def AND3(A, B, C):
    x = A and B
    y = x and C
    return y

def OR4(A, B, C, D):
    return (A or B) or (C or D)
def IFTHENELSE5(bit, char1, char2):
    return [(char1[0] and bit) or (char2[0] and not bit),
            (char1[1] and bit) or (char2[1] and not bit),
            (char1[2] and bit) or (char2[2] and not bit),
            (char1[3] and bit) or (char2[3] and not bit),
            (char1[4] and bit) or (char2[4] and not bit)]

    • Takes 11 bits as input and makes 5 as output. For clarity, the bits are grouped.
    • char1[0] means the leftmost bit of the group called “char1”.
    • “bit” selects char1 (True) or char2 (False).
Why “Or”, “And”, “Not”? 

- In addition to being familiar, these gates are “universal”. That is, all other logical functions can be expressed using these building blocks.
- How many distinct logic functions on 2 bits?

Some Truth Tables

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>
More Truth Tables

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

Universal Gate

- Take two inputs, A and B.
- Take four more inputs defining what the gate should output for each combination of A and B.
- Output the right bit!
Textual Version

def UNIV2(A, B, d00, d01, d10, d11):
    w = AND3(not A, not B, d00)
    x = AND3(not A, B, d01)
    y = AND3(A, not B, d10)
    z = AND3(A, B, d11)
    return OR4(w, x, y, z)

Bit Equality

def equal(bit1, bit2):
    return ((bit1 and bit2) or (not bit1 and not bit2))

• Output True if either bit1 = True and bit2 = True, or bit1 = False and bit2 = False (they are equal).
Group Equality

- Now that we can test two bits for equality, we would like to test a group of 5 bits for equality (two bit patterns).
- Two groups are equal if each of their bits are equal: bit 0 = bit 0, bit 1 = bit 1, etc.

```python
def equal5(char1, char2):
    return (equal(char1[0],char2[0])
            and (equal(char1[1],char2[1])
                 and (equal(char1[2],char2[2])
                      and (equal(char1[3],char2[3])
                           and equal(char1[4],char2[4]))))
```

Equal5 Diagram
Gates in EQUAL5

- 10 inputs (2 groups of 5), 1 output bit.
- The equal5 gate consists of
  - 1 “and5” gate
  - 4 “and” gates (4 total)
  - 5 “equal” gates
  - 2 “and”, 2 “not”, 1 “or” (5 total)
  - Total = 29 gates

Gates: Could Create

- And-\(k\): \(k\) ins, 1 out (True if all ins are True)
- Or-\(k\): \(k\) ins, 1 out (True if any ins are True)
- Ifthenelse-\(k\): 1 control bit in, \(k\) then ins, \(k\) else ins, \(k\) outs (outs match then if control bit is True, else otherwise)
- Equal-\(k\): 2 \(k\)-bit blocks in, 1 out (True if blocks same)
- Universal-\(k\): \(2^k\) table in, \(k\) control bits in, 1 out (equal to the value in the table specified by the control bits)
Counting Boolean Functions

• With 2 input bits, there are $2^2=4$ rows of the truth table (combinations of truth assignments to these variables).

• Each row can take an output of true or false, for a total of $2^4=16$ tables.

• For $n$ inputs: $2^n$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$2^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>256</td>
</tr>
<tr>
<td>4</td>
<td>65536</td>
</tr>
<tr>
<td>5</td>
<td>4294967296</td>
</tr>
<tr>
<td>6</td>
<td>18446744073709551616</td>
</tr>
<tr>
<td>7</td>
<td>34028236692093846346</td>
</tr>
<tr>
<td></td>
<td>3374607431768211456</td>
</tr>
</tbody>
</table>

Can Represent Them All

• Almost all multi-input functions require an enormous number of logic gates.

• However, the most useful ones can be represented succinctly.
Number Magic

- Let’s do a magic trick!
- How does it work?

Which Have Your Number?

