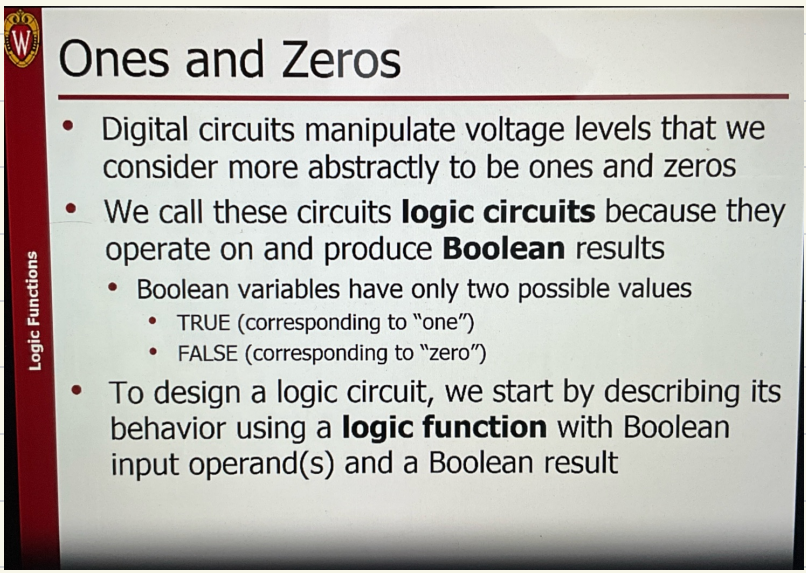
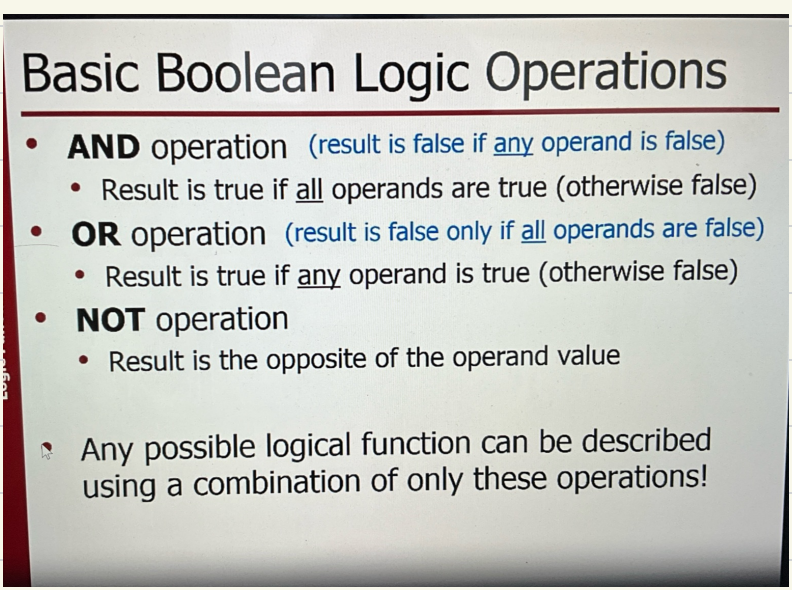
Week 4

Logic Functions





Truth Tables

- Logic functions are defined by their **truth table**
  - Table that indicates whether the result is true or false for all possible combinations of input operand values

Logic Functions

Logic Operator Symbols

- Logic functions are often expressed using operator symbols instead of words

F = A AND B	G = A OR B	H = NOT A
F = A • B = AB	G = A + B	$H = \overline{A}$

- You may see other symbols used in other places for these (and similar) types of operations

Logic Functions

Functions From Truth Tables

- Each row of a truth table represents one possible combination of input operand values
  - Can build a function that implements a truth table by ORing together terms for all rows where result is 1

K	M	F
0	0	0
0	1	1
1	0	1
1	1	1

$F = \overline{K}M + K\overline{M} + KM$

- Different (logically-equivalent) functions can implement the same truth table
  - Some are more optimized than others...

