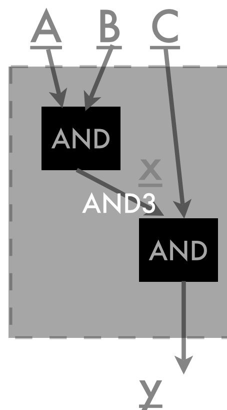
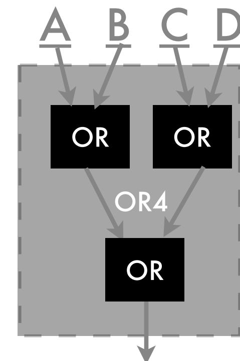
Chapter 2: Universal Building Blocks

CS105: Great Insights in Computer Science

A Few Gates



x = A and B
y = x and C



(A or B) or (C 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

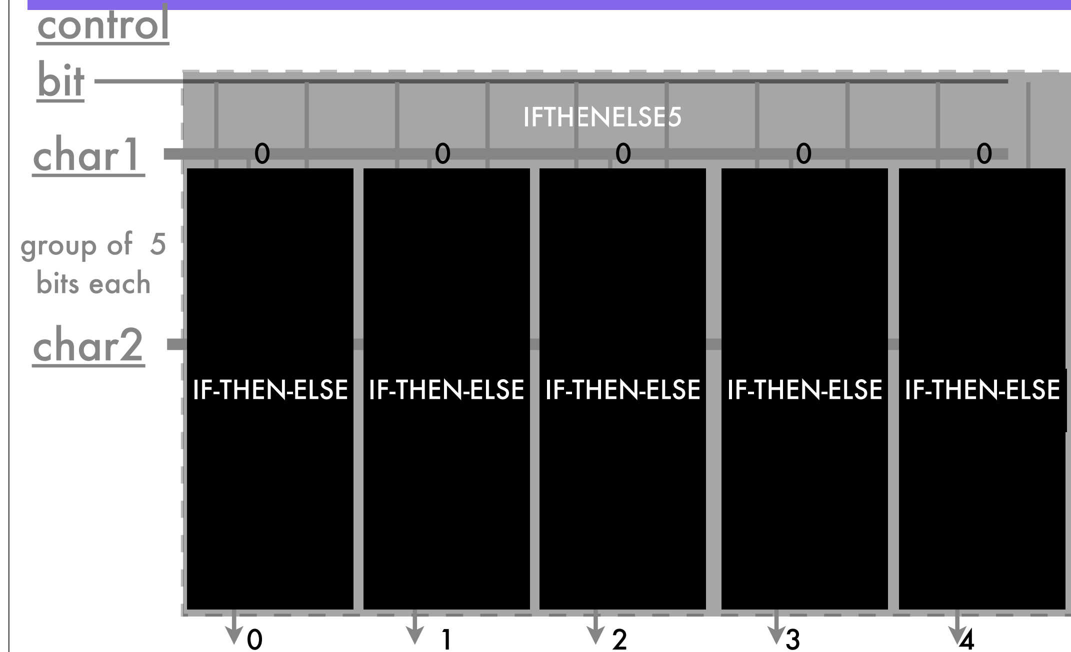
d = not a and c

If Then Else #4

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

d = (a and b) or (not a and c)

