Boolean Logic

Steven R. Bagley

Introduction

- Last week, looked inside a computer…
- Saw how information is represented in binary as digital signals
- Today, understand how to manipulate those signals
- To do useful things…

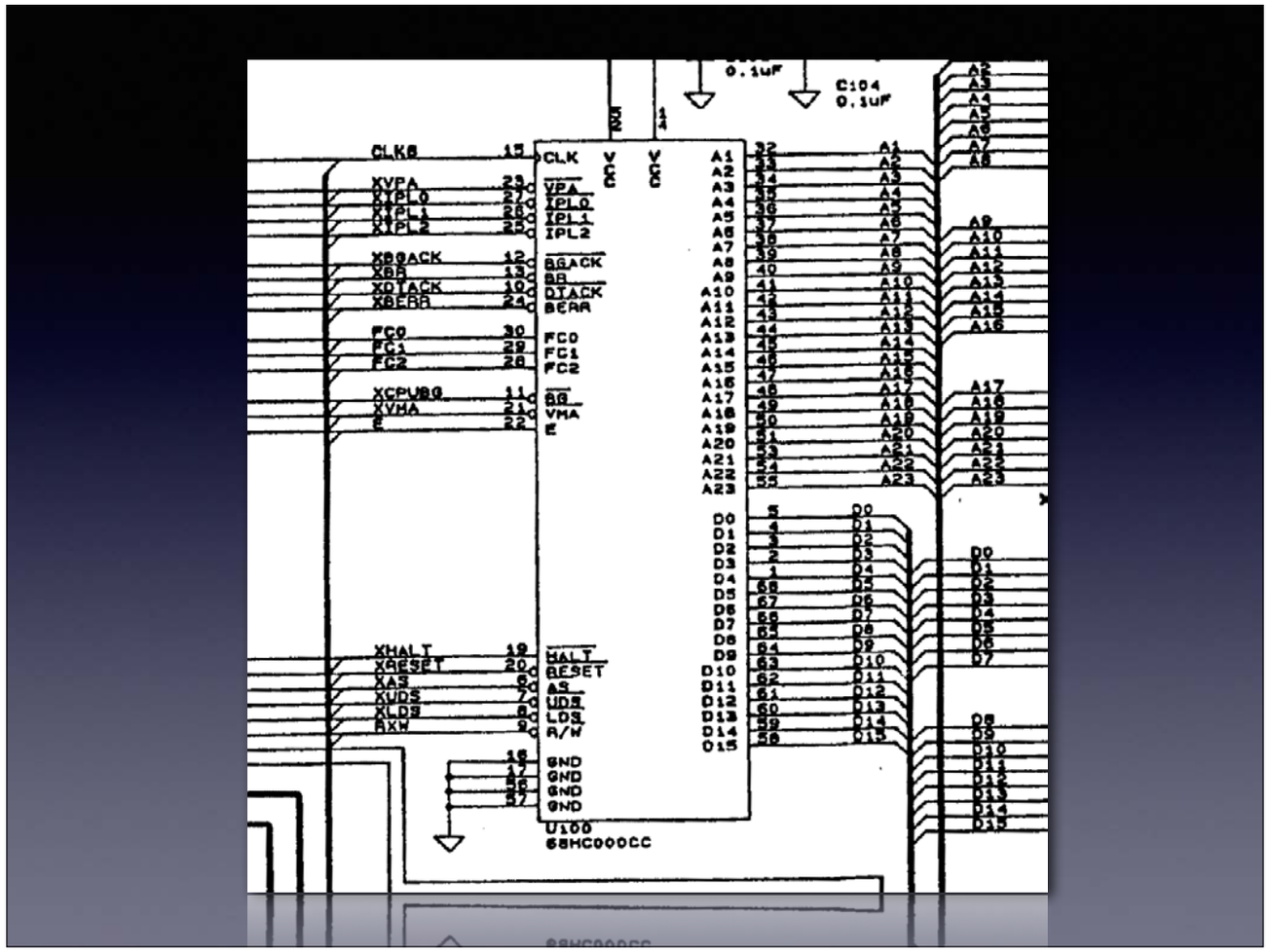
Bits in wires

- Saw how we can represent any number as a sequence of binary digits (bits)
- But how do we transfer that electrically
- Easy for one bit — just send it down the wire

