ECE 550: Fundamentals of Computer Systems and Engineering

Combinatorial Logic

Admin

- Piazza: Updated from Friday night's rolls
  - Should have everyone now.
- Homework
  - Released now
  - Homework 1: due Sept 13
- Lab
  - 01A
  - Door code on Piazza

Last time….

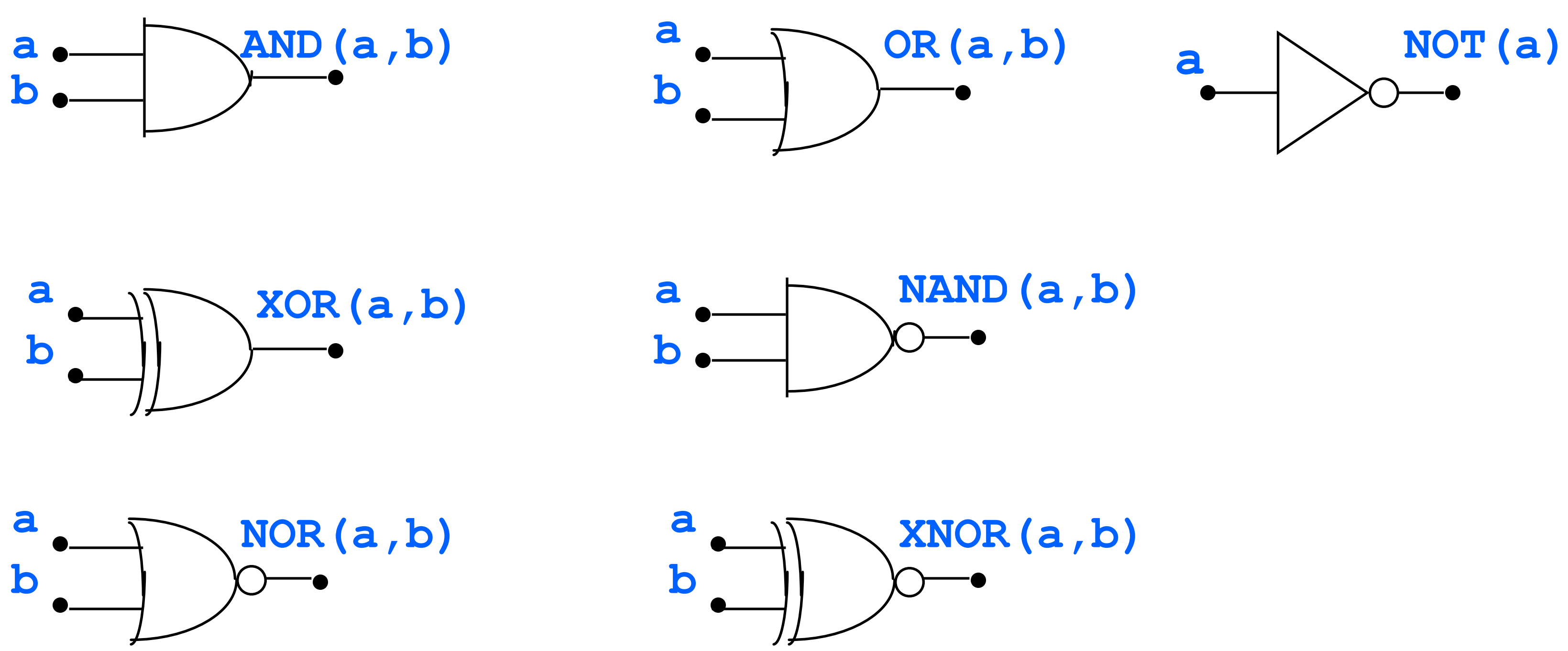
- Who can remind us what we talked about last time?

Last time….

- Who can remind us what we talked about last time?
  - Electric circuit basics
    - Vcc = 1
    - Ground = 0
  - Transistors
    - PMOS
    - NMOS
  - Gates
    - Complementary PMOS + NMOS
    - Output is logical function of inputs

Boolean Gates

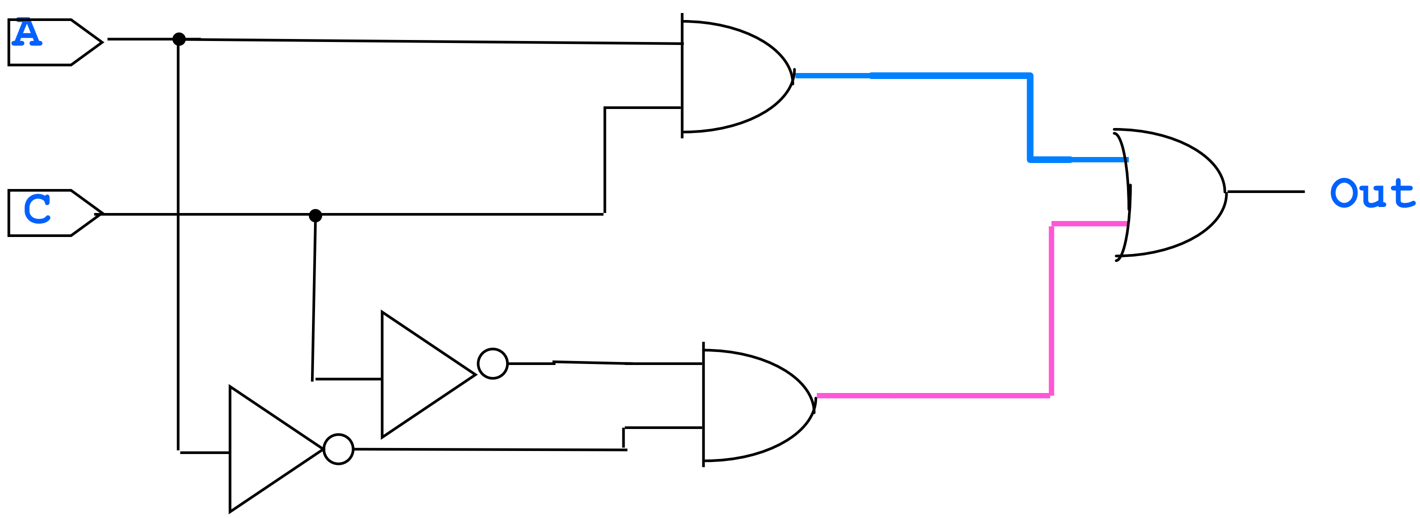
- Saw these gates
  - Mnemonic to remember them



