Chapter 2: Universal Building Blocks

CS105: Great Insights in Computer Science

A Few Gates

\[ x = A \text{ and } B \]
\[ y = x \text{ and } C \]

\[ (A \text{ or } B) \text{ or } (C \text{ or } D) \]
If Then Else #1

- Input: a
- Output: d
  - if a = True, d = True
  - Else, d = False

\[ d = a \]

If Then Else #2

- Input: a, b
- Output: d
  - if a = True, d = b
  - Else, d = False

\[ d = a \text{ and } b \]
If Then Else #3

- Input: a, c
- Output: d
  - if a = True, d = False
  - Else, d = c

\[ d = \text{not } a \text{ and } c \]

If Then Else #4

- Input: a, b, c
- Output: d
  - if a = True, d = b
  - Else, d = c

\[ d = (a \text{ and } b) \text{ or (not } a \text{ and } c) \]
### IFTHENELSE5

<table>
<thead>
<tr>
<th>control</th>
<th>bit</th>
<th>char1</th>
<th>char2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Algebraic Version

\[
[(\text{char1}[0] \text{ and bit}) \text{ or } (\text{char2}[0] \text{ and not bit})],
(\text{char1}[1] \text{ and bit}) \text{ or } (\text{char2}[1] \text{ and not bit})],
(\text{char1}[2] \text{ and bit}) \text{ or } (\text{char2}[2] \text{ and not bit})],
(\text{char1}[3] \text{ and bit}) \text{ or } (\text{char2}[3] \text{ and not bit})],
(\text{char1}[4] \text{ and bit}) \text{ or } (\text{char2}[4] \text{ and not bit})]
\]

- Takes 11 bits as input and makes 5 as output. For clarity, the bits are grouped.
- \text{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

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

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

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

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<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>A</td>
<td>B</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
## More Truth Tables

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

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

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

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

## Truth Table to Formula

If table is mostly False...

1. Make a clause for each “True”.
   - not A and not B and C
   - not A and B and C
2. Or them together.
   - (not A and not B and C) or (not A and B and C)
3. Simplify, if possible
   - D = (not A and C) or (not B and B) = not A and C

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>
Truth Table to Formula

If table is mostly True...

1. Make a clause for each “False”.
   - not A and B and not C
   - A and B and C

2. Or them together.
   - (not A and B and not C) or (A and B and C)

3. Simplify, if possible
   - B and ((not A and not C) or (A and C)) = B and (A=C)

4. Invert via DeMorgan’s Law
   - D = not(B and (A=C)) = not B or (A xor C)

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!
Algebraic Version

Input: \((A, B, d_{00}, d_{01}, d_{10}, d_{11})\):
\[w = \text{AND3}(\text{not } A, \text{not } B, d_{00})\]
\[x = \text{AND3}(\text{not } A, B, d_{01})\]
\[y = \text{AND3}(A, \text{not } B, d_{10})\]
\[z = \text{AND3}(A, B, d_{11})\]
Output: \(\text{OR4}(w,x,y,z)\)

hard to follow...

Bit Equality

Input: bit1, bit2:
Output: \(((\text{bit1 and bit2}) \text{ or } (\text{not bit1 and not bit2}))\)

- 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.

   Input: char1, char2
   Output: (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>2</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.
Patterns of Bits

• An auditory demo...
• http://scratch.mit.edu/projects/cs105/36694

Number Magic

• Let’s do a magic trick!
• How does it work?
  • http://www.brainbashers.com/games/number.asp
Which Have Your Number?

• Think of a number from 0 to 31.
• A = appears, B = does not appear
• Is it on...

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
</tbody>
</table>

Reminder: Decimal Notation

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>0</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>100</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>10^3</td>
<td>10^2</td>
<td>10^1</td>
<td>10^0</td>
</tr>
</tbody>
</table>

• $2 \times 100 + 0 \times 10 + 7 \times 1 = 207$
Binary Notation

<table>
<thead>
<tr>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>128</td>
<td>64</td>
<td>32</td>
<td>16</td>
<td>8</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$2^7$</td>
<td>$2^6$</td>
<td>$2^5$</td>
<td>$2^4$</td>
<td>$2^3$</td>
<td>$2^2$</td>
<td>$2^1$</td>
<td>$2^0$</td>
</tr>
</tbody>
</table>

$1 \times 128 + 0 \times 64 + 0 \times 32 + 1 \times 16 + 1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1$

$= 154$

How can you tell if a binary number is even?

Octopus’s Counting

- Inspired by Schoolhouse Rock
- Music by the Beatles
- Midi from Beatles Worldsite
- Words, pictures, vocals: Michael Littman
- Additional vocals: Max and Molly Littman
- for CS105: Fall 2006
Conversion To Binary

• 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: $2^9 = 512$. 1
  - $651-512=139$.
  - 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

01010001011 = 651
Welcome to the Club

- There are 10 kinds of people in the world.
- Those who understand binary.
- And those who don’t.

Binary Addition

\[
\begin{array}{c}
01110011 \\
+ 10110010 \\
\hline
111001\_ \quad 115 \\
+ 178 \\
\hline
293
\end{array}
\]

- 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: Easy Example

Borrowing is really replacing a higher denomination “coin” with lower ones.

Subtraction

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

\[
\begin{array}{c}
0111_2 \\
100100101 \\
-10110010 \\
1100111
\end{array}
\]
**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):

  \[
  \begin{array}{c}
  01010011 \\
  +10110010 \\
  \hline
  (1)00000101
  \end{array}
  \]

**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, \( y = -x \).

  \[
  \begin{array}{c}
  01111111 \\
  -00000000 \quad 10101101 \\
  -01010011 \quad +01010011 \\
  10101101 \quad (1)00000000
  \end{array}
  \]
Two’s Complement

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

\[
\begin{align*}
01010011 & \quad 83 \\
10101100 & \quad 172 = 255-83 \\
+00000001 & \quad 173 = 256-83 = "-83"
\end{align*}
\]

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

\[
\begin{array}{c}
\text{10101101} \\
\times \text{01010011} \\
\hline
\text{10101101} \\
\text{10101101} \\
\text{10101101} \\
\hline
\text{11100000010111}
\end{array}
\]
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.
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>

copy back with 1 second delay

Christmas Light Programs

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

  ![Diagram]

  copy back with 1 second delay
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

How Reset Works

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

<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
- \( A = \text{False} \)
- \( B = (\text{not } X \text{ and } B) \text{ or } (X \text{ and } ((B \text{ and } \text{not } C) \text{ or } (\text{not } B \text{ and } C))) \)
- \( C = (X \text{ and } \text{not } C) \text{ or } (\text{not } X \text{ and } C) \)
- 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