Parallel or Serial

- Two approaches
  - Either use many wires, one for each power of two needed — in *Parallel*
  - Or send the bits one after each other down one wire — *serial*ly
    - Usually synchronized with a clock

Saw parallel approach with the address and data buses on the motherboard



Schematic for connections to the CPU in the Atari ST
Notice how there are 16 data lines, labelled d0-d15 — each corresponds to a power of 2
Similarily for the 24 address lines

Parallel or serial

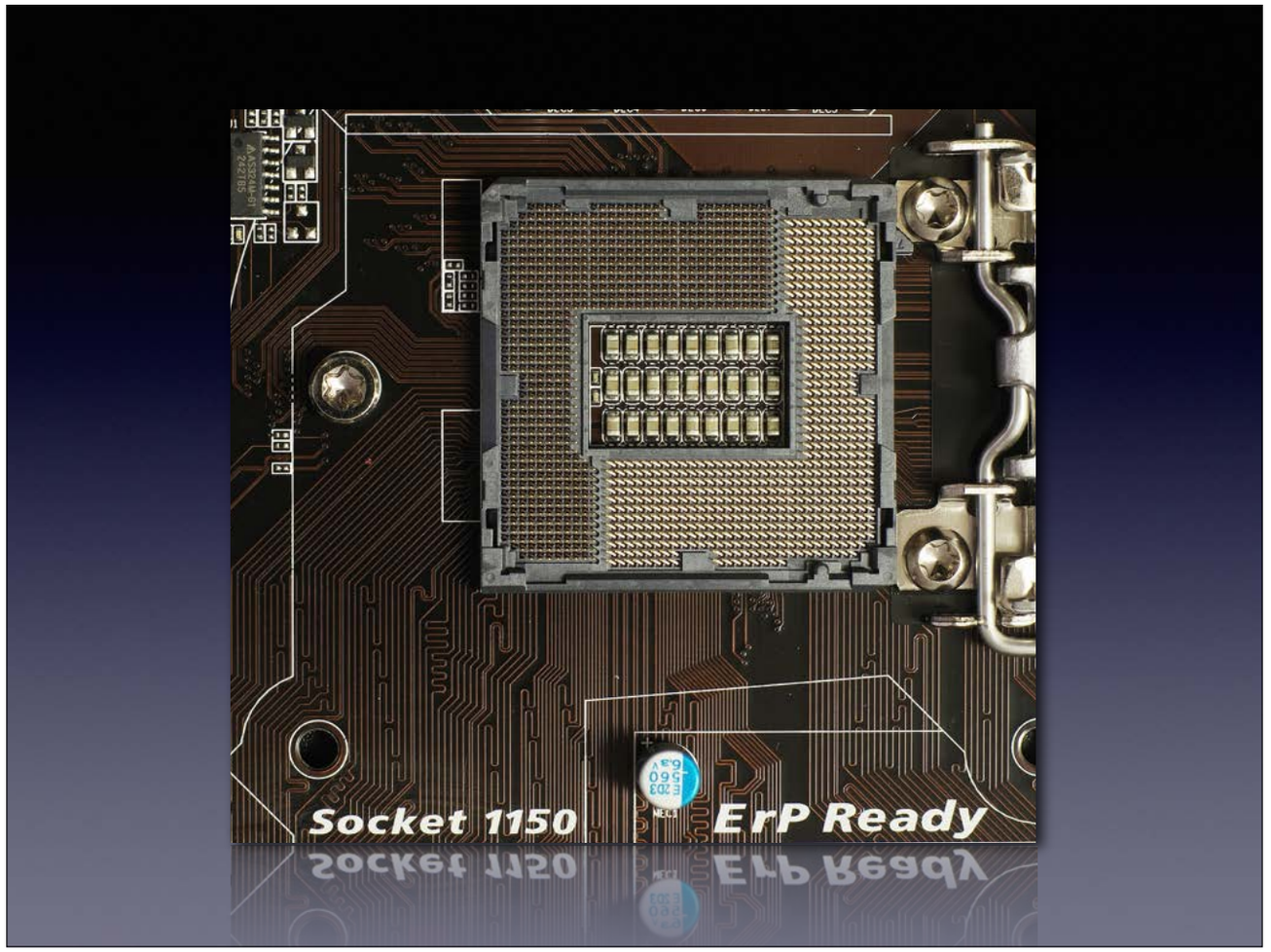
- Parallel transfers all the bits in one go so is faster for the same clock speed
- But as the clock gets faster its starts to be come harder to transfer the data in parallel
- Signal path must be the same length for each bit
- Otherwise some will arrive before others