Boolean Functions, Gates and Circuits

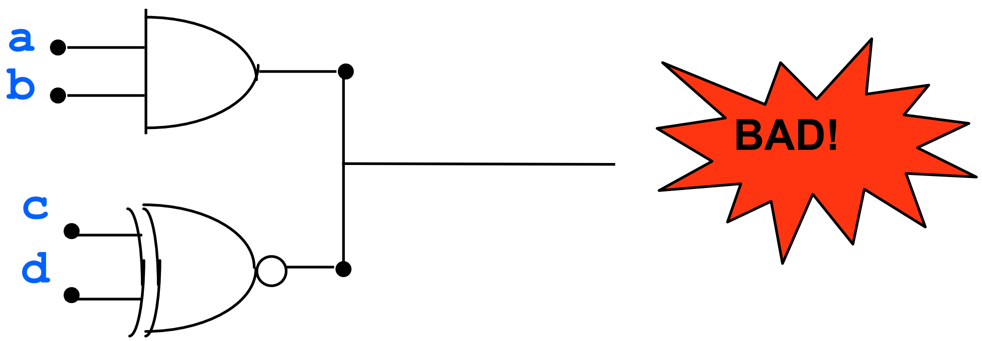
- Circuits are made from a network of gates.

`(!A & !C)|(A & C)`



A few more words about gates

- Gates have inputs and outputs
  - If you try to hook up two outputs, get short circuit
    - (Think of the transistors each gate represents)



  - If you don't hook up an input, it behaves kind of randomly

(also not good, but not set-your-chip-on-fire bad)

Let's Make a Useful Circuit

- Pick between 2 inputs (called 2-to-1 MUX)
  - Short for multiplexor
- What might we do first?

Let's Make a Useful Circuit

- Pick between 2 inputs (called 2-to-1 MUX)
  - Short for multiplexor
- What might we do first?
  - Make a truth table?
    - S is selector:
      - S=0, pick A
      - S=1, pick B

A	B	S	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Let's Make a Useful Circuit

- Pick between 2 inputs (called 2-to-1 MUX)
  - Short for multiplexor
- What might we do first?
  - Make a truth table?
    - S is selector:
      - S=0, pick A
      - S=1, pick B
- Next: **sum-of-products**
  - Always works to find formula

A	B	S	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Sum of Products

- Find the rows where the output is 1

A	B	S	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	**1**	**1**	**1**
1	**0**	**0**	**1**
1	0	1	0
1	**1**	**0**	**1**
1	**1**	**1**	**1**

Sum of Products

- Find the rows where the output is 1
- Write a formula that exactly specifies each row

	A	B	S	Output
	0	0	0	0
	0	0	1	0
	0	1	0	0
!A & B & S	**0**	**1**	**1**	**1**
A & !B & !S	**1**	**0**	**0**	**1**
	1	0	1	0
A & B & !S	**1**	**1**	**0**	**1**
A & B & S	**1**	**1**	**1**	**1**

Sum of Products

- Find the rows where the output is 1
- Write a formula that exactly specifies each row
- OR these all together
  - Possible ways to get 1.

	A	B	S	Output
	0	0	0	0
	0	0	1	0
	0	1	0	0
!A & B & S →	**0**	**1**	**1**	**1**
A & !B & !S →	**1**	**0**	**0**	**1**
	1	0	1	0
A & B & !S →	**1**	**1**	**0**	**1**
A & B & S →	**1**	**1**	**1**	**1**

Let’s Make a Useful Circuit

- Pick between 2 inputs (called 2-to-1 MUX)
  - Short for multiplexor
- What might we do first?
  - Make a truth table?
    - S is selector:
      - S=0, pick A
      - S=1, pick B
- Next: **sum-of-products**

```
(!A & B & S) |
(A & !B & !S) |
(A & B & !S ) |
(A & B & S)
```

A	B	S	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Let’s Make a Useful Circuit

- Pick between 2 inputs (called 2-to-1 MUX)
  - Short for multiplexor
- What might we do first?
  - Make a truth table?
    - S is selector:
      - S=0, pick A
      - S=1, pick B
- Next: sum-of-products
- Simplify!

A	B	S	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Simplifying The SOP Formula

- Simplifying this formula:

```
(!A & B & S) |
(A & !B & !S) |
(A & B & !S ) |
(A & B & S)
```

B doesn't matter

Simplifying The SOP Formula

- Simplifying this formula:

```
(!A & B & S) |
(A & !S) |
(A & B & S)
```

A doesn't matter

Simplifying The SOP Formula

- Simplifying this formula:

```
(A & !S) |
(B & S)
```

Let’s Make a Useful Circuit

- Pick between 2 inputs (called 2-to-1 MUX)
  - Short for multiplexor
- What might we do first?
  - Make a truth table?
    - S is selector:
      - S=0, pick A
      - S=1, pick B
- Next: sum-of-products
- Simplify