IFTHENELSE5



Algebraic Version

[(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

A	B	C
False	False	False
False	True	False
True	False	False
True	True	False

A	B	C
False	False	False
False	True	False
True	False	False
True	True	True

A	B	C
False	False	False
False	True	False
True	False	True
True	True	False

A	B	C
False	False	False
False	True	False
True	False	True
True	True	True

A	B	C
False	False	False
False	True	True
True	False	False
True	True	False

A	B	C
False	False	False
False	True	True
True	False	False
True	True	True

A	B	C
False	False	False
False	True	True
True	False	True
True	True	False

A	B	C
False	False	False
False	True	True
True	False	True
True	True	True

More Truth Tables

A	B	C
False	False	True
False	True	False
True	False	False
True	True	False

A	B	C
False	False	True
False	True	False
True	False	False
True	True	True

A	B	C
False	False	True
False	True	False
True	False	True
True	True	False

A	B	C
False	False	True
False	True	False
True	False	True
True	True	True

A	B	C
False	False	True
False	True	True
True	False	False
True	True	False

A	B	C
False	False	True
False	True	True
True	False	False
True	True	True

A	B	C
False	False	True
False	True	True
True	False	True
True	True	False

A	B	C
False	False	True
False	True	True
True	False	True
True	True	True

Truth Table to Formula

A	B	C	D
F	F	F	F
F	F	T	T
F	T	F	F
F	T	T	T
T	F	F	F
T	F	T	F
T	T	F	F
T	T	T	F

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

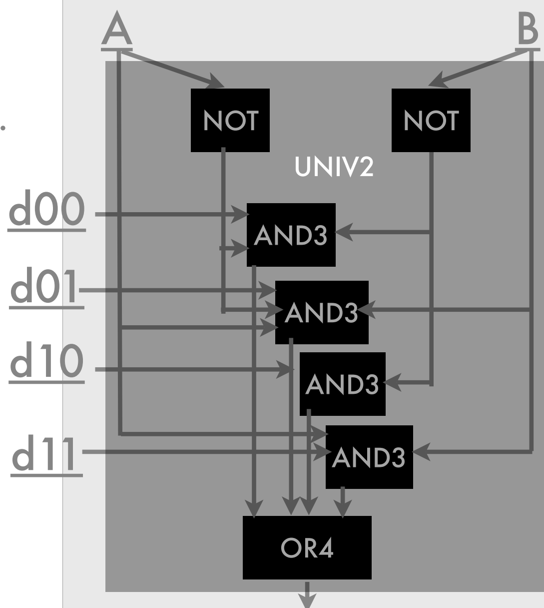
If table is mostly True...

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))

A	B	C	D
F	F	F	T
F	F	T	T
F	T	F	F
F	T	T	T
T	F	F	T
T	F	T	T
T	T	F	T
T	T	T	F

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!



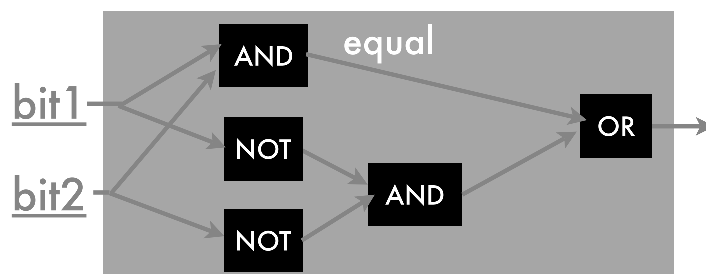
Algebraic Version

Input: (A, B, 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)
Output: OR4(w,x,y,z)

hard to follow...

Bit Equality

Input: bit1, bit2:
Output: ((bit1 and bit2) or (not bit1 and not bit2))



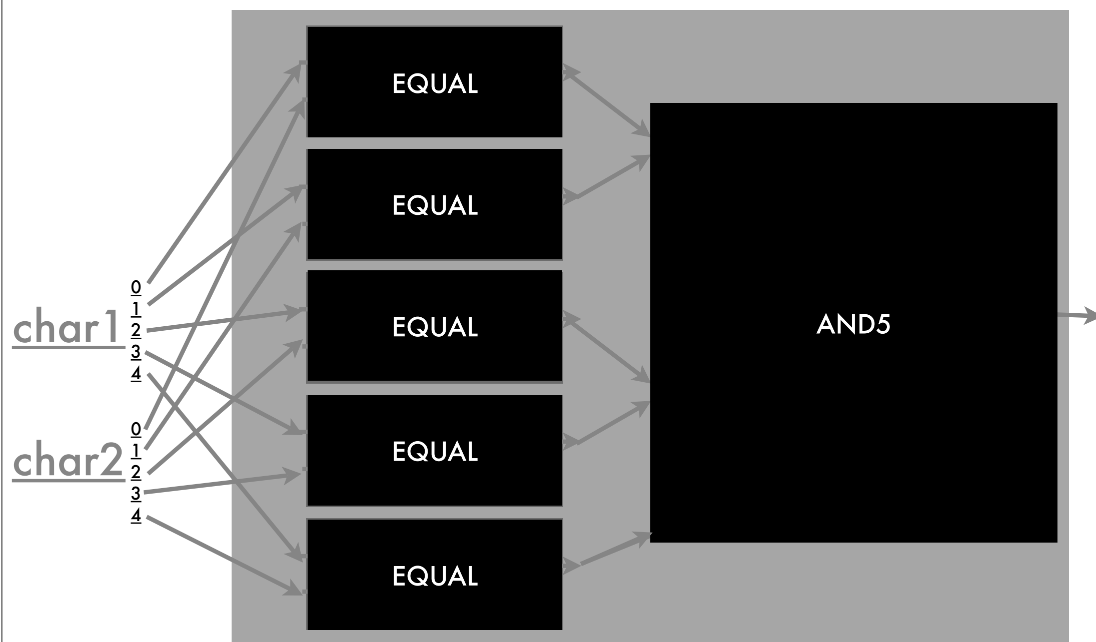
- Output True if either bit1 = True and bit2 = True or bit1 = False and bit2 = False (they are equal).