- Think of a number from 0 to 31.
- Add the upper left number from each card your number appears on.
- It is...

```
16 17 18 19
20 21 22 23
24 25 26 27
28 29 30 31

8 9 10 11
12 13 14 15
20 21 22 23
28 29 30 31

4 5 6 7
12 13 14 15
20 21 22 23
28 29 30 31

2 3 6 7
10 11 14 15
18 19 22 23
26 27 30 31

1 3 5 7
9 11 13 15
17 19 21 23
25 27 29 31
```
Conversion

• To go from decimal to binary, start with the biggest power of 2 no bigger than your number.

• Write down a 1. Subtract the power of 2 from your number.

• Cut the power of 2 in half.

• If your remaining number is larger than the power of 2, write down a 1 and subtract the power of 2.

• If not, write down 0.

• Repeat by cutting the power of 2 in half (until you get to 1).

It’s a bit like making change.

Example: Convert 651

• Bigger than: \(2^9 = 512\).
  \[651 - 512 = 139\].

• Next power of 2 = 256. 0

• Next power of 2 = 128. 1
  \[139 - 128 = 11\].

• Next power of 2 = 64. 0

• Next power of 2 = 32. 0  Last power of 2 = 1. 1

1010001011 = 651
**Binary Addition**

\[
\begin{array}{c}
01110011 \\
+ 10110010 \\
\hline
111001\_ \\
\end{array}
\]

115
+178
\hline
293

- Just like in school: work right to left, carry when needed.
- \(0+0+0=0, 0+0+1 = 1, 0+1+1=10, 1+1+1=11\)
- Can check via conversion.

**Subtraction**

\[
\begin{array}{c}
0110_10_1 \\
100100101 \\
-10110010 \\
\hline
1110011 \\
\end{array}
\]

- As in decimal, proceed right to left, borrowing if not doing so would force us to subtract a bigger number from a smaller one.
**Overflow**

- When working with numbers made of a fixed number of bits, carries can “overflow”, meaning we might not be able to represent the full sum. Example (8 bits):

\[
\begin{align*}
01010011 \\
+10110010 \\
\text{(1)00000101}
\end{align*}
\]

**Negation**

- Overflow provides an interesting way to think of negation.
- Recall in algebra, an additive inverse of \( x \) is the number \( y \) such that \( x + y = 0 \). So, \( y = -x \).

\[
\begin{align*}
01111111_2 \\
10000000_2 \\
-01010011_2 \\
+01010011_2 \\
10101101_2 (1)00000000_2
\end{align*}
\]
Two’s Complement

• To find the negation of a number, flip all the bits, then add one:

\[
\begin{array}{c}
01010011 \\
10101100 \\
+00000001 \\
\hline
10101101
\end{array}
\]

83
172 = 255–83

83
173 = 256–83 = “–83”

Subtraction as Negate/Add

• Combining these ideas, we can subtract one number from another by taking the two’s complement and adding!
Multiplication

- Of course, multiplication can be carried out by repeated addition, but it’s a very inefficient way to go with big numbers.
- Our standard grade-school approach to multiplication carries forward to binary numbers as well.

Multiplication Example

- Boils down to:
  - copy, shift, add

  \[
  \begin{array}{c}
  10101101 \\
  \times 01010011 \\
  \hline
  10101101 \\
  10101101 \\
  10101101
  \end{array}
  \]

  \[
  \begin{array}{c}
  10101101 \\
  +10101101 \\
  \hline
  11100000010111
  \end{array}
  \]
Other Operations

- Can also define long division.
- Can do bitwise logic operations (and, or, not).
- All are quite useful...

Other Number Schemes

- Can represent negative numbers, often via twos complements. \(-1 = 256 - 1 = 255\).
- Fixed-width fractions (for dollar amounts).
- Floating point representations via exponential notation: \(a \times 10^b\).
- Complex numbers: real and imaginary parts.
  They are just bits: you can use them as you see fit.
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
```

State Machines

- 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

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

<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
- \(A=\text{not } A\)
- \(B=\text{False}\)
- \(C=\text{False}\)
- \(A\)
- \(A\)
- \(A\)
- \(A\)
- \(A\)
- \(A\)
- \(A\)
- \(A\)
- \(A\)
- Odds/Evens
- \(A=\text{not } A\)
- \(B=\text{False}\)
- \(C=\text{False}\)
- \(A\)
- \(\text{not } A\)
- \(A\)
- \(\text{not } A\)
- \(A\)
- \(\text{not } A\)
- \(A\)
- \(\text{not } A\)

### 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:

```
<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>
```

State Sequence

```
A  True  False  False  True
B  False  True  False  False
C  False  False  True  False
```
### 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>True</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</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
- \( A = C \) or X
- \( B = A \) and not X
- \( C = B \) and not X
- A
- B
- C
- A
- B
- C
- A
- B
How Reset Works

<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>X</td>
<td>A</td>
<td>B</td>
<td>C</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>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>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
<td>True</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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<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
- \( A = \text{not } A \text{ and not } B \)
- \( B = A \)
- \( C = \text{False} \)
- \( A \)
- \( B \)
- \( \text{not } A \text{ and not } B \)
- \( A \)
- \( B \)
- \( \text{not } A \text{ and not } B \)
- \( A \)
- \( B \)
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)

<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>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</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>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

- (A and not (B and C)) or (not A and (B and C))
- not C
- (B and not C) or (not B and C)
- not (B == C)
Christmas Light Programs

- Travel-8
  - \( A = (A \text{ and not } (B \text{ and } C)) \text{ or } (\text{not } A \text{ and } (B \text{ and } C)) \)
  - \( B = B \oplus C \) \[“xor”\]
  - \( C = \text{not } C \)
  - \( \text{not } A \text{ and not } B \text{ and not } C \)
  - \( \text{not } A \text{ and not } B \text{ and } C \)
  - \( \text{not } A \text{ and } B \text{ and not } C \)
  - \( A \text{ and not } B \text{ and not } C \)
  - \( A \text{ and not } B \text{ and } C \)
  - \( A \text{ and } B \text{ and not } C \)
  - \( A \text{ and } B \text{ and } C \)

- Bounce-5
  - \( A = (A \text{ and not } (B \text{ and } C)) \text{ or } (\text{not } A \text{ and } (B \text{ and } C)) \)
  - \( B = B \oplus C \) \[“xor”\]
  - \( C = \text{not } C \)
  - \( \text{not } A \text{ and not } B \text{ and not } C \)
  - \( (\text{not } A \text{ and not } B \text{ and } C) \text{ or } (A \text{ and } B \text{ and } C) \)
  - \( (\text{not } A \text{ and not } B \text{ and not } C) \text{ or } (A \text{ and } B \text{ and not } C) \)
  - \( (\text{not } A \text{ and } B \text{ and } C) \text{ or } (A \text{ and not } B \text{ and } C) \)
  - \( A \text{ and not } B \text{ and not } C \)
  - \( (\text{leave others False}) \)

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>

\[(X \text{ and not } C) \text{ or } (\text{not } X \text{ and } C)\]

\[(\text{not } X \text{ and } B) \text{ or } (X \text{ and } ((B \text{ and not } C) \text{ or } (\text{not } B \text{ and } C)))\]

### Christmas Light Program

- Counter-4
- \(A = \text{False}\)
- \(B = (\text{not } X \text{ and } B) \text{ or } (X \text{ and } ((B \text{ and not } C) \text{ or } (\text{not } B \text{ and } C)))\)
- \(C = (X \text{ and not } C) \text{ or } (\text{not } X \text{ and } C)\)
- \(\text{not } A \text{ and not } B \text{ and not } C\)
- \(\text{not } A \text{ and not } B \text{ and } C\)
- \(\text{not } A \text{ and } B \text{ and not } C\)
- \(\text{not } A \text{ and } B \text{ and } C\)
- False
- False
- False
- False
Where Are We?

- We now have all the pieces to build a simple, working computer...
- Each cycle, inputs propagate to outputs, which are copied back to inputs to begin again.
- We need a language to talk to it in, though.

Recap

