# Lecture 4c: State Machines

CS105: Great Insights in Computer Science
Michael L. Littman, Fall 2006

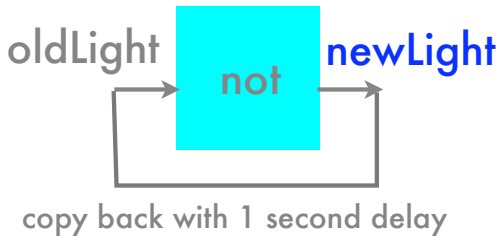
## Perspective

- Ok, here's where we are: We can use logic gates to take a set of input values (Trues and Falses) and create a set of output values.
- Things start to get interesting when we take those outputs and feed them back in as inputs!
- Such a device can be called a "state machine".

# Simple, Concrete Example

- Let's say we want to create blinking Christmas lights (once every second).
- Let "oldLight" be a Boolean variable that represents whether the light was on a second ago and "newLight" represent whether it should be on now.
- What is "newLight" in terms of "oldLight"?

# Blinking




| oldLight | newLight |
|---|---|
| False | True |
| True | False |

- Want:
  - oldLight = False makes newLight = True
  - oldLight = True makes newLight = False

# Christmas Light Programs

- **All Flash**
- A=not A
- B=False
- C=False
- A
- A
- A
- A
- A
- A
- A
- A

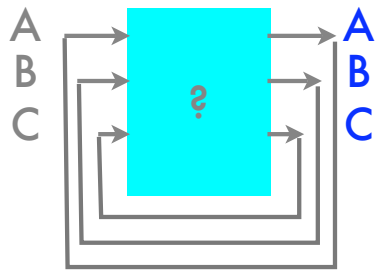
- **Odds/Evens**
- A=not A
- B=False
- C=False
- A
- not A
- A
- not A
- A
- not A
- A
- not A

# That's It!?

- So, that's a computer.
- Well, actually a computer has more inputs and outputs and the internal logic is more complex.
- But, that's it. So, let's start increasing the complexity to bridge the gap.

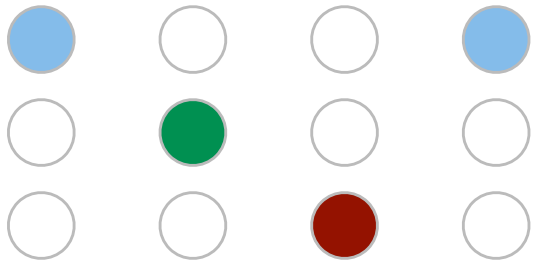
# "Traveling" Lights

- Flashing three lights in sequence gives the illusion of the light "traveling" in one direction.
- Need a few more bits to make it work:




copy back with 1 second delay

# State Sequence



| | | | | |
|---|---|---|---|---|
| A | True | False | False | True |
| B | False | True | False | False |
| C | False | False | True | False |

# Truth Table Segment

| A | B | C | A | B | C |
|---|---|---|---|---|---|
| True | False | False | False | True | False |
| False | True | False | False | False | True |
| False | False | True | True | False | False |
| | | | A=C | B=A | C=B |

# Christmas Light Programs

- **Travel-3**
- A=C
- B=A
- C=B
- A
- B
- C
- A
- B
- C
- A
- B

- **Travel-3 with reset**
- A=C or X
- B=A and not X
- C=B and not X
- A
- B
- C
- A
- B
- C
- A
- B

# How Reset Works

| A | B | C | X | A | B | C |
|---|---|---|---|---|---|---|
| True | False | False | False | False | True | False |
| False | True | False | False | False | False | True |
| False | False | True | False | True | False | False |
| True | False | False | True | True | False | False |
| False | True | False | True | True | False | False |
| False | False | True | True | True | False | False |

# Puzzle… Traveling for Less!

- We're using all three bits (A, B, and C) to create the traveling effect.
- Can we do the same thing with only A and B?
- Note that the logical expressions on the light bulbs will have to be somewhat different.

# Truth Table Segment

| A | B | A | B |
|---|---|---|---|
| True | False | False | True |
| False | True | False | False |
| False | False | True | False |

A=not A and not B B=A

"C" lights go on when A and B off: not A and not B

# Christmas Light Program

- **Travel-3 with 2 bits**
- A=not A and not B
- B=A
- C=False
- A
- B
- not A and not B
- A
- B
- not A and not B
- A
- B

# Insight: Binary Addition

- With 3 bits (A, B, and C), the state machine should be able to represent 8 patterns.
- If A, B, and C encode a number in binary, we want A, B, and C to represent that number plus 1.

# Incrementing (Adding 1)

| A | B | C | A | B | C |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 |

not C

(B and not C) or (not B and C)

not (B == C)

(A and not (B and C)) or (not A and (B and C))

# Christmas Light Programs

- **Travel-8**
- A= (A and not (B and C)) or (not A and (B and C))
- B= B ^ C ["xor"]
- C=not C
- not A and not B and not C
- not A and not B and C
- not A and B and not C
- not A and B and C
- A and not B and not C
- A and not B and C
- A and B and not C
- A and B and C

- **Bounce-5**
- A= (A and not (B and C)) or (not A and (B and C))
- B= B ^ C ["xor"]
- C=not C
- not A and not B and not C
- (not A and not B and C) or (A and B and C)
- (not A and B and not C) or (A and B and not C)
- (not A and B and C) or (A and not B and C)
- A and not B and not C (leave others False)

# Inputs and Outputs

- A computer is (roughly!):
  - A state machine with **a lot** of bits
  - Complex logic relating their values
  - Very fast cycle time
  - Devices that set the bits (input)
  - Devices that display the bits (output)

# Count To 4

| X | B | C | B | C |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | | | | |
| 1 | | | | |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |

(X and not C) or
(not X and C)

(not X and B) or
(X and ((B and not C) or (not B and C)))

# Christmas Light Program

- **Counter-4**
- A = False
- B = (not X and B) or (X and ((B and not C) or (not B and C)))
- C =(X and not C) or (not X and C)
- not A and not B and not C
- not A and not B and C
- not A and B and not C
- not A and B and C
- False
- False
- False
- False

# Implementing Addition

- Half adder: Takes two bits and a carry and outputs a bit and a carry (addc).
- Adder: Adds two 8-bit numbers (discards last carry) (addbyte).

```
def addbyte(x,y):
    z = [0]*8
    sum7 = addc(x[7],y[7],0)
    z[7] = sum7[1]
    sum6 = addc(x[6],y[6],sum7[0])
    z[6] = sum6[1]
    sum5 = addc(x[5],y[5],sum6[0])
    z[5] = sum5[1]
    sum4 = addc(x[4],y[4],sum5[0])
    z[4] = sum4[1]
    sum3 = addc(x[3],y[3],sum4[0])
    z[3] = sum3[1]
    sum2 = addc(x[2],y[2],sum3[0])
    z[2] = sum2[1]
    sum1 = addc(x[1],y[1],sum2[0])
    z[1] = sum1[1]
    sum0 = addc(x[0],y[0],sum1[0])
    z[0] = sum0[1]
    return z
```

```
def addc(a,b,c):
    bit = (a and not b and not c) or (not a and b and not c) or (not a and not b and c) or (a and b and c)
    carry = (a and b and not c) or (a and not b and c) or (not a and b and c) or (a and b and c)
    return([carry, bit])
```

# Next Time

- We're ready to build ourselves a computer!
- Read Hillis, Chapter 3.