Logic Functions

Logical Equivalency

- Two functions are logically equivalent only if their truth tables match for each input combination

$K = M + G\overline{L}$

G	L	M	K
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	**1**
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	**1**

$J = (M + G)\overline{L}$

G	L	M	J
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	**0**
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	**0**

when L is 1, it forces J to be 0
when L is 0, J is equal to (M + G)

Logic Functions

Boolean Identities

- Can derive Boolean identities from the truth tables of the logic operations...

A	B	A•B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

A	$\overline{A}$
0	1
1	0

$A \cdot 0 = 0$
$A \cdot 1 = A$

$A + 0 = A$
$A + 1 = 1$

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

- Another useful transformation: **DeMorgan's Law**

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

Logic Functions

- XOR (exclusive)

A	B	out
0	0	0
0	1	1
1	0	1
1	1	0

A XOR B = $A \oplus B$

The Odd Function

- The name "exclusive OR" only works when there are exactly two operands...
- **Example:**

$$Y = (A \oplus B) \oplus C$$

  - Compute intermediate value $(A \oplus B)$ for each row
  - Then XOR intermediate value with C in each row to determine Y

A	B	C	$A \oplus B$	Y
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	0	1

- Y is 1 only when an odd # of the inputs are 1

Logic Functions

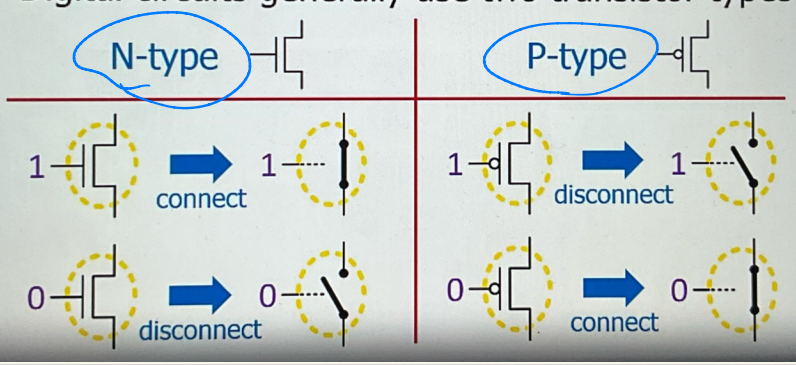
Logic Gates And Combinational Logic

Logic Calculations In Hardware

- We think of logic circuits as having Boolean input values and producing Boolean results
  - In reality, they are built from transistors that connect each output to power or ground based on the design of the circuit and the voltage level of each input
  - Logic circuits are at a higher level of abstraction
- How can we use transistors to "compute" a voltage level?

Transistors

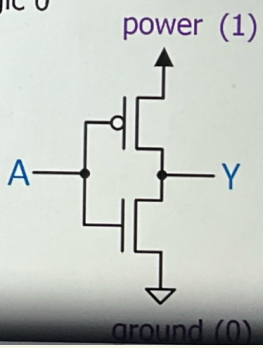
- A field-effect transistor, or FET, is basically a switch that is controlled by a voltage
  - Logic 0 and logic 1 are different voltages
- Digital circuits generally use two transistor types



Transistors in Logic Gates

- A logic gate is an electric circuit that contains transistors that connect the output to power or ground based on the input voltage(s)
  - power → voltage interpreted as logic 1
  - ground → voltage interpreted as logic 0

- **Example:** NOT gate ($Y = \overline{A}$)
  - P-type transistor connects output Y to power (logic 1) if input A is connected to ground (logic 0)
  - N-type transistor connects output Y to ground (logic 0) if input A is connected to power (logic 1)



Designing Logic Circuits

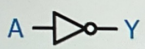
- Design logic circuits using **logic gates**, which correspond to AND, OR, NOT, XOR, and other small but useful logic operations
  - Logic values 0 and 1 are more abstract than voltages

- Use complicated software that transforms our logic circuits into an implementable form at a lower level of abstraction
  - For example: into a transistor-level layout that can be used to fabricate a microchip