Think runners on a race track are staggered so the ones on the outside don’t get disadvantaged

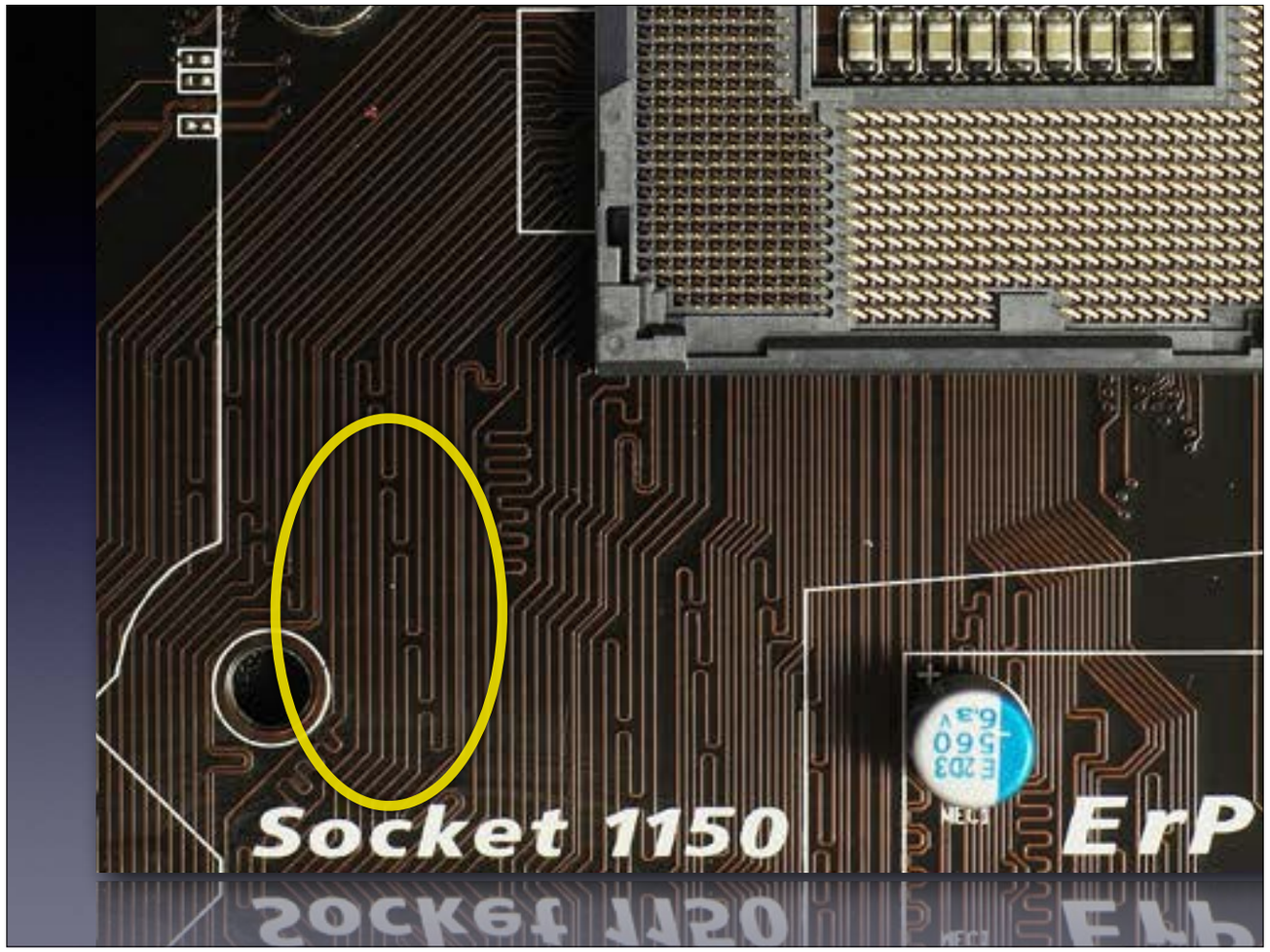
Serial

- Because of this there's now a trend for serial connections at high speed