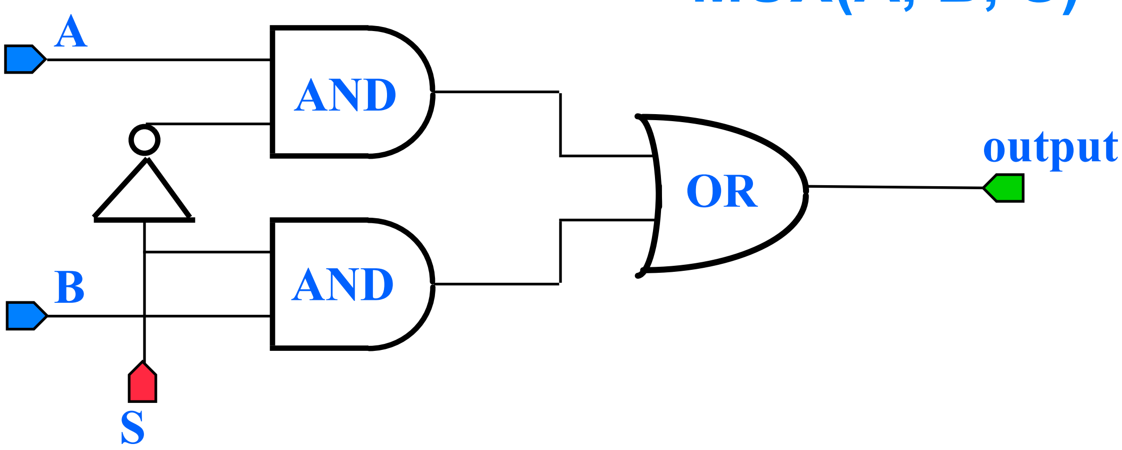
```
(A & !S) |
(B &  S)
```

A	B	S	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

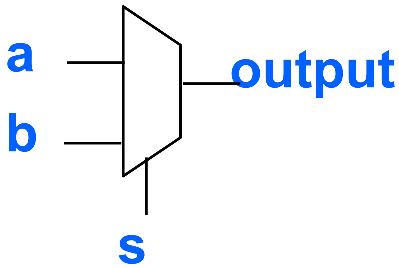
Circuit Example: 2x1 MUX

Draw it in gates:

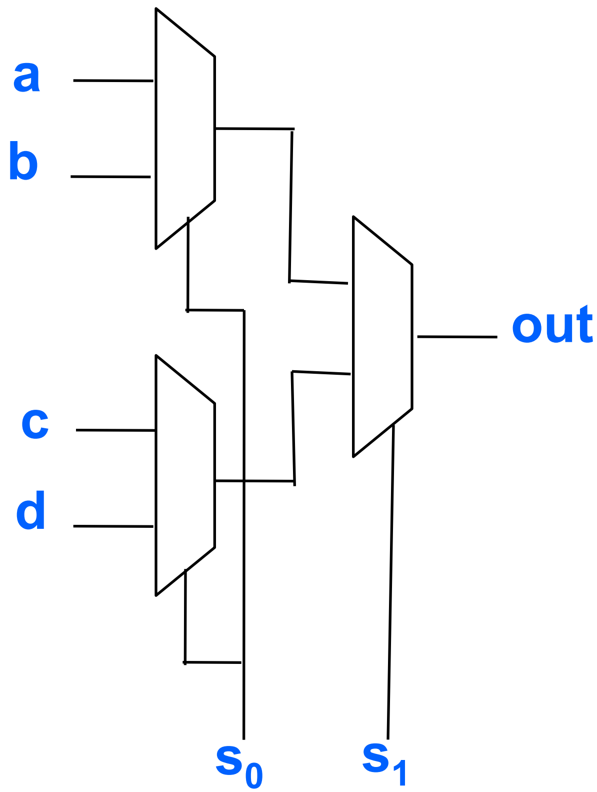
MUX(A, B, S) = (A & !S) | (B & S)

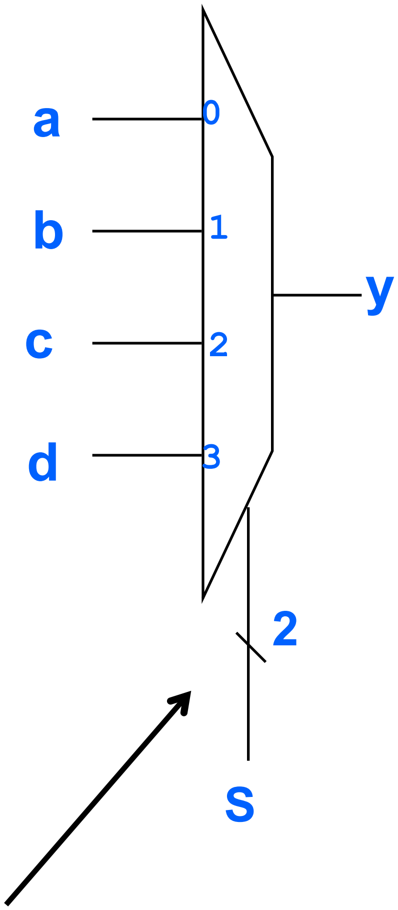


So common, we give it
its own symbol:



Example 4x1 MUX





The / 2 on the wire means "2 bits"

Boolean Function Simplification

- Boolean expressions can be simplified by using the following rules (bitwise logical):
  - `A & A = A` `A | A = A`
  - `A &  0 = 0` `A | 0 = A`
  - `A & 1 = A` `A | 1 = 1`
  - `A & !A = 0` `A | !A = 1`
  - `!!A  = A`
  - & and | are both commutative and associative
  - & and | can be distributed: `A & (B | C) = (A & B) | (A & C)`
  - & and | can be subsumed: `A | (A & B) = A`
  - Can typically just let synthesis tools do this dirty work, but good to know

DeMorgan’s Laws

- Two (less obvious) Laws of Boolean Algebra:
  - Let us push negations inside, flipping & and |

```
!(A & B) = (!A) | (!B)
```

```
!(A | B) = (!A) & (!B)
```

Alluded to these last time

Very good rules to know in general

Next piece of logic: Adder

- Computers do one thing: math
  - And they do it well/fast
  - Fundamental rule of computation: “Everything is a number”
    - Computers can only work with numbers
    - Represent things as numbers

Next: logic to work with numbers

- Computers do one thing: math
  - And they do it well/fast
  - Fundamental rule of computation: “Everything is a number”
    - Computers can only work with numbers
    - Represent things as numbers
  - Specifically: good at binary math
    - Base 2 number system: matches circuit voltages
      - 1 (Vcc)
      - 0 (Ground)
    - Use fixed sized numbers
      - How many **bits**
  - Quick primer on binary numbers/math
    - Then how to make circuits for it

Numbers for computers

- We usually use base 10:
  - $12345 = 1 * 10^4 + 2 * 10^3 + 3 * 10^2 + 4 * 10^1 * 5 * 10^0$
  - Recall from third grade: 1's place, 10's place, 100's place…
    - Yes, we are going to re-cover 3rd grade math, but in binary
  - What is the biggest digit that can go in any place?
- Base 2:
  - 1's place, 2's place, 4's place, 8's place, ….
  - What is the biggest digit that can go in any place?

