ECE/CS 250: Computer Architecture

Basics of Logic Design:
Boolean Algebra, Logic Gates

Benjamin Lee

Slides based on those from Alvin Lebeck, Daniel Sorin, Andrew Hilton, Amir Roth, Gershon Kedem

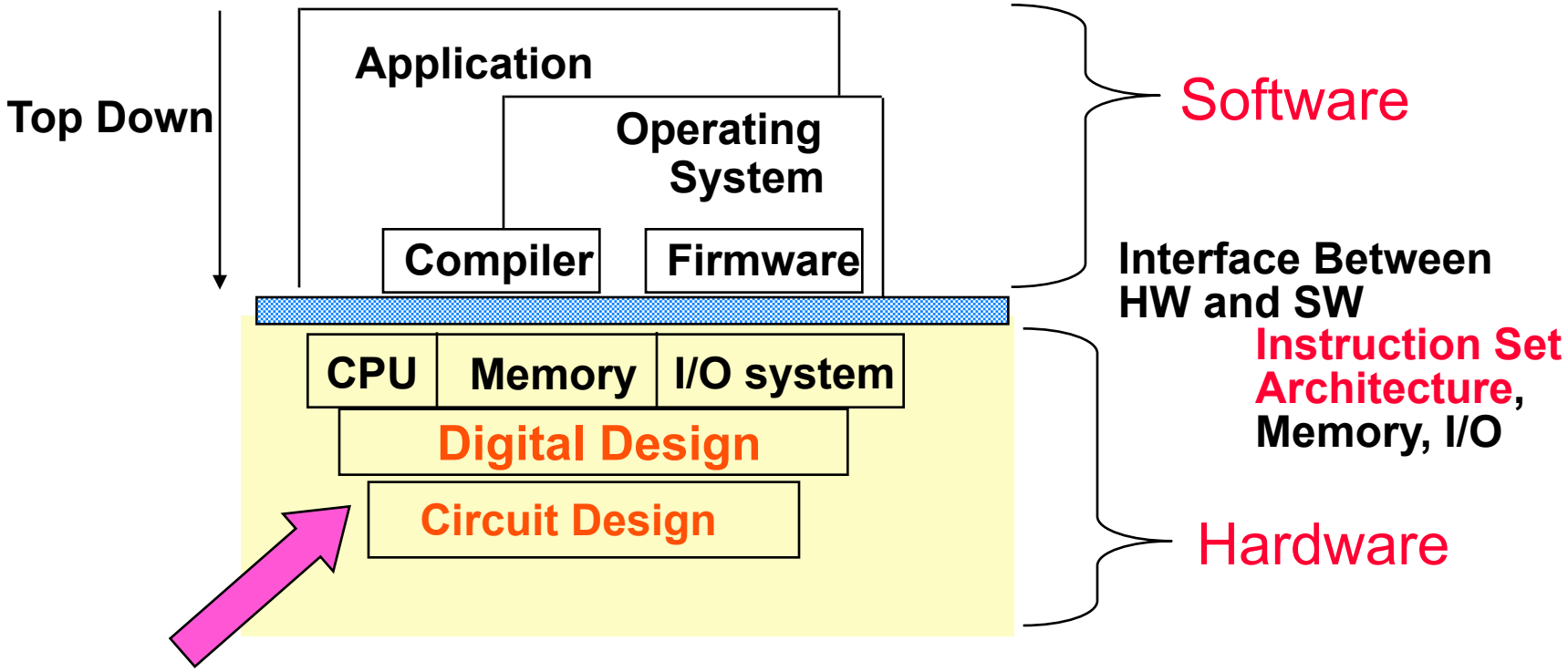
Admin

- **Resource: Pragmatic Logic by William Eccles**
  - **In Sakai Resources and linked off web page.**
  - **Skim for what you need, way too much in there**

- **This material is covered in MUCH greater depth in ECE/CS 350 – please take ECE/CS 350 if you want to learn enough digital design to build your own processor**

- **Download Logisim before Friday's recitation**
  - **http://ozark.hendrix.edu/~burch/logisim/download.html**

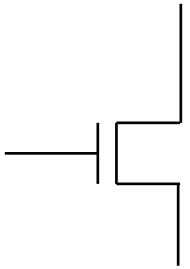
What We've Done, Where We're Going



(Almost) Bottom UP to CPU

Computer = Machine That Manipulates Bits

- Everything is in binary (bunches of 0s and 1s)
  - Instructions, numbers, memory locations, etc.
- Computer is a machine that operates on bits



- Computers physically made of transistors
  - Electrically controlled switches
- We can use transistors to build logic
  - E.g., if this bit is a 0 and that bit is a 1, then set some other bit to be a 1
  - E.g., if the first 5 bits of the instruction are 10010 then set this other bit to 1 (to tell the adder to subtract instead of add)

How Many Transistors Are We Talking About?