Logic gates for humans, more abstract than the actual hardware

Logic Gates

NOT gate

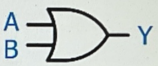


$Y = \overline{A}$

AND gate

$Y = AB$

OR gate



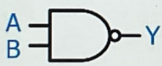
$Y = A+B$

XOR gate

$Y = A \oplus B$

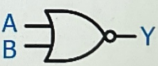
"bubble" indicates NOT

NAND gate



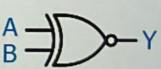
$Y = \overline{AB}$

NOR gate



$Y = \overline{A+B}$

XNOR gate



$Y = \overline{A \oplus B}$

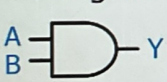
In a logic circuit, the logic function "variables" become **signals** transmitted over wires

Combinational Circuits

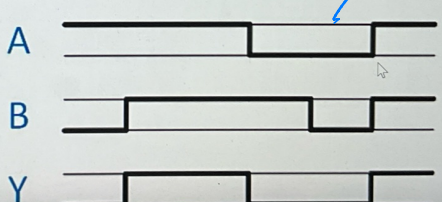
- A **combinational circuit** is a type of logic circuit whose output value depends only on its design and the present values of its inputs
  - A given circuit's output will always be the same for the same set of input values
- **Example:** AND gate behavior over time

"waveform"

AND gate

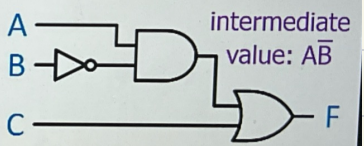


$Y = AB$



Waveforms

- A waveform is a visual way to represent signal values over time
  - Often used to show how a circuit output changes in response to input signal changes
- **Example:**
  - Circuit with inputs A, B, C and output F



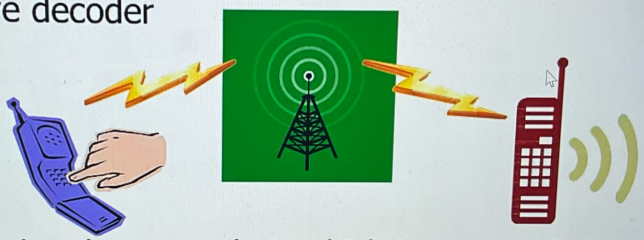
Combinational Building Blocks

Decoder

- Converts an **n**-bit input codeword into a unique **m**-bit output value, where $\mathbf{m}=2^{n}$ → In Decimal
  - There are other types of decoders, but these are the ones that are important in this class
  - This type of decoder has one output for each possible combination of input values
- Exactly one output can be true at any given time
  - The input signal values determine which one it will be
- Here we are only concerned with how the decoder **behaves**, not how it is constructed internally
  - It is, of course, built from logic gates

Decoder Analogy

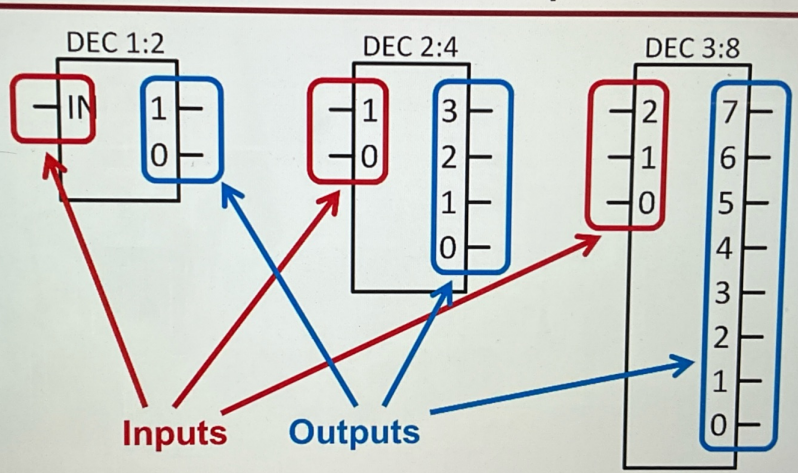
- The global telephone system behaves like a massive decoder



- You dial a phone number, which is a multi-digit numeric code for the person you want to talk to
  - *A binary value at the input(s) of the decoder*
- One telephone somewhere in the world rings
  - *One and only one output of the decoder is true*

(相当于解码)

Decoder Symbol Examples

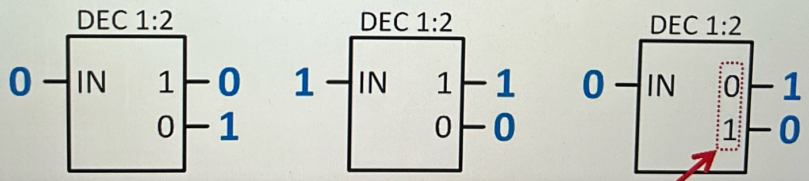


- We label the inputs and outputs in a sensible way
  - If the input value is ***N***, then output ***N*** is true!