Binary continued:

- Binary Number Example: 101101
  - Take a second and figure out what number this is

Binary continued:

- Binary Number Example: 101101
  - Take a second and figure out what number this is

1 in 32's place = 32

0 in 16's place

1 in 8's place = 8

1 in 4's place = 4

0 in 2's place

1 in 1's place = 1

—-

45

Converting Numbers

- Converting Decimal to Binary

  Suppose I want to convert 4872 to binary

  Think for a second about how to do this

Converting Numbers

- Converting Decimal to Binary

  Suppose I want to convert 4872 to binary

  Think for a second about how to do this

$$\begin{array}{r} 4872 \\ \underline{-\ 4096} \\ 776 \end{array}$$

4096	1
2048	
1024	
512	
256	
128	
64	
32	
16	
8	
4	
2	
1	

Converting Numbers

- Converting Decimal to Binary

  Suppose I want to convert 4872 to binary

  Think for a second about how to do this

```
  4872
- 4096
------
   776
-  512
------
   264
```

4096	1
2048	0
1024	0
512	1
256	
128	
64	
32	
16	
8	
4	
2	
1	

Converting Numbers

- Converting Decimal to Binary

    Suppose I want to convert 4872 to binary

    Think for a second about how to do this

```
  4872
- 4096
------
   776
-  512
------
   264
-  256
------
     8
```

4096	1
2048	0
1024	0
512	1
256	1
128	
64	
32	
16	
8	
4	
2	
1	

Converting Numbers

- Converting Decimal to Binary

  Suppose I want to convert 4872 to binary

  Think for a second about how to do this

$$
\begin{array}{r}
4872 \\
\underline{-\ 4096} \\
776 \\
\underline{-\ \ 512} \\
264 \\
\underline{-\ \ 256} \\
8 \\
\underline{-\ \ \ \ 8} \\
0
\end{array}
$$

4096	1
2048	0
1024	0
512	1
256	1
128	0
64	0
32	0
16	0
8	1
4	
2	
1	

Converting Numbers

- Converting Decimal to Binary

  Suppose I want to convert 4872 to binary

  Think for a second about how to do this

```
   4872
 - 4096
 ------
    776
 -  512
 ------
    264
 -  256
 ------
      8
 -    8
 ------
      0
```

4096	1
2048	0
1024	0
512	1
256	1
128	0
64	0
32	0
16	0
8	1
4	0
2	0
1	0

Hexadecimal: Convenient shorthand for

- Binary is unwieldy to write
  - 425,000 decimal = 1100111110000101000 binary
  - Generally about 3x as many binary digits as decimal
  - Converting (by hand) takes some work and thought
- Hexadecimal (aka "hex")—base 16—is convenient:
  - Easy mapping to/from binary
  - Same or fewer digits than decimal
  - 425,000 decimal = 0x67C28
  - Generally write "0x" on front to make clear "this is hex"
  - Digits from 0 to 15, so use A—F for 10—15.

Binary ⇔ Hex

- Binary ⇔ Hex conversion is straightforward
  - Every 4 binary bits = 1 hex digit
  - If # of bits not a multiple of 4, add implicit 0s on left as needed

0000 ⇔ 0	1000 ⇔ 8
0001 ⇔ 1	1001 ⇔ 9
0010 ⇔ 2	1010 ⇔ A
0011 ⇔ 3	1011 ⇔ B
0100 ⇔ 4	1100 ⇔ C
0101 ⇔ 5	1101 ⇔ D
0110 ⇔ 6	1110 ⇔ E
0111 ⇔ 7	1111 ⇔ F

Binary ⇔ Hex

- **1100111110000101000**
- **6 7 C 2 8**

Binary Math : Addition

- Suppose we want to add two numbers:

```
      00011101
+     00101011
```

- How do we do this?

Binary Math : Addition

- Suppose we want to add two numbers:

```
    00011101          695
+   00101011        + 232
```

- How do we do this?
  - Let's revisit decimal addition
  - Think about the process as we do it

Binary Math : Addition

- Suppose we want to add two numbers:

```
    00011101          695
+   00101011        + 232
                        7
```

- First add one's digit 5+2 = 7

Binary Math : Addition

- Suppose we want to add two numbers:

```
       00011101            1
   +   00101011           695
                        + 232
                           27
```

- First add one's digit 5+2 = 7
- Next add ten's digit 9+3 = 12 (2 carry a 1)

Binary Math : Addition

- Suppose we want to add two numbers:

```
                          1
     00011101            695
+    00101011          + 232
                         927
```

- First add one's digit 5+2 = 7
- Next add ten's digit 9+3 = 12 (2 carry a 1)
- Last add hundred's digit 1+6+2 = 9

Binary Math : Addition

- Suppose we want to add two numbers:

```
      00011101
  +   00101011
```

- Back to the binary:
- First add 1's digit 1+1 = ...?

Binary Math : Addition

- Suppose we want to add two numbers:

```
          1
    00011101
 +  00101011
    --------
           0
```

- Back to the binary:
- First add 1's digit 1+1 = 2 (0 carry a 1)

Binary Math : Addition

- Suppose we want to add two numbers:

```
           11
     00011101
 +   00101011
 -------------
           00
```

- Back to the binary:
- First add 1's digit 1+1 = 2 (0 carry a 1)
- Then 2's digit: 1+0+1 =2 (0 carry a 1)
- You all finish it out….