- Using logic gates, we know how to do a bunch of things with bits:
  - test equality
  - if-then-else gate
  - select one bit from a set (universal gate)
What Can We Do?

• Lots: Any function of bits, we can specify with logic gates.

• But, creating dedicated circuitry for every new problem is daunting and inefficient.

• Would like a way of using a fixed set of circuits to act like any circuitry we might want.

• We can use the state-machine idea to trade gates for time...

Concrete Example: Adding

\[
\begin{array}{c}
10 \\
11 \\
+ 10 \\
101 \\
\end{array} \quad \begin{array}{c}
c_1c_0 \\
x_1x_0 \\
+y_1y_0 \\
z_2z_1z_0 \\
\end{array}
\]

• We want to compute the sum of \( x \) and \( y \) (2-bit numbers). \( z \) (3 bits) is the answer and \( c \) (2 bits) is the carry.

• \( z_0 = (x_0 \text{ and not } y_0) \) or \((\text{not } x_0 \text{ and } y_0)\)

• \( c_0 = (x_0 \text{ and } y_0)\)
Concrete Example: Adding

\[
\begin{array}{cccc}
10 & c_1c_0 \\
11 & x_1x_0 \\
+ 10 & y_1y_0 \\
101 & z_2z_1z_0 \\
\end{array}
\]

- \( z_1 = (x_1 \text{ and not } y_1 \text{ and not } c_0) \text{ or } (\text{not } x_1 \text{ and } y_1 \text{ and not } c_0) \text{ or } (\text{not } x_1 \text{ and not } y_1 \text{ and } c_0) \text{ or } (x_1 \text{ and } y_1 \text{ and } c_0) \)
- \( c_1 = z_2 = (x_1 \text{ and } y_1 \text{ and not } c_0) \text{ or } (x_1 \text{ and not } y_1 \text{ and } c_0) \text{ or } (\text{not } x_1 \text{ and } y_1 \text{ and } c_0) \text{ or } (x_1 \text{ and } y_1 \text{ and } c_0) \)

Adding Bytes

- Computing \( z_i \) and \( c_i \) from \( x_i, y_i, \) and \( c_{i-1} \) can be carried out with 6 ands, 3 ors, 4 nots.
- The previous slide uses 16 ands, 6 ors, and 9 nots (not as good).
- This operation is called a “full adder”.
- By chaining together one full adder per bit, we can make a circuit that adds any number of bits (4, 8, 16, 32, 64, etc.).
**Hardware**

- So, any function we want to implement from bits to bits can be carried out by constructing the right circuit of and/or/nots.
- Creating a circuit solves the problem “in hardware”.
- The advantage of hardware solutions are that they are fast.
- The disadvantage is that they are inflexible.

**Software**

- The lovely thing about a computer is that the hardware does not have to change for the computer to change its behavior.
- A fixed set of circuits can actually change its behavior to represent any desired function!
- Build one, reprogram into anything.
- Disadvantage of the software approach: Can be much slower.
Programming an Adder

Circuit level:

Instruction level:

• H = A xor B = (A and not B) or (not A and B)

• C₁ = (A and B) or (C and H)

• S = H xor C = (H and not C) or (not H and C)

Simple Statements

• Still too many different statements.

• Break complex statements down into a set of simple statements.

• Instead of E = (H and not C) or (not H and C):
  • acc = not C
  • acc = acc and H
  • P = acc
  • acc = not H
  • acc = acc and C
  • acc = acc or P
  • E = acc
Instruction Set: 7 Bits

- V in 0000...1111 (variables A- P)
- 000V: acc = acc or V
- 001V: acc = acc and V
- 010V: acc = V
- 011V: acc = not V
- acc: special temporary variable
- 100V: V = acc or V
- 101V: V = acc and V
- 110V: V = acc
- 111V: V = not acc

Memory

- Need a place to store the various quantities we’re working with.
- Main memory is like a giant filing cabinet, where each drawer is numbered consecutively and can store one value.
- Need to be able to store and retrieve values.
Memory Circuit

- We’ll have 32 memory locations, for instructions.
- Each one has an 5-bit name (0-31) called its “address”.
- Memory circuit takes the contents of memory (32 x 7 bits) and an address, 229 bits in all, & outputs the data stored at the corresponding address.

Memory Lookup