Pentium III

- **Processor Core 9.5 Million Transistors**
- **Total: 28 Million Transistors**

Pentium 4

- **Total: 42 Million Transistors**

Core2 Duo (two processor cores)

- **Total: 290 Million Transistors**

Core2 Duo Extreme (4 processor cores, 8MB cache)

- **Total: 590 Million Transistors**

Core i7 with 6-cores

- **Total: 2.27 Billion Transistors**

How do they design such a thing? Carefully!

Abstraction!

- Use of abstraction (key to design of any large system)
  - Put a few (2-8) transistors into a logic gate (or, and, xor, …)
  - Combine gates into logical functions (add, select,….)
  - Combine adders, shifters, etc., together into modules

    Units with well-defined interfaces for large tasks: e.g., decode
  - Combine a dozen of those into a core…
  - Stick 4 cores on a chip…

You are here:

- Use of abstraction (key to design of any large system)
  - Put a few (2-8) transistors into a logic gate
  - Combine gates into logical functions (add, select,….)
  - Combine adders, muxes, etc together into modules
    Units with well-defined interfaces for large tasks: e.g., decode
  - Combine a dozen of those into a core…
  - Stick 4 cores on a chip…

Boolean Algebra

- **First step to logic: Boolean Algebra**
  - **Formal apporoach for manipulating of True / False (1/0)**
  - **After all: everything is just 1s and 0s**

- **Boolean Functions:**
  - **Given inputs (variables): A, B, C, P, Q…**
  - **Compute outputs using logic operators, such as:**
- **NOT:** !A **(=** ~A **=** $\bar{A}$ = A')
- **AND:** A&B **(=** A·B **=** A*B = AB **=** A∧B) = A&&B in C/Java
- **OR:** A | B **(=** A+B **=** A ∨ B**) =** A || B in C/Java
- **XOR:** A ^ B (= A ⊕ B)
- NAND, NOR, XNOR, Etc.

Truth Tables

- **Can represent as Truth Table: shows outputs for all inputs**

a	NOT(a)
0	1
1	0

Truth Tables

- **Can represent as truth table: shows outputs for all inputs**

a	NOT(a)
0	1
1	0

a	b	AND(a,b)
0	0	0
0	1	0
1	0	0
1	1	1

Truth Tables

- **Can represent as truth table: shows outputs for all inputs**

a	NOT(a)
0	1
1	0

a	b	AND(a,b)
0	0	0
0	1	0
1	0	0
1	1	1

a	b	OR(a,b)
0	0	0
0	1	1
1	0	1
1	1	1

Truth Tables

- **Can represent as truth table: shows outputs for all inputs**

a	NOT(a)
0	1
1	0

a	b	AND(a,b)
0	0	0
0	1	0
1	0	0
1	1	1

a	b	OR(a,b)
0	0	0
0	1	1
1	0	1
1	1	1

a	b	XOR(a,b)
0	0	0
0	1	1
1	0	1
1	1	0

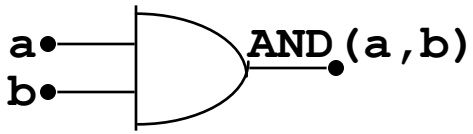
a	b	XNOR(a,b)
0	0	1
0	1	0
1	0	0
1	1	1

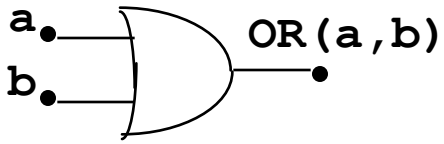
a	b	NOR(a,b)
0	0	1
0	1	0
1	0	0
1	1	0

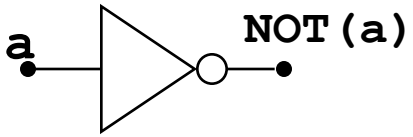
Boolean Gates (More Later)

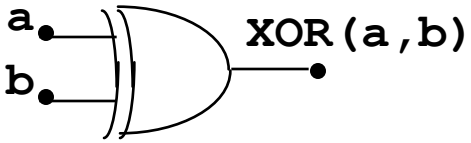
- **Gates** are electronic devices that implement simple Boolean functions (building blocks of hardware)