Binary Math : Addition

- Suppose we want to add two numbers:

```
      111111
     00011101    = 29
 +   00101011    = 43
 ------------
     01001000    = 72
```

- Can check our work in decimal

Negative Numbers

- May want negative numbers too!
- Many ways to represent negative numbers:
  - Sign/magnitude
  - Biased
  - 1's complement
  - 2's complement

2's Complement Integers

- To negate, flip bits, add 1:
  - 1's complement + 1
- Pros:
  - Easy to compute with
  - One representation of 0
- Cons:
  - More complex negation
  - Extra negative number (-8)
- Ubiquitous choice

0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	-8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

Binary Math : Addition

- Revisit binary math for a minute:

```
      01011101
  +   01101011
```

Binary Math : Addition

- What about this one:

```
      1111111
      01011101   =   93
  +   01101011   =  107
      --------
      11001000   =  -56
```

- But… that can't be right?
  - What do you expect for the answer?
  - What is it in 8-bit signed 2's complement?
-

Integer Overflow

- Answer should be 200
  - Not representable in 8-bit signed representation
  - No right answer
- Called Integer Overflow
  - Signed addition: CI != CO of last bit
  - Unsigned addition: CO != 0 of last bit
- Can detect in hardware
  - Signed: XOR CI and CO of last bit
  - Unsigned: CO of last bit
  - What processor does: depends

Subtraction

- 2’s complement makes subtraction easy:
  - Remember: A - B = A + (-B)
  - And: -B = ~B + 1

    ↑ that means flip bits (“not”)
  - So we just flip the bits and start with CI = 1
  - Fortunate for us: makes circuits easy (next time)

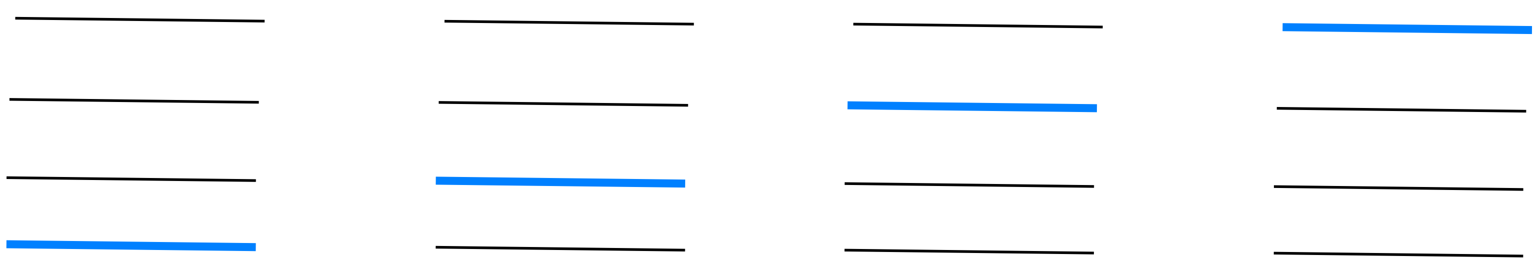
```
                          1
   0110101    ->    0110101
 - 1010010        + 0101101
 ---------        ---------
```

Signed and Unsigned Ints

- Most programming languages support two int types
  - Signed: negative and positive
  - Unsigned: positive only, but can hold larger positive numbers
- Addition and subtraction:
  - Same, except overflow detection
  - x86: one add instruction, sets two different flags for overflows
- Inequalities
  - Different operations for signed/unsigned
  - Can someone give an example? (Let’s say 4-bit numbers)

One hot representation

- Binary representation convenient for math
- Another representation:
  - One hot: one wire per number
  - At any time, one wire = 1, others = 0



  - Wow, that is one hot representation!
  - Very convenient in many cases (e.g., homework 1)

Converting to/from one hot

- Converting from 2^N bits one hot to N bits binary=**encoder**
  - E.g., "an 8-to-3 encoder"
- Converting from N bits binary to 2^N bits one hot=**decoder**
  - E.g., "a 4-to-16 decoder"
    (which may be quite useful on hwk1)

Lets build a 4-to-2 encoder

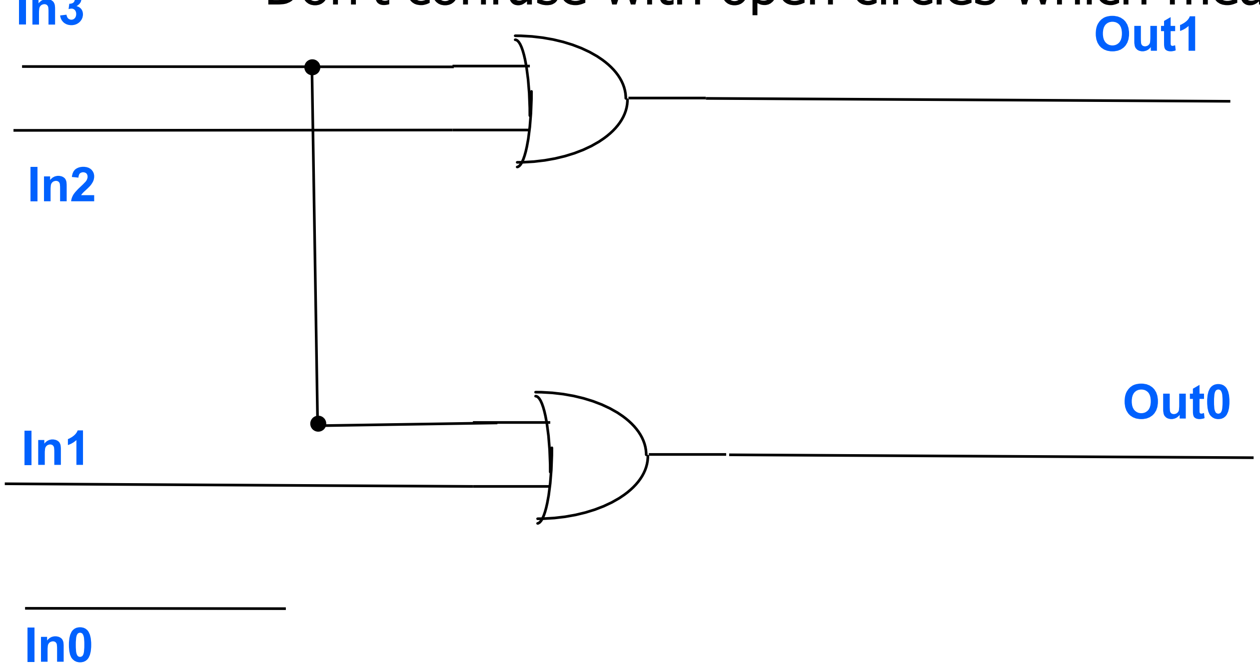
- Start with a truth table
  - Input constrained to 1-hot: don't care about invalid inputs
    - Can do anything we want

