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 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

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

Control bit

char1

Group of 5 bits each

char2

\[
\begin{align*}
\text{char1[0] and bit} & \text{ or } \text{char2[0] and not bit}, \\
\text{char1[1] and bit} & \text{ or } \text{char2[1] and not bit}, \\
\text{char1[2] and bit} & \text{ or } \text{char2[2] and not bit}, \\
\text{char1[3] and bit} & \text{ or } \text{char2[3] and not bit}, \\
\text{char1[4] and bit} & \text{ or } \text{char2[4] and not bit}
\end{align*}
\]

• 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>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</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>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</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>False</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>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</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>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</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
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: \((A, B, d_{00}, d_{01}, d_{10}, d_{11})\):
\[
\begin{align*}
w &= \text{AND3}(\text{not } A, \text{not } B, d_{00}) \\
x &= \text{AND3}(\text{not } A, B, d_{01}) \\
y &= \text{AND3}(A, \text{not } B, d_{10}) \\
z &= \text{AND3}(A, B, d_{11})
\end{align*}
\]
Output: \(\text{OR4}(w, x, y, z)\)

hard to follow...

Bit Equality

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

- Output \textbf{True} if either \textbf{bit1 = True} and \textbf{bit2 = True} or \textbf{bit1 = False} and \textbf{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: $2^n$.

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...

**Number Magic**

- Let’s do a magic trick!
- How does it work?
Which Have Your Number?

- Think of a number from 0 to 31.
- A = appears, B = does not appear
- It is on...

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

1 0 0 1 1 0 1 0

128 64 32 16 8 4 2 1

2⁷ 2⁶ 2⁵ 2⁴ 2³ 2² 2¹ 2⁰

1x128+0x64+0x32+1x16+1x8+0x4+1x2+0x1

= 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.

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

1010001011 = 651
### Binary Addition

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

- 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.

\[
\begin{array}{c}
115 \\
+178 \\
\hline
293
\end{array}
\]

### Subtraction

\[
\begin{array}{c}
01101_1 \\
100100101 \\
-10110010 \\
\hline
1110011
\end{array}
\]

- 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):

```
  01010011
+10110010
(1)00000101
```

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, $y = -x$.

```
  01111111
  100000000  10101101
-01010011  +01010011
  10101101  (1)00000000
```
Two’s Complement

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

\[
\begin{align*}
01010011 & \quad 83 \\
10101100 & \quad 172 = 255-83 \\
+00000001 & \\
10101101 & \quad 173 = 256-83 = "-83"
\end{align*}
\]

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

```
10101101
x 01010011
10101101
10101101
10101101
+10101101
11100000010111
```
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

oldLight → not → 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>

copy back with 1 second delay

• 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
• 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:

```plaintext
A B C
?  A B C

copy back with 1 second delay
```
**State Sequence**

- A: True
- B: False
- C: False
- D: True
- E: False
- F: False
- G: False
- H: True
- I: False
- J: False
- K: True
- L: False
- M: False

**Truth Table Segment**

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

A=C, B=A, C=B
Christmas Light Programs

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

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
- $A = \neg A$ 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></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th></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>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

### Christmas Light Programs

- **Travel-8**
  - \( A = (A \text{ and } \text{not } (B \text{ and } C)) \text{ or } (\text{not } A \text{ and } (B \text{ and } C)) \)
  - \( B = B \oplus C \) [“xor”]
  - \( C = \text{not } C \)
  - 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**
  - \( A = (A \text{ and } \text{not } (B \text{ and } C)) \text{ or } (\text{not } A \text{ and } (B \text{ and } C)) \)
  - \( B = B \oplus C \) [“xor”]
  - \( C = \text{not } C \)
  - 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></th>
<th>X</th>
<th>B</th>
<th>C</th>
<th></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>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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