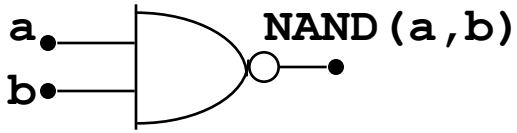
Examples

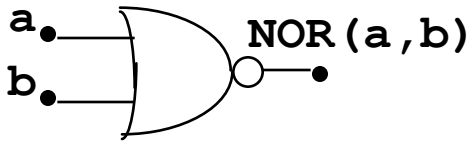


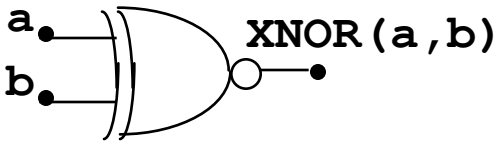












Any Inputs, Any Outputs

- Can have any # of inputs, any # of outputs
- Can have arbitrary functions:

a	b	c	f_1	f_2
0	0	0	0	1
0	0	1	1	1
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	1	0	0	1
1	1	1	1	1

Let's Write a Truth Table for a Function…

- **Example:**
  (A & B) | !C

Start with Empty TT

Column Per Input

Column Per Output

A	B	C	Output

Let's write a Truth Table for a function…

- **Example:**
  (A & B) | !C

Start with Empty TT

- **Column Per Input**
- **Column Per Output**

Fill in Inputs

- **Counting in Binary**

A	B	C	Output
0	0	0	

Let's write a Truth Table for a function…

- **Example:**
  (A & B) | !C

Start with Empty TT

Column Per Input

Column Per Output

Fill in Inputs

Counting in Binary

A	B	C	Output
0	0	0	
0	0	1	

Let's write a Truth Table for a function…

- **Example:**
  (A & B) | !C

Start with Empty TT

Column Per Input

Column Per Output

Fill in Inputs

Row per set of input values

Counting in Binary

A	B	C	Output
0	0	0	
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Let's write a Truth Table for a function…

- **Example:**
  (A & B) | !C

Start with Empty TT

- **Column Per Input**
- **Column Per Output**

Fill in Inputs

- **Row per set of input values**
- **Counting in Binary**

Compute Output for reach row

(0 & 0) | !0 = 0 | 1 = 1

A	B	C	Output
0	0	0	1
0	0	1	
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Let's write a Truth Table for a function…

- **Example:**
  (A & B) | !C

Start with Empty TT

- **Column Per Input**
- **Column Per Output**

Fill in Inputs

- **Row per set of input values**
- **Counting in Binary**

Compute Output for each row

(0 & 0) | !1 = 0 | 0 = 0

A	B	C	Output
0	0	0	1
0	0	1	0
0	1	0	
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Let's write a Truth Table for a function…

- **Example:**
  (A & B) | !C

Start with Empty TT

- **Column Per Input**
- **Column Per Output**

Fill in Inputs

- **Row per set of input values**
- **Counting in Binary**

Compute Output for each row

(0 & 1) | !0 = 0 | 1 = 1

A	B	C	Output
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	
1	0	0	
1	0	1	
1	1	0	
1	1	1	

Let's write a Truth Table for a function…

- **Example:**
  (A & B) | !C

Start with Empty TT

- **Column Per Input**
- **Column Per Output**

Fill in Inputs

- **Row per set of input values**
- **Counting in Binary**

Compute Output for each row

A	B	C	Output
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

You try one…

- **Try one yourself:**
  (!A | B) & !C

You try one…

- **Try one yourself:**
  (!A | B) & !C

Answer:

A	B	C	Output
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

Suppose I turn it around…

- **Given a Truth Table, find the formula?**

Hmmm..

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Suppose I turn it around…

- **Given a Truth Table, find the formula?**

Hmmm …

Could write down every "true" case

Then OR together:

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

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Suppose I turn it around…

- **Given a Truth Table, find the formula?**

Hmmm..

Could write down every "true" case

Then OR together:

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

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Suppose I turn it around…

- **Given a Truth Table, find the formula?**

Hmmm..

Could write down every "true" case

Then OR together:

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

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Suppose I turn it around…

- **This approach: "sum of products"**
  - **Works every time**
  - **Result is right…**
  - **But really ugly**

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

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Suppose I turn it around…

- **This approach: "sum of products"**
  - **Works every time**
  - **Result is right…**
  - **But really ugly**

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

Could just be (A & B) here ?

A	B	C	Output
0	**0**	**0**	**1**
0	**0**	**1**	**1**
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	**1**	**0**	**1**
1	**1**	**1**	**1**

Suppose I turn it around…

- **This approach: "sum of products"**
  - **Works every time**
  - **Result is right…**
  - **But really ugly**

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

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Suppose I turn it around…

- This approach: "sum of products"
  - Works every time
  - Result is right…
  - But really ugly

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

Could just be (!A & !B) here

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Suppose I turn it around…

- This approach: "sum of products"
  - Works every time
  - Result is right…
  - But really ugly

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

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Suppose I turn it around…

- **This approach: "sum of products"**
  - **Works every time**
  - **Result is right…**
  - **But really ugly**

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

Looks nicer…

Can we do better?