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.

```
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^{2^n}.

n	2^{2^n}
0	2
1	4
2	16
3	256
4	65536
5	4294967296
6	18446744073709551616
7	340282366920938463463374607431768211456

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

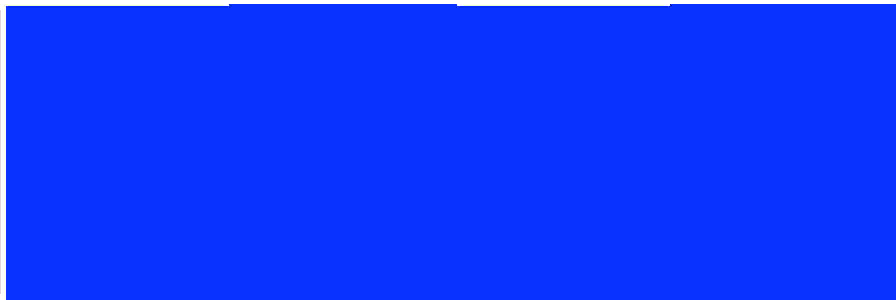
Number Magic

- 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...

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



Reminder: Decimal Notation

	2	**0**	**7**
1000	100	10	1
10^3	10^2	10^1	10^0

- 2x100 +0x10 +7x1 = 207

Binary Notation

1	0	0	1	1	0	1	0
128	64	32	16	8	4	2	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

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.
- 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
- Next power of 2 = 16. 0
- Next power of 2 = 8. 1
  - 11-8=3
- Next power of 2 = 4. 0
- Next power of 2 = 2. 1
  - 3-2=1
- Last power of 2 = 1. 1

1010001011=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

```
   01110011     115
+  10110010    +178
               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: Easy Example

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

Subtraction

```
          1
 011 01    01
 100100101
-10110010
   1110011
```

- 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, $y = -x$.

```
01111111₁
100000000       10101101
-01010011      +01010011
 10101101   (1)00000000
```

(In the left-hand subtraction, the original 100000000 is struck through and 01111111 with a small 1 is written above it.)

Two's Complement

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

```
 01010011    83
 10101100   172 = 255−83
+00000001
 10101101   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

```
      10101101
    x 01010011
      10101101
     10101101
   10101101
+10101101
 1110000010111
```

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.

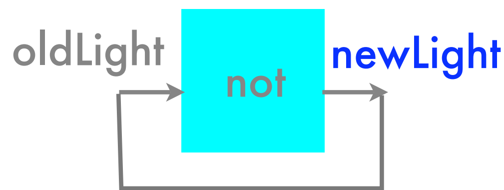
State Machines

- Ok, here's where we are: We can use logic gates to take a set of input values (**True**s and **False**s) 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

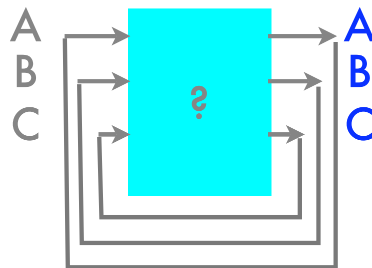
- **Odds/Evens**
- A=not A
- B=False
- C=False
- A
- not A
- A
- not A
- A
- not A
- A
- not A

That's It!?

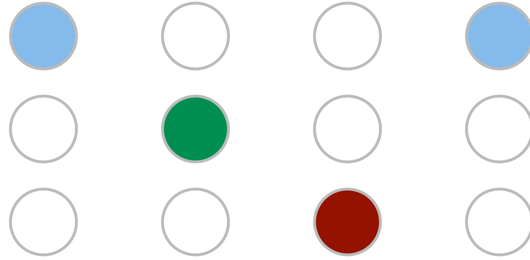
- So, that's a computer.
- Well, actually a computer has more inputs and outputs and the internal logic is more complex.
- But, that's it. So, let's start increasing the complexity to bridge the gap.

"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:



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	[illegible]	[illegible]
0	1	1	1	0	0
1	0	0	1	0	1
1	0	[illegible]	[illegible]	[illegible]	0
1	1	[illegible]	[illegible]	[illegible]	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	
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