In0	In1	In2	In3	Out1	Out0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

- Simplest formulas:
  - Out0 = In1 or In3 [alternatively: Out0 = In0 nor In2]
  - Out1 = In2 or In3 [alternatively: Out1 = In0 nor In1]

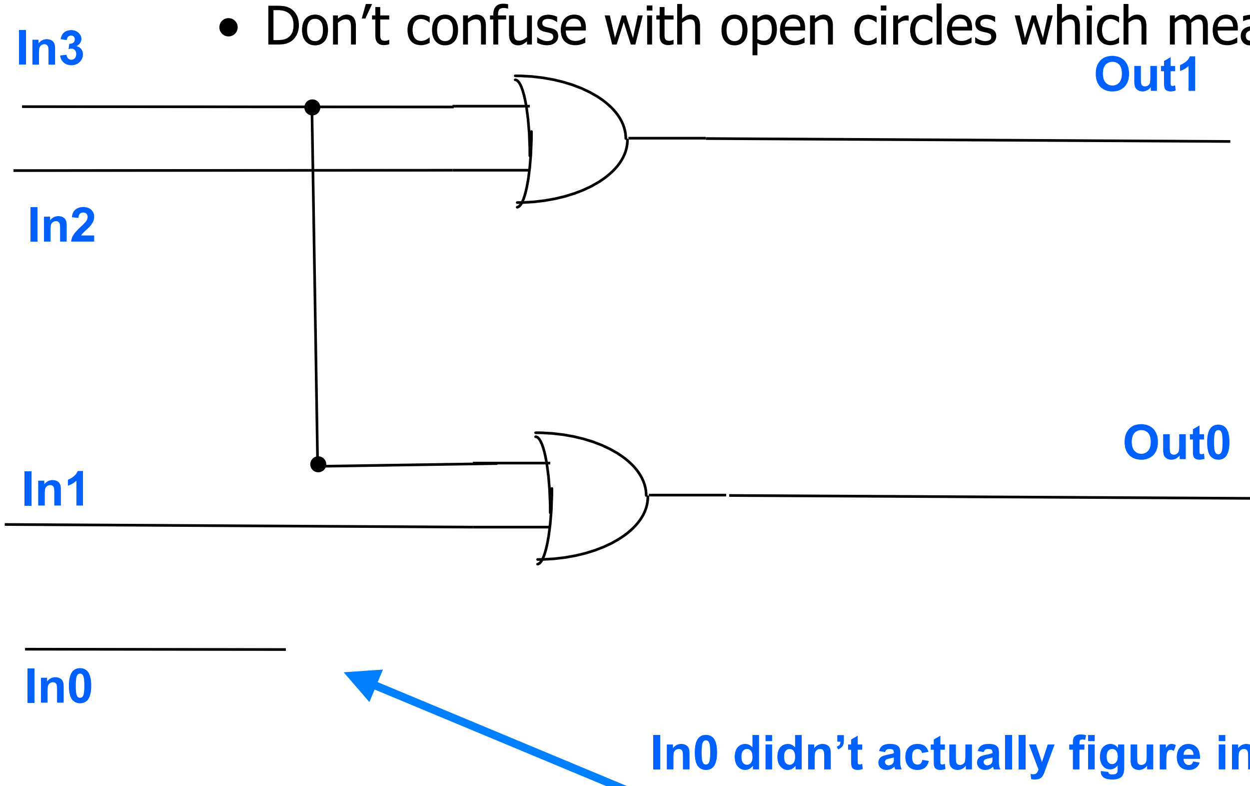
4-to-2 encoder

- Our 4-to-2 encoder
  - Note: the dots here show connections
    - Don’t confuse with open circles which mean NOT



4-to-2 encoder

- Our 4-to-2 encoder
  - Note: the dots here show connections
    - Don’t confuse with open circles which mean NOT



In0 didn’t actually figure into the logic. That’s ok
Synthesis will eliminate it if its not used elsewhere…
And whatever logic creates it, etc..

Lets build a 2-to-4 decoder

- Start with a truth table
  - Now input unconstrained

In1	In0	Out0	Out1	Out2	Out3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Now SOP more useful, do for each of the 4 outputs:

Lets build a 2-to-4 decoder

- Start with a truth table
  - Now input unconstrained

In1	In0	Out0	Out1	Out2	Out3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Now SOP more useful, do for each of the 4 outputs:

Out0 = (Not In1) and (Not In0)