A	B	C	Output
0	**0**	**0**	**1**
0	**0**	**1**	**1**
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Suppose I turn it around…

- This approach: "sum of products"
  - Works every time
  - Result is right…
  - But really ugly

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

This has a lot in common:

!A & (something)

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Suppose I turn it around…

- **This approach: "sum of products"**
  - **Works every time**
  - **Result is right…**
  - **But really ugly**

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

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Just did some of these by intuition.. but

- Somewhat intuitive approach to simplifying
- This is math, so there are formal rules
  - Just like “regular” algebra

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`

DeMorgan's Laws

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

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

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

  - **You should try this at home – build truth tables for both the left and right sides and see that they're the same**

Suppose I turn it around…

- One more simplification on early example:

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

=

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

A	B	C	Output
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Another Simplification Example

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

Simplification Example:

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

DeMorgan's

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

Simplification Example:

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

DeMorgan's

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

Double Negation Elimination

```
A & (A & (B | C))
```

Simplification Example:

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

DeMorgan's

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

Double Negation Elimination

```
A & (A & (B | C))
```

Associativity of &

```
(A & A) & (B | C)
```

Simplification Example:

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

DeMorgan's

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

Double Negation Elimination

```
A & (A & (B | C))
```

Associativity of &

```
(A & A) & (B | C)
```

A & A = A

```
A & (B | C)
```

You try this:

Come up with a formula for this Truth Table

Simplify as much as possible

A	B	C	Output
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

You try this:

Come up with a formula for this Truth Table

Simplify as much as possible

Sum of Products:

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

A	B	C	Output
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

You try this:

Simplify first two terms:

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

You try this:

Simplify:

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

Regroup (associative/commutative):

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

You try this:

Simplify:

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

Regroup (associative/commutative):

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

Un-distribute (pull out common factor):

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

You try this:

```
Simplify:
   (!A & !B & !C) | (!A & B & !C)
Regroup (associative/commutative):
   ((!A & !C) & !B) | ((!A & !C) & B)
Un-distribute:
   (!A & !C) & (!B | B)
OR identities:
   (!A & !C) & true  = (!A & !C)
```

You try this:

Come up with a formula for this Truth Table

Simplify as much as possible

Sum of Products:

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

A	B	C	Output
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

You try this:

Come up with a formula for this Truth Table

Simplify as much as possible

Sum of Products:

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

A	B	C	Output
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

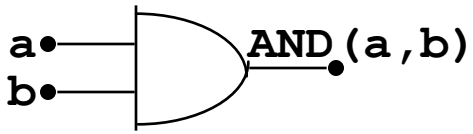
Applying the Theory

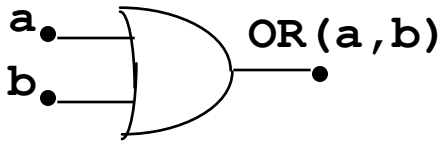
- Lots of good theory
- Can reason about complex Boolean expressions
  - Can design software to minimize
- But why is this useful?

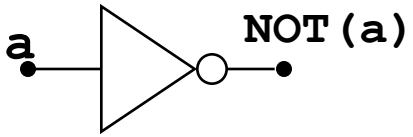
Boolean Gates

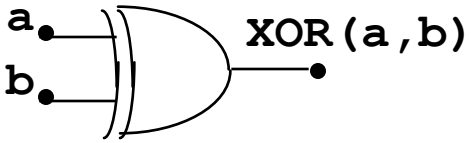
- **Gates** are electronic devices that implement simple Boolean functions (building blocks of hardware)

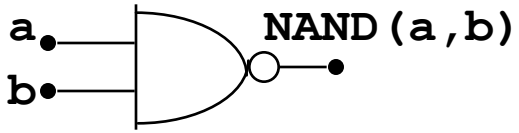
Examples

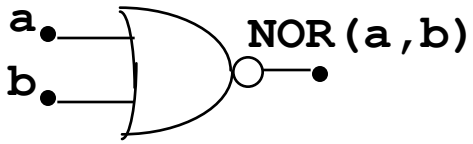


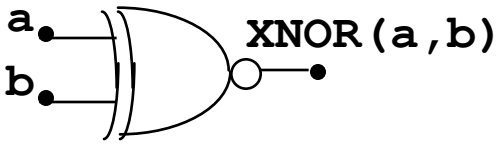






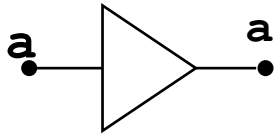






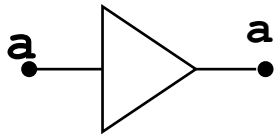
Guide to Remembering your Gates

- **This one looks like it just points its input where to go**
  - **It just produces its input as its output**
  - **Called a buffer**

