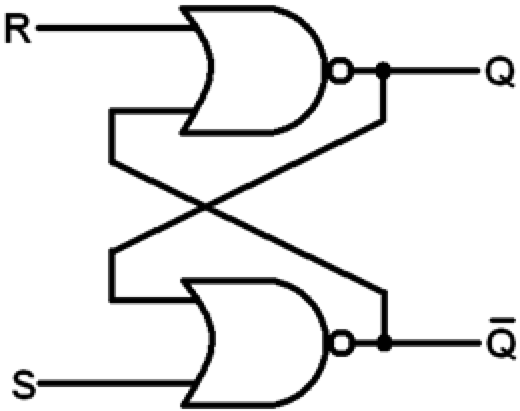
Storage

CSCI 2050U - Computer Architecture

Ontario Tech University

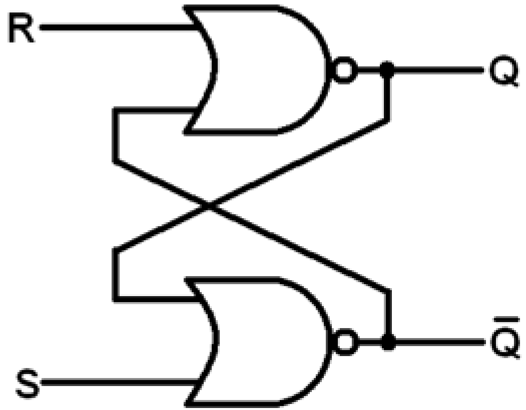
Latches

- The sequential logic circuits, with feedback, that we've seen are used to create *latches* and *flip flops*
- A latch is a 1-bit storage sequential circuit
- Here is a common latch, called an S-R latch:



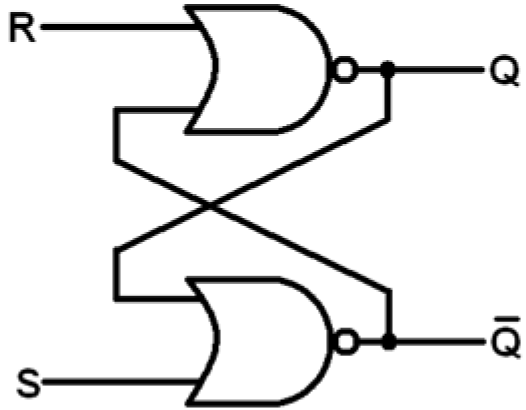
Latches

- The S-R latch will be a good introductory storage component
    - `S`: Set - when this input is `1`, the value stored in the latch will become `1`
    - `R`: Reset - when this input is `1`, the value stored in the latch will become `0`



Latches

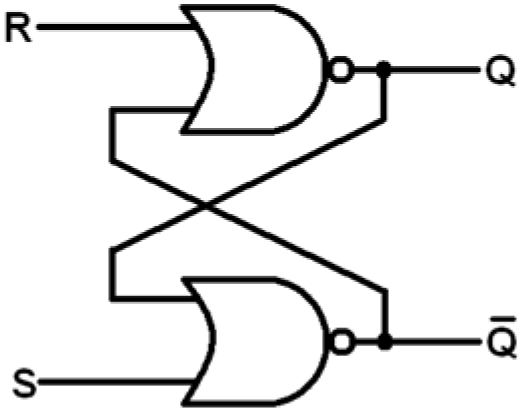
- Let's examine every combination of input values:



Q_i	R	S	Q_{i+1}	$\overline{Q_{i+1}}$
0	0	0	?	?
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Latches

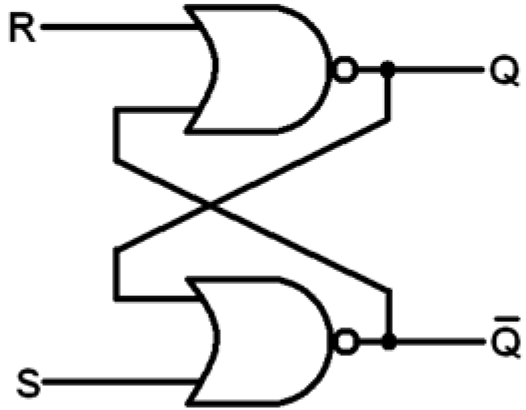
- Let's examine every combination of input values:



Q_i	R	S	Q_{i+1}	Q_{i+1}'
0	0	0	0	1
0	0	1	?	?
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Latches

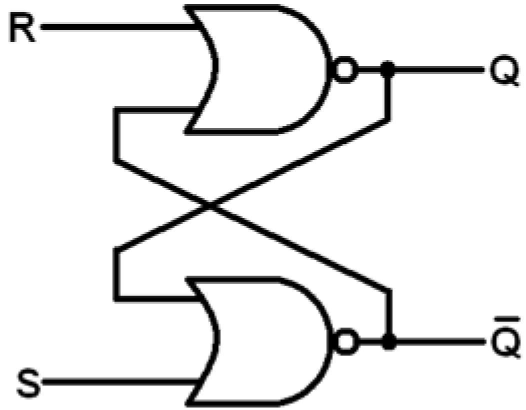
- Let's examine every combination of input values:



Q_i	R	S	Q_{i+1}	Q_{i+1}'
0	0	0	0	1
0	0	1	1	0
0	1	0	?	?
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Latches

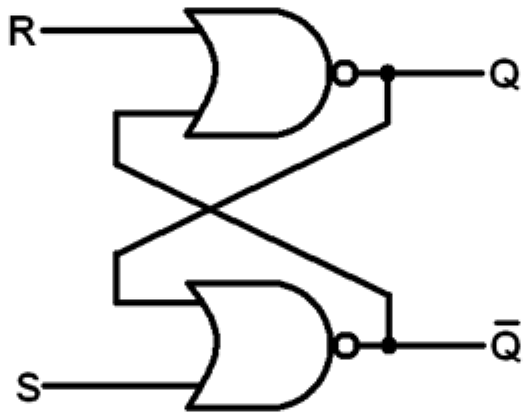
- Let's examine every combination of input values:



Q_i	R	S	Q_{i+1}	Q_{i+1}'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	?	?
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Latches

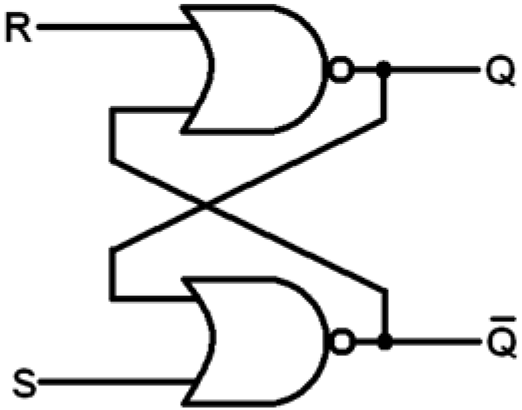
- Let's examine every combination of input values:



Q_i	R	S	Q_{i+1}	Q_{i+1}'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	undef	undef
1	0	0	?	?
1	0	1		
1	1	0		
1	1	1	undef	undef

Latches

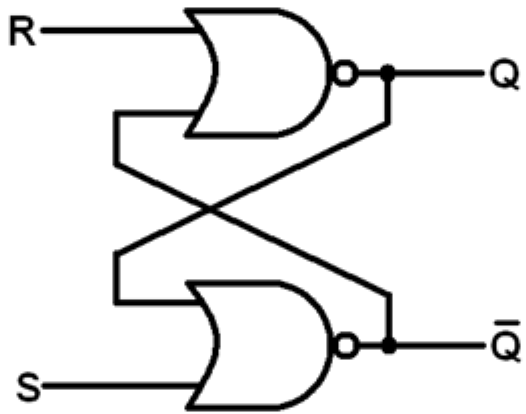
- Let's examine every combination of input values:



Q_i	R	S	Q_{i+1}	Q_{i+1}'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	undef	undef
1	0	0	1	0
1	0	1	?	?
1	1	0		
1	1	1	undef	undef

Latches

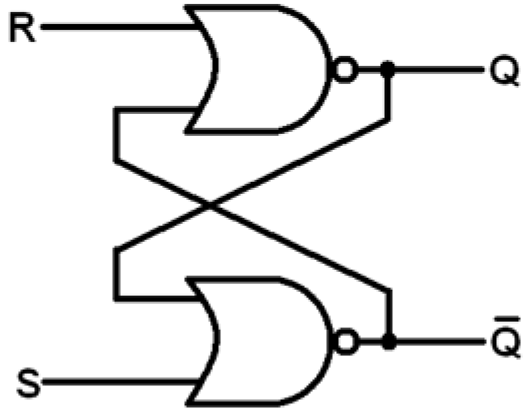
- Let's examine every combination of input values:



Q_i	R	S	Q_{i+1}	Q_{i+1}'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	undef	undef
1	0	0	1	0
1	0	1	1	0
1	1	0	?	?
1	1	1	undef	undef

Latches

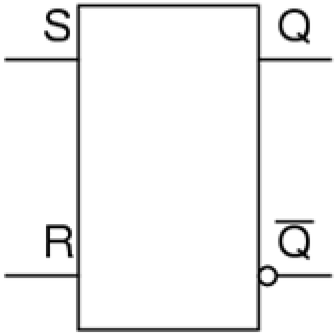
- Let's examine every combination of input values:



Q_i	R	S	Q_{i+1}	Q_{i+1}'
0	0	0	0	1
0	0	1	1	0
0	1	0	0	1
0	1	1	undef	undef
1	0	0	1	0
1	0	1	1	0
1	1	0	0	1
1	1	1	undef	undef

SR Latches

- We can re-write this table more efficiently:
- This is the block diagram representation of an SR latch:



R	S	Q_{i+1}
0	0	Q_i
0	1	1
1	0	0
1	1	undef

Wrap-up

- Finite state machines
- Oscillators (clock)
- Latches

Up Next

- Flip flops
- Registers
  - Counters
- RAM
- The memory hierarchy