```python
def memlookup(add, mem):
    mem00 = mem[0]
    mem01 = ifthenelse7(equal5(intToByte( 1),add), mem[ 1], mem00)
    mem02 = ifthenelse7(equal5(intToByte( 2),add), mem[ 2], mem01)
    mem03 = ifthenelse7(equal5(intToByte( 3),add), mem[ 3], mem02)
    mem04 = ifthenelse7(equal5(intToByte( 4),add), mem[ 4], mem03)
    mem05 = ifthenelse7(equal5(intToByte( 5),add), mem[ 5], mem04)
    mem06 = ifthenelse7(equal5(intToByte( 6),add), mem[ 6], mem05)
    mem07 = ifthenelse7(equal5(intToByte( 7),add), mem[ 7], mem06)
    mem08 = ifthenelse7(equal5(intToByte( 8),add), mem[ 8], mem07)
    mem09 = ifthenelse7(equal5(intToByte( 9),add), mem[ 9], mem08)
    mem10 = ifthenelse7(equal5(intToByte(10),add), mem[10], mem09)
    mem11 = ifthenelse7(equal5(intToByte(11),add), mem[11], mem10)
    mem12 = ifthenelse7(equal5(intToByte(12),add), mem[12], mem11)
    mem13 = ifthenelse7(equal5(intToByte(13),add), mem[13], mem12)
    mem14 = ifthenelse7(equal5(intToByte(14),add), mem[14], mem13)
    mem15 = ifthenelse7(equal5(intToByte(15),add), mem[15], mem14)
    mem16 = ifthenelse7(equal5(intToByte(16),add), mem[16], mem15)
    mem17 = ifthenelse7(equal5(intToByte(17),add), mem[17], mem16)
    mem18 = ifthenelse7(equal5(intToByte(18),add), mem[18], mem17)
    mem19 = ifthenelse7(equal5(intToByte(19),add), mem[19], mem18)
    mem20 = ifthenelse7(equal5(intToByte(20),add), mem[20], mem19)
    mem21 = ifthenelse7(equal5(intToByte(21),add), mem[21], mem20)
    mem22 = ifthenelse7(equal5(intToByte(22),add), mem[22], mem21)
    mem23 = ifthenelse7(equal5(intToByte(23),add), mem[23], mem22)
    mem24 = ifthenelse7(equal5(intToByte(24),add), mem[24], mem23)
    mem25 = ifthenelse7(equal5(intToByte(25),add), mem[25], mem24)
    mem26 = ifthenelse7(equal5(intToByte(26),add), mem[26], mem25)
    mem27 = ifthenelse7(equal5(intToByte(27),add), mem[27], mem26)
    mem28 = ifthenelse7(equal5(intToByte(28),add), mem[28], mem27)
    mem29 = ifthenelse7(equal5(intToByte(29),add), mem[29], mem28)
    mem30 = ifthenelse7(equal5(intToByte(30),add), mem[30], mem29)
    mem31 = ifthenelse7(equal5(intToByte(31),add), mem[31], mem30)

    return mem31
```

64-loc-input bitwise OR
Writing to Memory

- Similar circuit allows memory cells to be altered.
- `mem[address] = newval`
- If needed for future processing, copied back up at the end of the cycle.