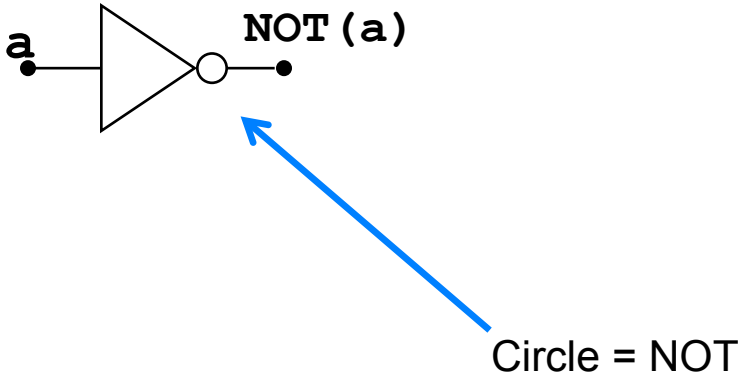


Guide to Remembering your Gates

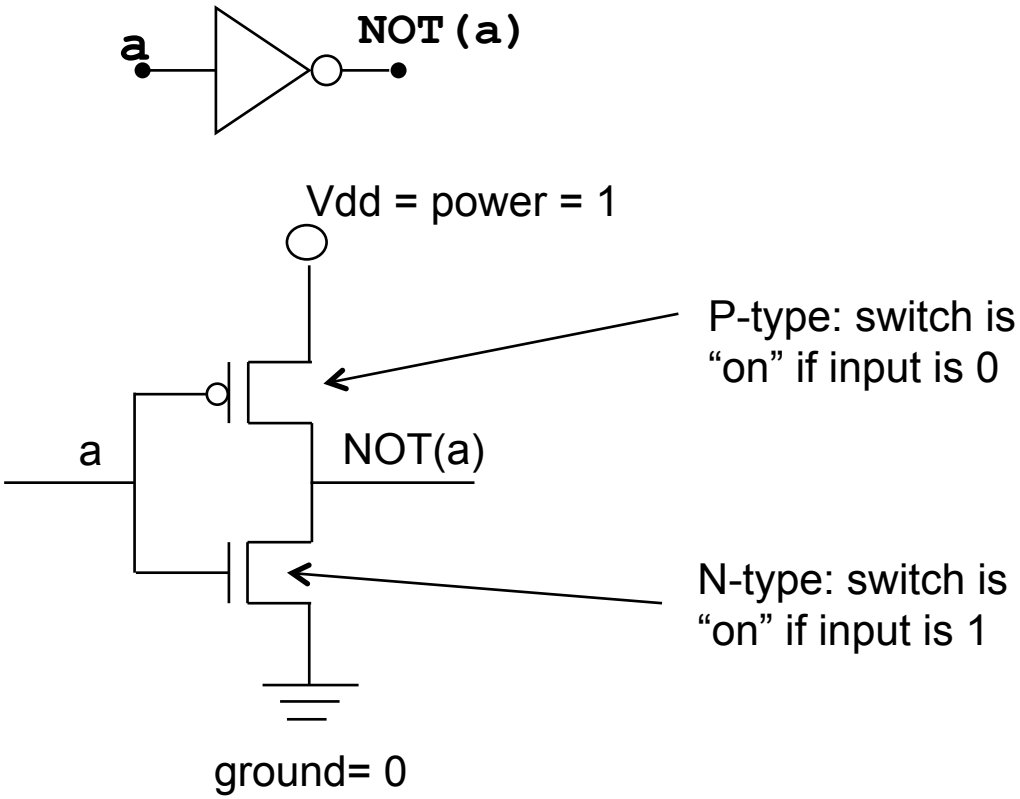
- **This one looks like it just points its input where to go**
  - **It just produces its input as its output**
  - **Called a buffer**