  - Serial ATA, Thunderbolt, DMI, PCI express
- But these sometimes use multiple 'lanes' to increase the speed

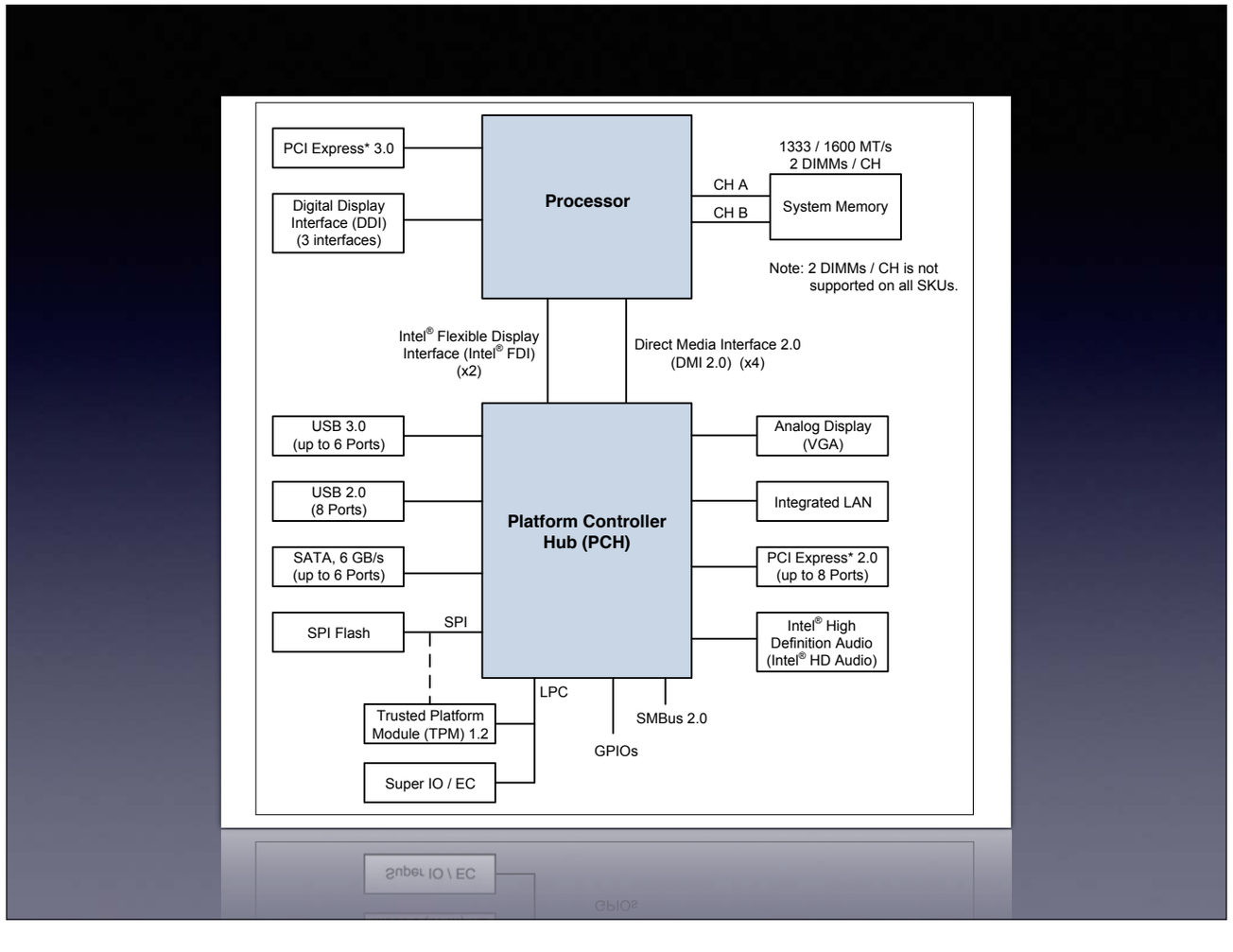
Parallel serial?