[kinda like from binary to Decimal]

1:2 Decoder

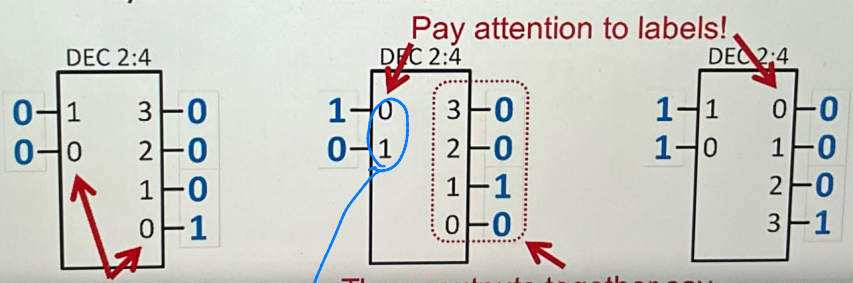
- One input, two outputs ($2^{\#inputs} \rightarrow 2^1 \rightarrow$ 2 outputs)
- The input value indicates **which one** of the outputs will be 1 (the other output will be zero)
- The outputs **must be labeled** so we know which is output 1 and which is output 0
  - The top output will not always be the higher number...



Pay attention to labels! They explain which output

2:4 Decoder

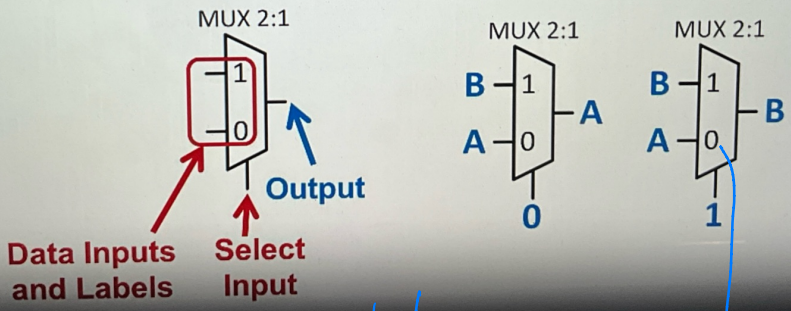
- Two inputs, four outputs
- The input can be interpreted as a binary number, and that number indicates **which** output will be 1
  - Inputs/outputs **must be labeled** if more than one
  - They should be in an order that makes sense



This represents the "position of digits"

Multiplexers (Muxes)

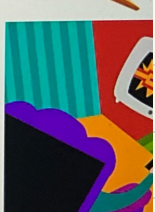
- Selection: based on the select input(s), choose **one** of the data inputs to drive the output
  - Output is equal to **one** of the data inputs
  - Select inputs indicate **which** data input
- Typically: n select lines, 2^n data input lines



"position"

Multiplexer Analogy

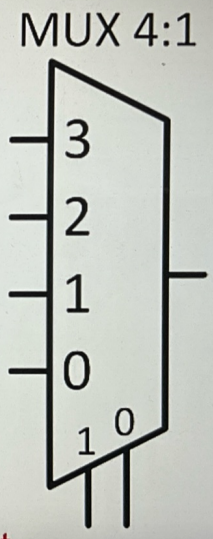
- Selecting a television channel
  - All of the available channels are present as inputs to your TV
  - You **choose** which one to watch



- You enter the number of the channel
  - *A value applied to the select inputs*
- The television picture changes to match the content for that channel
  - *The mux output becomes the same value as is present at the selected data input*
  - *If the value at the selected data input changes, the output changes to match*