- **A circle always means negate**

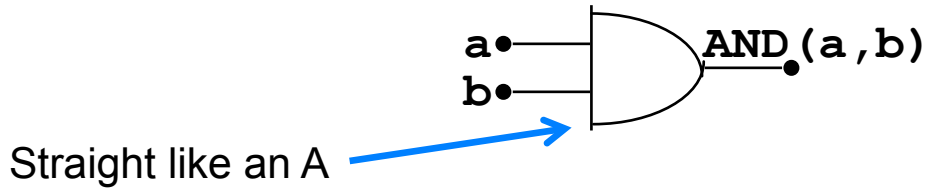


Brief Interlude: Building An Inverter

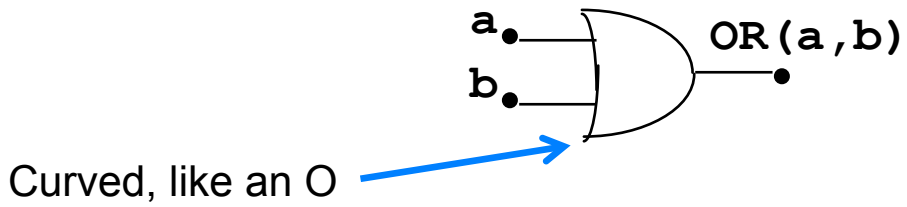


Guide to Remembering Your Gates

- **AND Gates have a straight edge, like an A (in AND)**

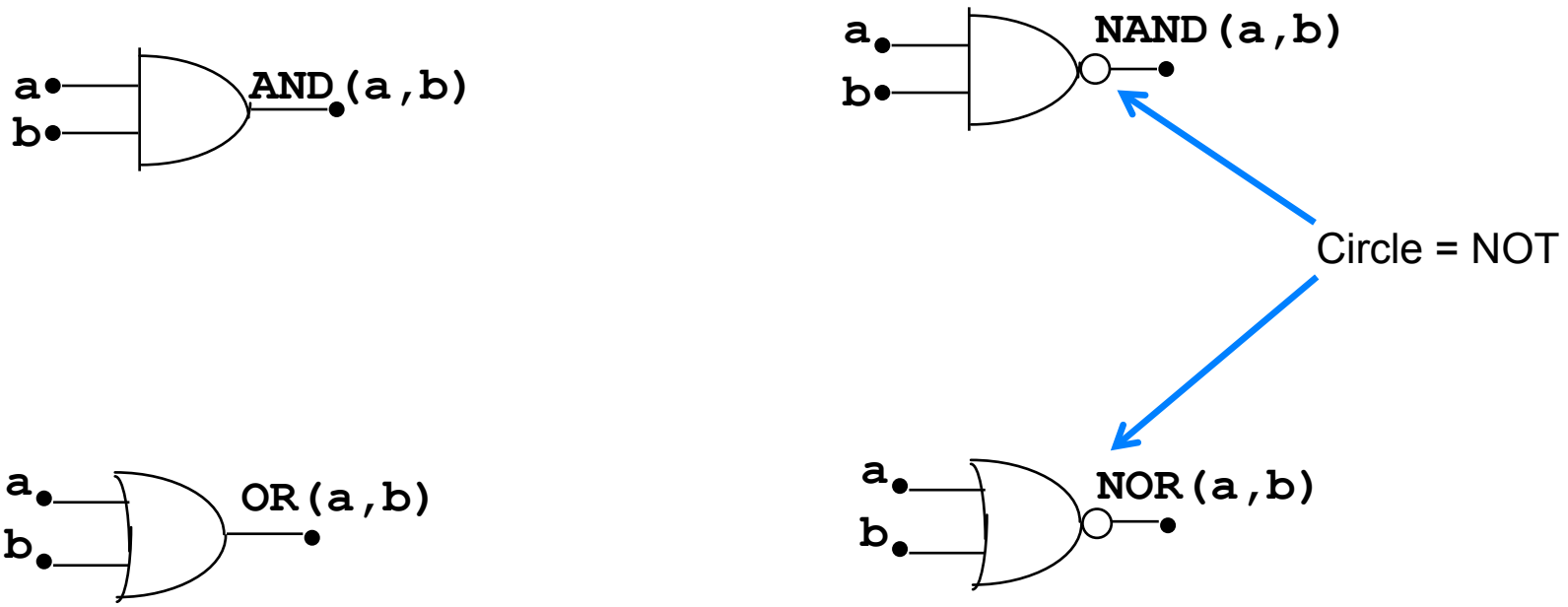


OR Gates have a curved edge, like an O (in OR)



Guide to Remembering Your Gates

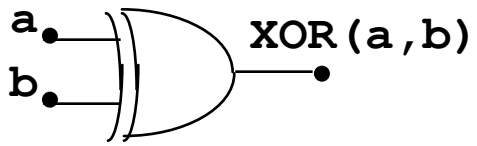
- **If we stick a circle on them…**



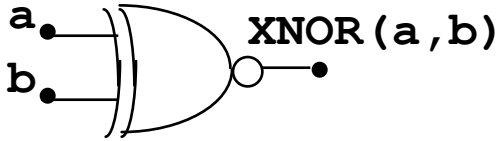
- **We get NAND (NOT-AND) and NOR (NOT-OR)**
  - **NAND(a,b) = NOT(AND(a,b))**