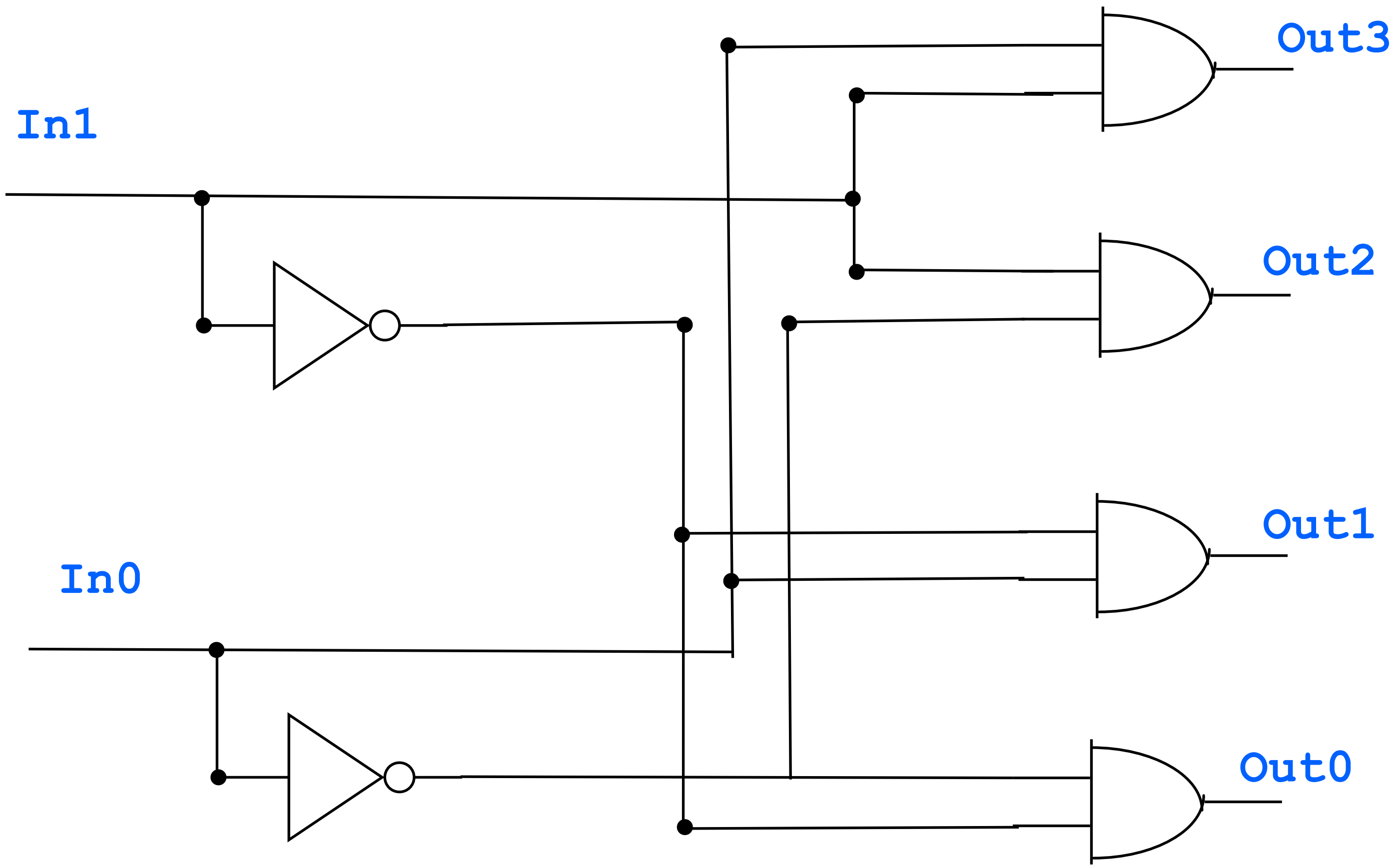
Out1 = (Not In1) and In0

Out2 = In1 and (Not In0)