LGA1150 socket motherboard, note how the tracks have weird bends to keep them the same length.



LGA1150 socket motherboard, note how the tracks have weird bends to keep them the same length.



Notice the difference with the ST diagram
DMI link replaces ‘traditional’ bus -- although in some ways is still similar…
High speed serial link

Logic Signals

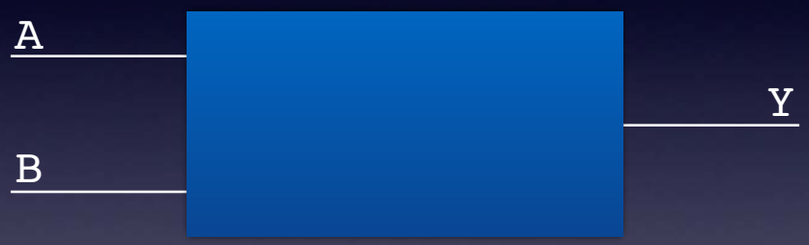
- Digital signals aren't just used to carry numbers
- For now, let's consider the signal to be a logical value
- Either *True* (`1`) or *False* (`0`) value
- These signals are then used to make things happen…

You will encounter logic in G51MFC as well
We'll tend to use 0 and 1 to represent things — elsewhere you'll see T and F or True and false. It's all the same

Logic Circuits

- Can build circuits that process these signals
- Generally have one output
- And one or more inputs…
- Can represent them using a *truth table*
- If we need more than one output, can consider it as two functions with the same input

Logic Circuits



Takes in two inputs, A and B, and produces an output Y based on the state of the two inputs.

Logic Circuits

- Can build circuits that process these signals
- Generally have one output
- And one or more inputs…
- Can represent them using a *truth table*
- If we need more than one output, can consider it as two functions with the same input

Truth Table

Table 1

A	B	Y
0	0	
0	1	
1	0	
1	1	

Specify the inputs on the left, iterate all the possible combinations

Outputs on the right

Truth Table

Table 1

A	B	Y
0	0	1
0	1	0
1	0	1
1	1	0

Specify the inputs on the left, iterate all the possible combinations

Outputs on the right

Might also be called f(A,B)

Truth Table

Table 1

A	B	*f*(A,B)
0	0	1
0	1	0
1	0	1
1	1	0

Specify the inputs on the left, iterate all the possible combinations

Outputs on the right

Might also be called f(A,B)