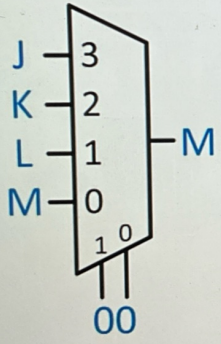
Multiplexer Labeling

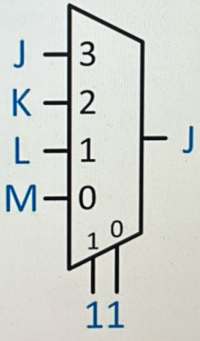
- Data inputs **must** be labeled to indicate which input is chosen for a given select value
- Select inputs **must** be labeled so we know the ordering of bits within the select value
  - We need to interpret the select as a binary number

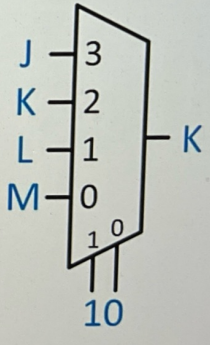


Exception: a 2:1 mux has only one select signal, so it does not need a label…

4:1 Multiplexer Examples







Adders

- Accept 3 inputs: A bit from each addend and a carry value from bit position to the right
- Produce 2 outputs: The sum bit at that position, and a carry value to send to the bit position to the left.

↓

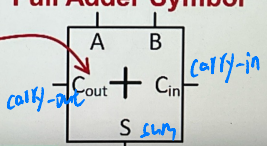
★ This is "full-Adder"
(Half-Adder does not have the carry bit)

Binary Addition Hardware

- At every position, determine 2-bit sum of adding each addend's bit in that position plus a carry-in
  - The carry-in might be 1 or might be 0
  - Lower bit of the result is the sum and the upper bit is the carry-out for that bit position
- A **Full Adder** performs addition for a single bit position

(Bit by bit)

Full Adder Symbol



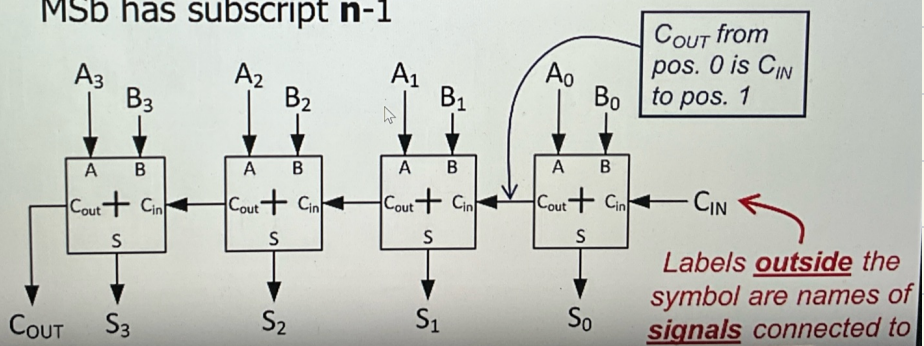
$C_{out}\ S = A + B + C_{in}$

Full Adder Truth Table

A	B	Cin	Cout	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

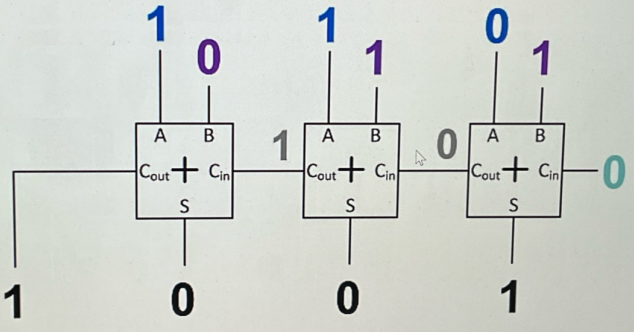
Ripple Carry Adder

- A series of full adders, where each full adder performs the addition for a single bit position
  - A carry can occur at any bit position and ***ripple*** through towards the most significant bit.
  - In an **n**-bit number the LSb has subscript 0 and the MSb has subscript **n**-1



Addition Example

- Perform F + G, where $F = 110_2$ and $G = 011_2$
  - Set the LSb C_{in} to 0 since we do not need it



- If unsigned, $F + G = 6_{10} + 3_{10} = 9_{10} = 1001_2$