Out3 = In1 and In0

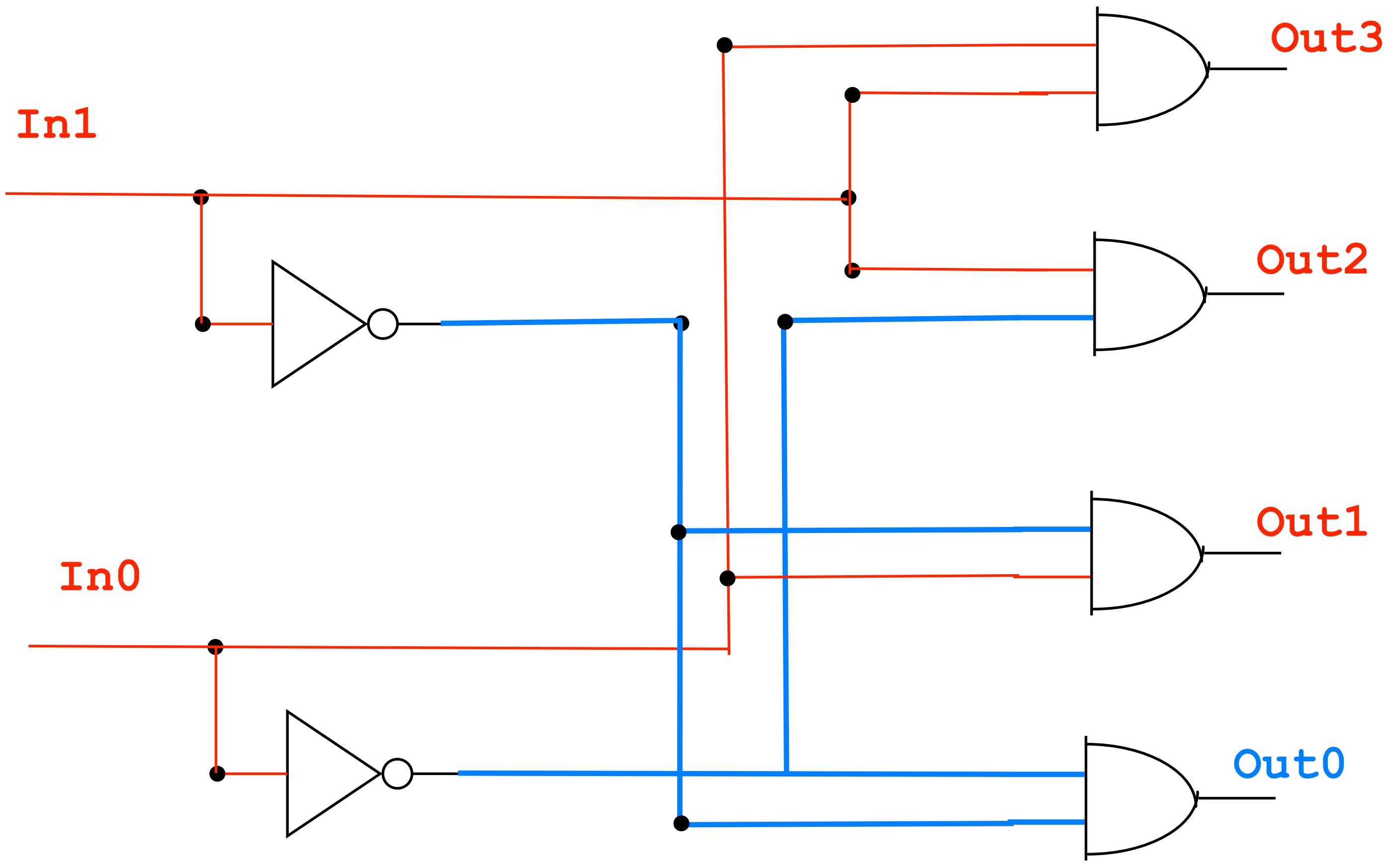
2-to-4 decoder

- 2-to-4 decoder



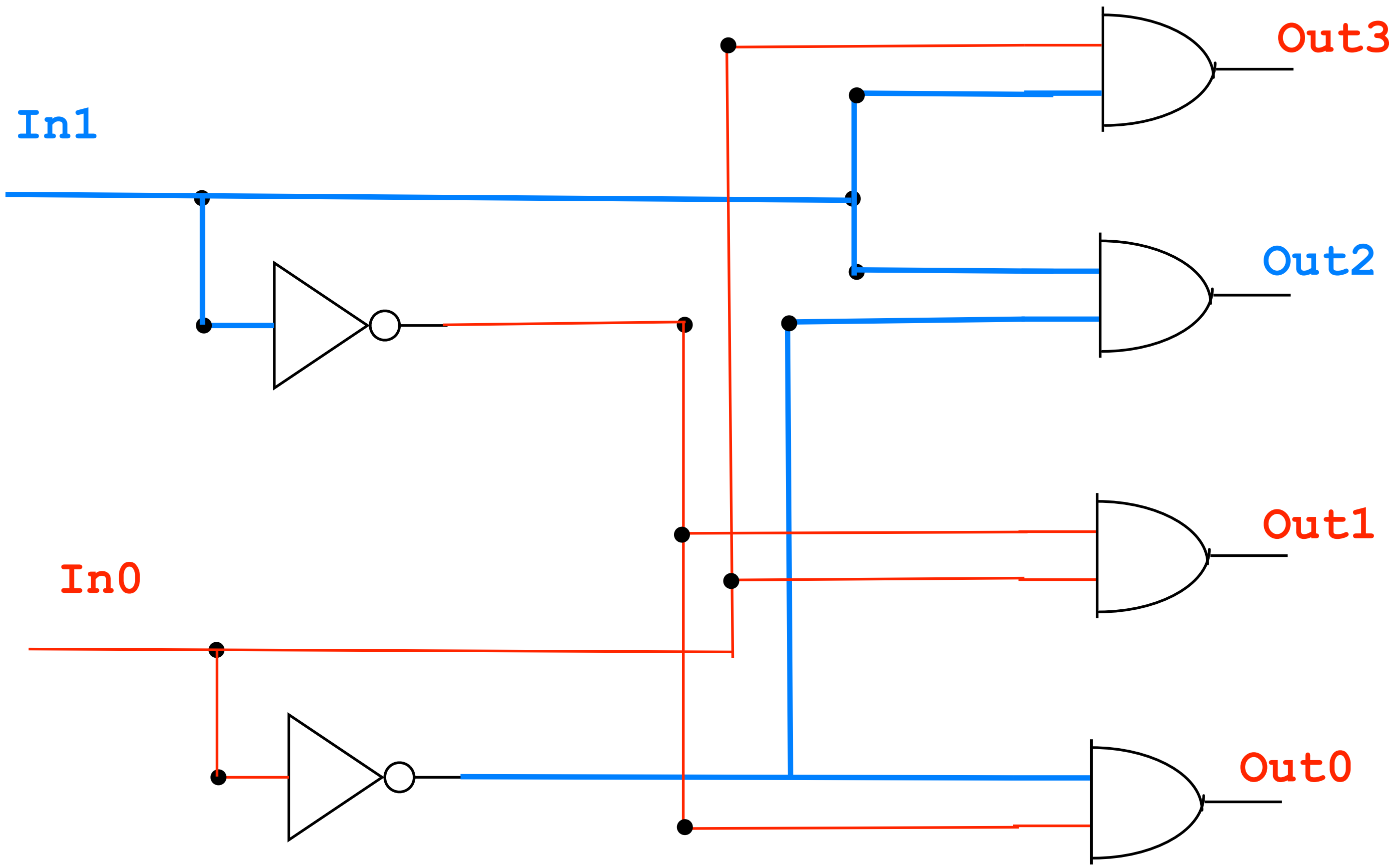
2-to-4 decoder

- 2-to-4 decoder



2-to-4 decoder

- 2-to-4 decoder



Delays

- Mentioned before: switching not instant
  - Not going to try to calculate delays by hand (tools can do)
  - But good to know where delay comes from, to tweak/improve
- Gates:
  - Switching the transistors in gates takes time
  - More gates (in series) = more delay
- Fan-out: how many gates the output drives
  - Related to capacitance
  - High fan-out = slow
  - Sometimes better to replicate logic to reduce its fan-out
- Wire delay:
  - Signals take time to travel down wires

Wrap Up

- Combinatorial Logic
  - Putting gates together
  - Sum-of-products
  - Simplification
  - Muxes, Encoders, Decoders
- Number Representations
  - One Hot
  - 2's complement