Guide to Remembering Your Gates

- **XOR looks like OR (curved line)**
  - **But has two lines (like an X does)**



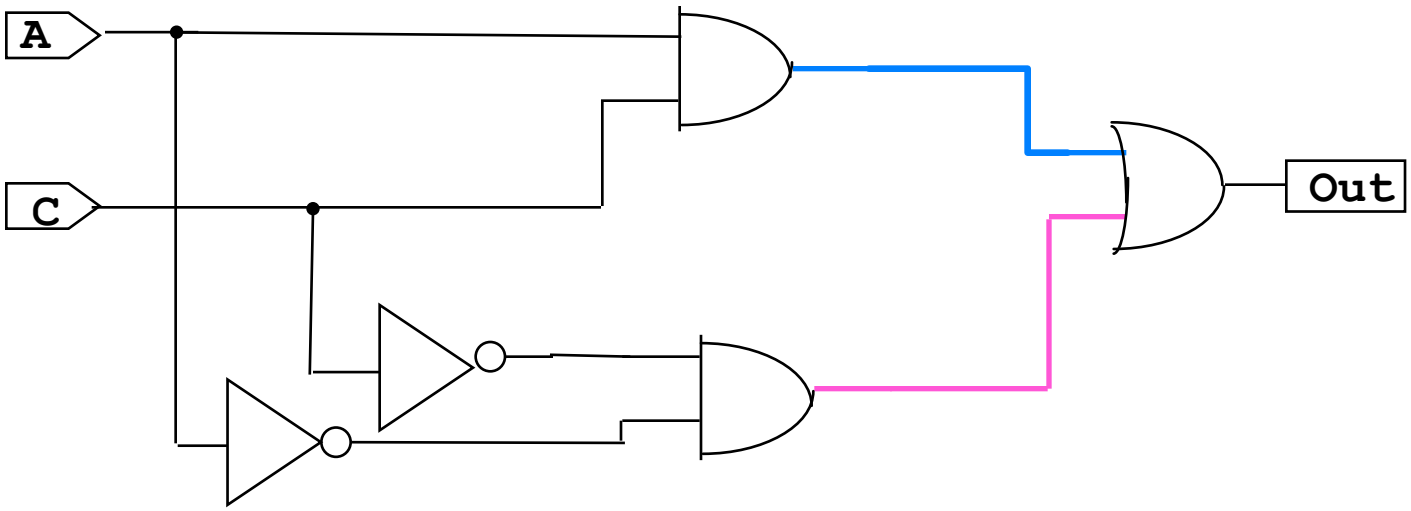
- **Can put a dot for XNOR**
  - **XNOR is 1-bit "equals" by the way**



Boolean Functions, Gates and Circuits

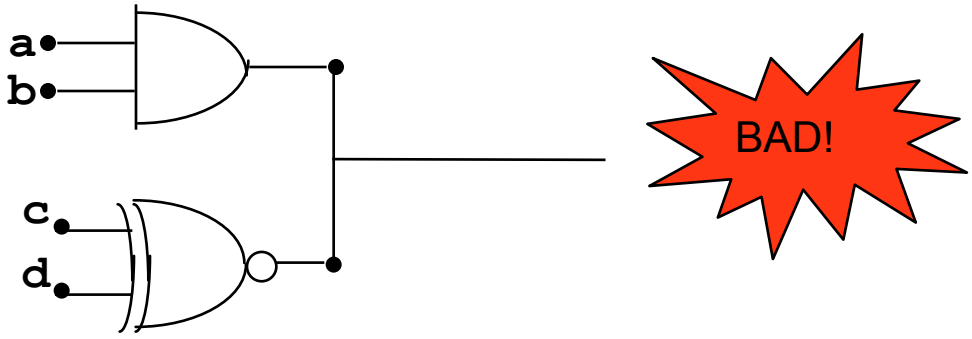
- **Circuits** are made from a network of gates.

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



A few more words about gates

- **Gates have inputs and outputs**
  - **If you try to hook up two outputs, bad things happen (your processor catches fire)**