Logic Functions

- Only a limited number of functions for a specific number of input
- For example, there are only three functions with one input

One input functions

Table 1

Function	A	0	1

Rearrnaged the table a bit

inputs now in the header

Outputs on the right under the input

One input functions

Table 1

Function	A	0	1
		0	0
		0	1
		1	0
		1	1

Rearrnaged the table a bit

Outputs on the right

One input functions

Table 1

Function	A	0	1
Constant Zero		0	0
A		0	1
Not		1	0
Constant One		1	1

Rearranged the table a bit

The second function Not is very useful

Logic Functions

- Again, we can list all two-input logic functions in a similar manner
- This time there are 15 possible combinations…

Two input functions

Function	A	0	0	1	1
	B	0	1	0	1
Constant Zero		0	0	0	0
And		0	0	0	1
A And Not B		0	0	1	0
A		0	0	1	1
Not A And B		0	1	0	0
B		0	1	0	1
Xor		0	1	1	0
Or		0	1	1	1
Nor		1	0	0	0
Equivalence		1	0	0	1
Not B		1	0	1	0
If B then A		1	0	1	1
Not A		1	1	0	0
If X then Y		1	1	0	1
Nand		1	1	1	0
Constant One		1	1	1	0

Rearranged the table a bit

inputs now in the header

Outputs on the right under the input

Two input functions

Function	A	0	0	1	1
	B	0	1	0	1
Constant Zero		0	0	0	0
And		0	0	0	1
A And Not B		0	0	1	0
A		0	0	1	1
Not A And B		0	1	0	0
B		0	1	0	1
Xor		0	1	1	0
Or		0	1	1	1
Nor		1	0	0	0
Equivalence		1	0	0	1
Not B		1	0	1	0
If B then A		1	0	1	1
Not A		1	1	0	0
If X then Y		1	1	0	1
Nand		1	1	1	0
Constant One		1	1	1	0

Rearrnaged the table a bit

inputs now in the header

Outputs on the right under the input

Logic Functions

- Every boolean function can be expressed using the And, Or and Not functions
- This is called it's *canonical representation*
- And, Or and Not have their own symbols that can be used to write down logic equations
- Similar to how we write down mathematical equations

Unfortunately there are lots of symbols used…

Logic Equations

- Often need to write logic equations down
- Different disciplines use different operators
- AND — `A^B`, `A&B`, `A•B`
- OR — `A∨B`, `A|B`, `A+B`
- NOT — `¬A`, `~A`, $\overline{A}$

Symbols in order, Maths, Programning languages (specifically, C), electronics?
Bar can be extend over other things to show what's inverted
Not C also uses ! in some cases to mean inversion

Precedence

- Operators have precedence
- NOT binds most tightly
- Then AND
- Finally OR
- So A+B•C is equivalent to A+(B•C)

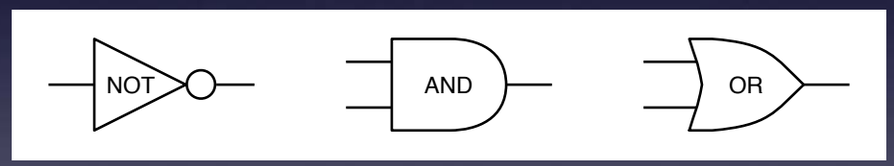
Two input functions

Function	A	0	0	1	1
	B	0	1	0	1
Constant Zero	0	0	0	0	0
And	$A \bullet B$	0	0	0	1
A And Not B	$A \bullet \overline{B}$	0	0	1	0
A	A	0	0	1	1
Not A And B	$\overline{A} \bullet B$	0	1	0	0
B	B	0	1	0	1
Xor	$A \bullet \overline{B} + \overline{A} \bullet B$	0	1	1	0
Or	$A+B$	0	1	1	1
Nor	$\overline{A+B}$	1	0	0	0
Equivalence	$A \bullet B + \overline{A} \bullet \overline{B}$	1	0	0	1
Not B	$\overline{B}$	1	0	1	0
If B then A	$A+\overline{B}$	1	0	1	1
Not A	$\overline{A}$	1	1	0	0
If X then Y	$\overline{A}+B$	1	1	0	1
Nand	$\overline{A \bullet B}$	1	1	1	0
Constant One	1	1	1	1	0

