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”.

• 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”.

Perspective
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

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

oldLight newLight
False True
True False

oldLight newLight
<table>
<thead>
<tr>
<th>oldLight</th>
<th>newLight</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
Christmas Light Programs

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

State Sequence

<table>
<thead>
<tr>
<th></th>
<th>True</th>
<th>False</th>
<th>False</th>
<th>True</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>C</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
### Truth Table Segment

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td></td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td></td>
</tr>
</tbody>
</table>

\[ A = C \quad B = A \quad C = B \]

### 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

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>X</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td><strong>True</strong></td>
<td><strong>True</strong></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td><strong>True</strong></td>
<td><strong>True</strong></td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td><strong>True</strong></td>
<td><strong>True</strong></td>
<td>False</td>
<td>False</td>
</tr>
</tbody>
</table>

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

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>

\[ A = \text{not} \ A \text{ and not} \ B \quad B = A \]

“\( C \)” lights go on when \( A \) and \( B \) off: not \( A \) and not \( B \)

Christmas Light Program

- Travel-3 with 2 bits
- \( A = \text{not} \ A \text{ and not} \ B \)
- \( B = A \)
- \( C = \text{False} \)
- \( A \)
- \( B \)
- 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)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- 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
  - \( A = (A \text{ and } \neg(B \text{ and } C)) \text{ or } (\neg A \text{ and } (B \text{ and } C)) \)
  - \( B = B \oplus C \) \[“xor”\]
  - \( C = \neg C \)
  - \( \neg A \text{ and } \neg B \text{ and } \neg C \)
  - \( \neg A \text{ and } \neg B \text{ and } C \)
  - \( \neg A \text{ and } B \text{ and } \neg C \)
  - \( \neg A \text{ and } B \text{ and } C \)
  - \( A \text{ and } \neg B \text{ and } \neg C \)
  - \( A \text{ and } \neg B \text{ and } C \)
  - \( A \text{ and } B \text{ and } \neg C \)
  - \( A \text{ and } B \text{ and } C \)

- Bounce-5
  - \( A = (A \text{ and } \neg(B \text{ and } C)) \text{ or } (\neg A \text{ and } (B \text{ and } C)) \)
  - \( B = B \oplus C \) \[“xor”\]
  - \( C = \neg C \)
  - \( \neg A \text{ and } \neg B \text{ and } \neg C \)
  - \( (\neg A \text{ and } \neg B \text{ and } C) \text{ or } (A \text{ and } B \text{ and } C) \)
  - \( (\neg A \text{ and } B \text{ and } \neg C) \text{ or } (A \text{ and } B \text{ and } \neg C) \)
  - \( (\neg A \text{ and } B \text{ and } C) \text{ or } (A \text{ and } \neg B \text{ and } C) \)
  - \( A \text{ and } \neg B \text{ and } \neg C \)
  - \( \text{leave others False}\)

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

<table>
<thead>
<tr>
<th>X</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(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 B and C
- not A and B and not 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).

```python
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])

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])
    sum5 = addc(x[5], y[5], sum6[0])
    sum4 = addc(x[4], y[4], sum5[0])
    sum3 = addc(x[3], y[3], sum4[0])
    sum2 = addc(x[2], y[2], sum3[0])
    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
```

Next Time

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