```python
def memwrite(active, add, mem, val):
    return [
        ifthenelse7(active and equal5(intToByte(0), add), val, mem[0]),
        ifthenelse7(active and equal5(intToByte(1), add), val, mem[1]),
        ifthenelse7(active and equal5(intToByte(2), add), val, mem[2]),
        ifthenelse7(active and equal5(intToByte(3), add), val, mem[3]),
        ifthenelse7(active and equal5(intToByte(4), add), val, mem[4]),
        ifthenelse7(active and equal5(intToByte(5), add), val, mem[5]),
        ifthenelse7(active and equal5(intToByte(6), add), val, mem[6]),
        ifthenelse7(active and equal5(intToByte(7), add), val, mem[7]),
        ifthenelse7(active and equal5(intToByte(8), add), val, mem[8]),
        ifthenelse7(active and equal5(intToByte(9), add), val, mem[9]),
        ifthenelse7(active and equal5(intToByte(10), add), val, mem[10]),
        ifthenelse7(active and equal5(intToByte(11), add), val, mem[11]),
        ifthenelse7(active and equal5(intToByte(12), add), val, mem[12]),
        ifthenelse7(active and equal5(intToByte(13), add), val, mem[13]),
        ifthenelse7(active and equal5(intToByte(14), add), val, mem[14]),
        ifthenelse7(active and equal5(intToByte(15), add), val, mem[15]),
        ifthenelse7(active and equal5(intToByte(16), add), val, mem[16]),
        ifthenelse7(active and equal5(intToByte(17), add), val, mem[17]),
        ifthenelse7(active and equal5(intToByte(18), add), val, mem[18]),
        ifthenelse7(active and equal5(intToByte(19), add), val, mem[19]),
        ifthenelse7(active and equal5(intToByte(20), add), val, mem[20]),
        ifthenelse7(active and equal5(intToByte(21), add), val, mem[21]),
        ifthenelse7(active and equal5(intToByte(22), add), val, mem[22]),
        ifthenelse7(active and equal5(intToByte(23), add), val, mem[23]),
        ifthenelse7(active and equal5(intToByte(24), add), val, mem[24]),
        ifthenelse7(active and equal5(intToByte(25), add), val, mem[25]),
        ifthenelse7(active and equal5(intToByte(26), add), val, mem[26]),
        ifthenelse7(active and equal5(intToByte(27), add), val, mem[27]),
        ifthenelse7(active and equal5(intToByte(28), add), val, mem[28]),
        ifthenelse7(active and equal5(intToByte(29), add), val, mem[29]),
        ifthenelse7(active and equal5(intToByte(30), add), val, mem[30]),
        ifthenelse7(active and equal5(intToByte(31), add), val, mem[31])]
```

Memory Write
Persistence of Memory

- We can use this memory idea to store the Boolean variables (A-P).
- We can also use another set of memory locations to store the series of instructions to be executed (program).
- How is the instruction stored?

One Instruction