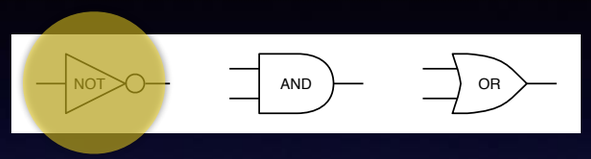
Logic Gates

- An electronic circuit representing a logic function is called a *gate*
- So with electronic circuits that represents And, Or and Not
- We can combine them to build any logic function as a circuit
- To process the signals in the computer

Logic Gates

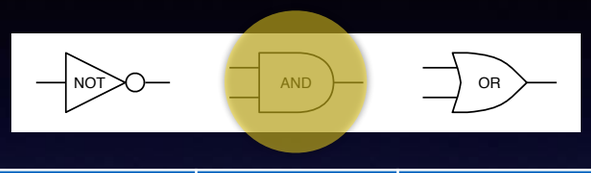


Logic Gates



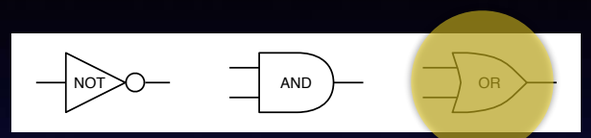
A	Result
0	1
1	0

Logic Gates



A	B	Result
0	0	0
0	I	0
I	0	0
I	I	I

Logic Gates



A	B	Result
0	0	0
0	I	I
I	0	I
I	I	I

Boolean Algebra

- There are a set of algebraic rules for logic functions
- Set out by George Boole in ‘A Calculus of Logic’
- Allows us to reason about logic functions and simplify them…
- Can also use them to show that we can build every logic function if we are given NAND

Boole, 1815-1864

Boolean Algebra

- Already seen how we can build all two-input functions from And, Or and Not
- So if we can build those three functions using NAND then we can build all the functions

Boolean Definitions

- First, set define the action of functions

0+0 = 0	0•0 = 0
0+1 = 1	0•1 = 0
1+0 = 0	1•0 = 0
1+1 = 1	1•1 = 1

Equivalent to the truth table

Boolean Definitions

- Second, functions for which one input is a variable

$$A+0 = A \qquad A \bullet 0 = 0$$
$$A+1 = 1 \qquad A \bullet 1 = A$$
$$A+A = A \qquad A \bullet A = A$$
$$A+\overline{A} = 1 \qquad A \bullet \overline{A} = 0$$

$$\overline{\overline{A}} = A$$

Go through them

Last one is NOT of NOT A

Boolean Definitions

- Third, for more than one variable

```
A+B          = B+A      A•B          = B•A
(A+B)+C      = A+(B+C)  (A•B)•C      = A•(B•C)
(A•B)+(A•C)  = A•(B+C)  (A+B)•(A+C)  = A+(B•C)
```

Go through them

Boolean Definitions

- Fourth, De Morgan's Theorem

$$\overline{A \cdot B} = \overline{A} + \overline{B} \qquad \overline{A + B} = \overline{A} \cdot \overline{B}$$

Go through them
These rules are what lets us build everything out of NAND gates
Show truth tables on board for these

Logic Gates

- Given a set of NAND gates, we can design any logic circuit we want
- Can get programmable logic chips that are just arrays of NAND gates
- Easier to design using And, Or and Not gates
- As it's *canonical* representation
- And let the computer convert into NAND gates