  - **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**

```
(!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**

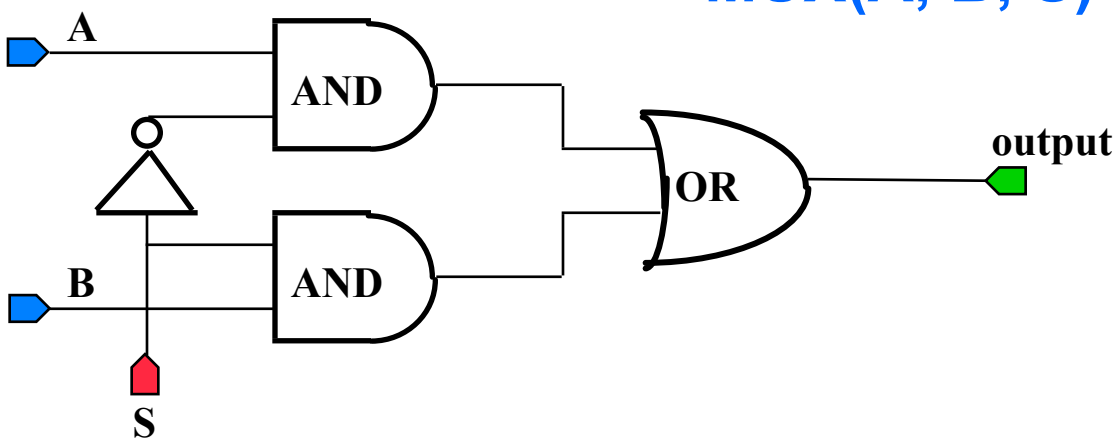
```
(A & !S) |
(B &  S)
```

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

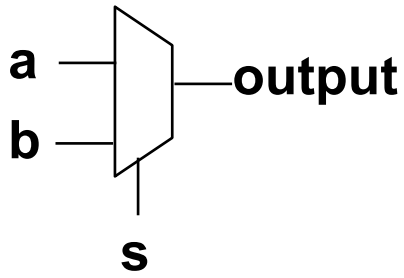
Circuit Example: 2x1 MUX

Draw it in gates:

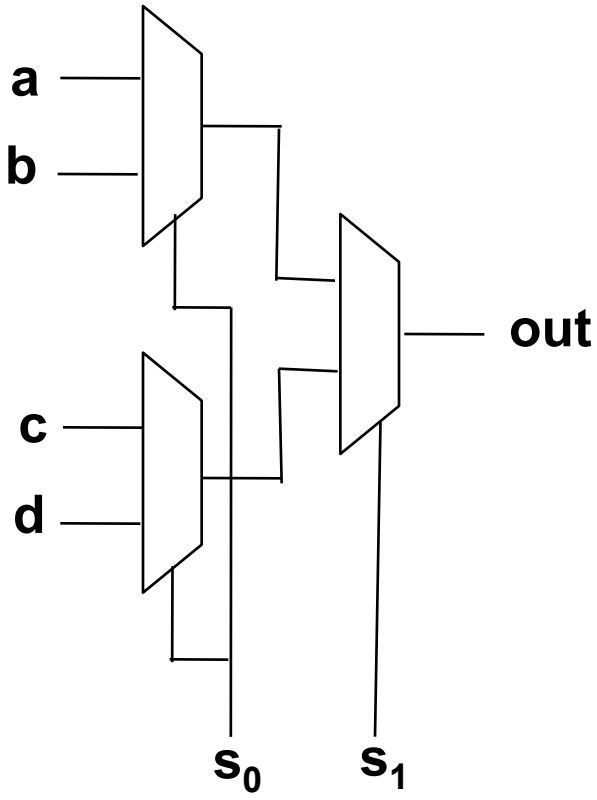
MUX(A, B, S) = (A & !S) | (B & S)

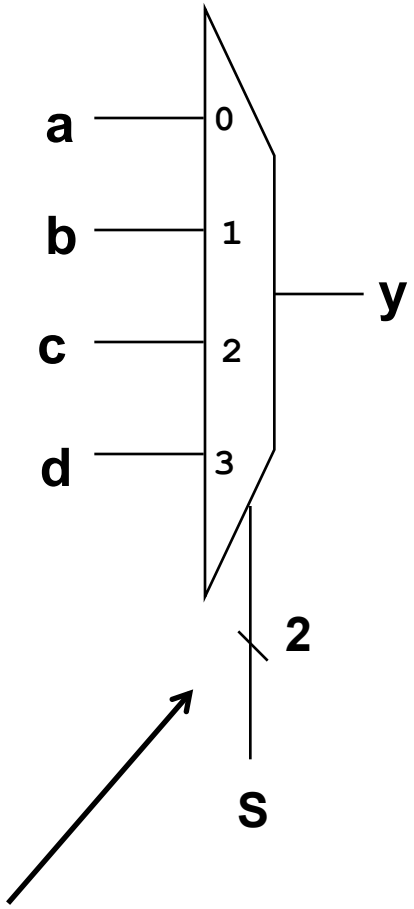


So common, we give it its own symbol:



Example 4x1 MUX





The / 2 on the wire means “2 bits”

Arithmetic and Logical Operations in ISA

- **What operations are there?**
- **How do we implement them?**
  - **Consider a 1-bit Adder**

Summary

- **Boolean Algebra & functions**
- **Logic gates (AND, OR, NOT, etc)**
- **Multiplexors**

- **Download Logisim for Recitation**
  - **http://ozark.hendrix.edu/~burch/logisim/download.html**