```
<table>
<thead>
<tr>
<th>Load/store (1 bit)</th>
<th>variable name (4 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: load; 1: store</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>instruction (2 bits)</th>
<th>1011000</th>
</tr>
</thead>
<tbody>
<tr>
<td>00: acc or V</td>
<td>store = 1</td>
</tr>
<tr>
<td>01: acc and V</td>
<td>instruction = 01</td>
</tr>
<tr>
<td>10: acc (load)/V (store)</td>
<td>constant = 1000 = I</td>
</tr>
<tr>
<td>11: not acc (load) / not V (store)</td>
<td>So, “I = acc and I”</td>
</tr>
</tbody>
</table>
```
**Series of Instructions**

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents (Decimal)</th>
<th>Contents (Binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>51</td>
<td>1110011 acc = not D</td>
</tr>
<tr>
<td>1</td>
<td>17</td>
<td>0010000 acc = acc and B</td>
</tr>
<tr>
<td>2</td>
<td>111</td>
<td>1011111 P = acc</td>
</tr>
<tr>
<td>3</td>
<td>49</td>
<td>0110001 acc = not B</td>
</tr>
<tr>
<td>4</td>
<td>19</td>
<td>0101111 acc = acc and D</td>
</tr>
<tr>
<td>5</td>
<td>15</td>
<td>0011111 acc = acc or P</td>
</tr>
<tr>
<td>6</td>
<td>104</td>
<td>1010000 I = acc</td>
</tr>
<tr>
<td>7</td>
<td>33</td>
<td>0100001 acc = B</td>
</tr>
<tr>
<td>8</td>
<td>19</td>
<td>0101011 acc = acc and D</td>
</tr>
</tbody>
</table>

Program counter: which address’s instruction to process next

Accumulator: Special register

Registers: Boolean variables and their values

Michael Littman’s Mini Logic Machine Language (ML³)

---

**von Neumann Architecture**

- A computer is just a big state machine.
- Input: registers, memory, input devices
- Output: new values for registers, memory, output devices
- PC = Program counter, the address of the statement to be executed.

<table>
<thead>
<tr>
<th>mem</th>
<th>acc</th>
<th>PC</th>
<th>reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>7x32 bits</td>
<td>5 bits</td>
<td>1 bit</td>
<td>1x16 bits</td>
</tr>
<tr>
<td>246 bits total</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU = Central Processing Unit
Cycle: A Whole Computer

mem look up 32x7
add byte 5

mem lookup 16x1


ir 0
mem write 16x1

v ir 1 ir 2

if then else and
val

not or and

and 3 if then else
not

more of the same...

Cycle: Symbolically

def cycle(input):
    [mem, pc, acc, reg] = input
    ir = memlookup32x7(pc, mem)
    pcnew = addbyte5(pc, intToByte5(1))
    v = ir[3:]
    val = memlookup16x1(v, reg)
    res = ifthenelse(not ir[1] and not ir[2], acc or val, False)
    res = ifthenelse(not ir[1] and ir[2], acc and val, res)
    res = ifthenelse(not ir[0] and ir[1] and not ir[2], val, res)
    res = ifthenelse(ir[0] and ir[1] and not ir[2], acc, res)
    res = ifthenelse(not ir[0] and ir[1] and ir[2], not val, res)
    res = ifthenelse(ir[0] and ir[1] and ir[2], not acc, res)
    accnew = ifthenelse(ir[0], acc, res)
    regnew = memwrite16x1(ir[0], v, reg, res)
    return [mem, pcnew, accnew, regnew]
Instruction Sets

- ML3 used a particular design that made it relatively easy to fit in a lecture slide while handling 2-bit addition.

- Computer manufacturers have different goals in mind: cost, speed, ease of running modern programs.

- Some quick examples:

x86: Intel's Old Set
Z80: My First

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>INP</td>
<td>Input – take a number from the input card and put it in a specified memory cell.</td>
</tr>
<tr>
<td>1</td>
<td>CLA</td>
<td>Clear and add – clear the accumulator and add the contents of a memory cell to the accumulator.</td>
</tr>
<tr>
<td>2</td>
<td>ADD</td>
<td>Add - add the contents of a memory cell to the accumulator.</td>
</tr>
<tr>
<td>3</td>
<td>TAC</td>
<td>Test accumulator contents – performs a sign test on the contents of the accumulator.</td>
</tr>
<tr>
<td>4</td>
<td>SFT</td>
<td>Shift – shifts the accumulator x places left, then y places right.</td>
</tr>
<tr>
<td>5</td>
<td>OUT</td>
<td>Output – take a number from the specified memory cell and write it on an output card.</td>
</tr>
<tr>
<td>6</td>
<td>STO</td>
<td>Store – copy the contents of the accumulator into a specified memory cell.</td>
</tr>
<tr>
<td>7</td>
<td>SUB</td>
<td>Subtract – subtract the contents of a specified memory cell from the accumulator.</td>
</tr>
<tr>
<td>8</td>
<td>JMP</td>
<td>Jump - jump to a specified memory cell.</td>
</tr>
<tr>
<td>9</td>
<td>HRS</td>
<td>Halt and reset – stop program execution, move bug to cell 00.</td>
</tr>
</tbody>
</table>

CARDiAC (1968)

Opcode Mnemonic Description
0 INP Input – take a number from the input card and put it in a specified memory cell.
1 CLA Clear and add – clear the accumulator and add the contents of a memory cell to the accumulator.
2 ADD Add - add the contents of a memory cell to the accumulator.
3 TAC Test accumulator contents – performs a sign test on the contents of the accumulator.
4 SFT Shift – shifts the accumulator x places left, then y places right.
5 OUT Output – take a number from the specified memory cell and write it on an output card.
6 STO Store – copy the contents of the accumulator into a specified memory cell.
7 SUB Subtract – subtract the contents of a specified memory cell from the accumulator.
8 JMP Jump - jump to a specified memory cell.
9 HRS Halt and reset – stop program execution, move bug to cell 00.