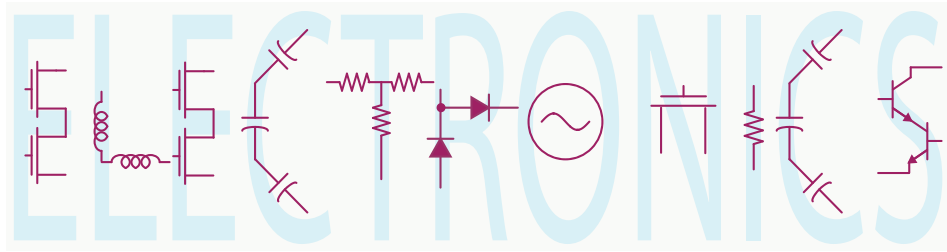
EE 42/100
Lecture 21: Digital Gates and Combinatorial Logic



Rev B 4/9/2012 (9:39 PM)

Prof. Ali M. Niknejad

University of California, Berkeley

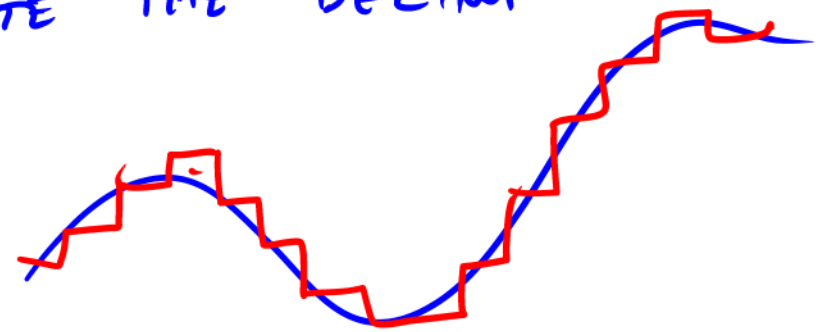
Digital versus Analog

- Real signals are continuous (at least we perceive them that way) whereas binary signals can only take on two values.
- For instance the current temperature in this room is a certain number of degrees and the resolution of this number is in principle infinite (noise limited).
- But for many applications, the precise value is not important. A finite representation (certain number of significant figures) is all that we need. We may therefore sample the signal periodically (not continuously) and also round off the actual amplitude of the signal. Note that we are making two approximations: the discrete time sampling of the signal and the discretization process.
- Intuitively, as long as we sample the signal faster than the rate at which it changes (can be made precise using Nyquist's Sampling Theorem), then no information is lost.

→ COMPUTERS HAVE FINITE PRECISION

→ "TRUNCATE THE DECIM"

→ FINITE RANGE



Quantization Noise

- On the other hand, the quantization of the signal introduces round-off errors in our signal. If we subtract out the ideal signal $s(t)$ from the quantized signal $\hat{s}(t)$, the difference is the error signal, which varies randomly from sample to sample.

$$n(t) = s(t) - \hat{s}(t)$$

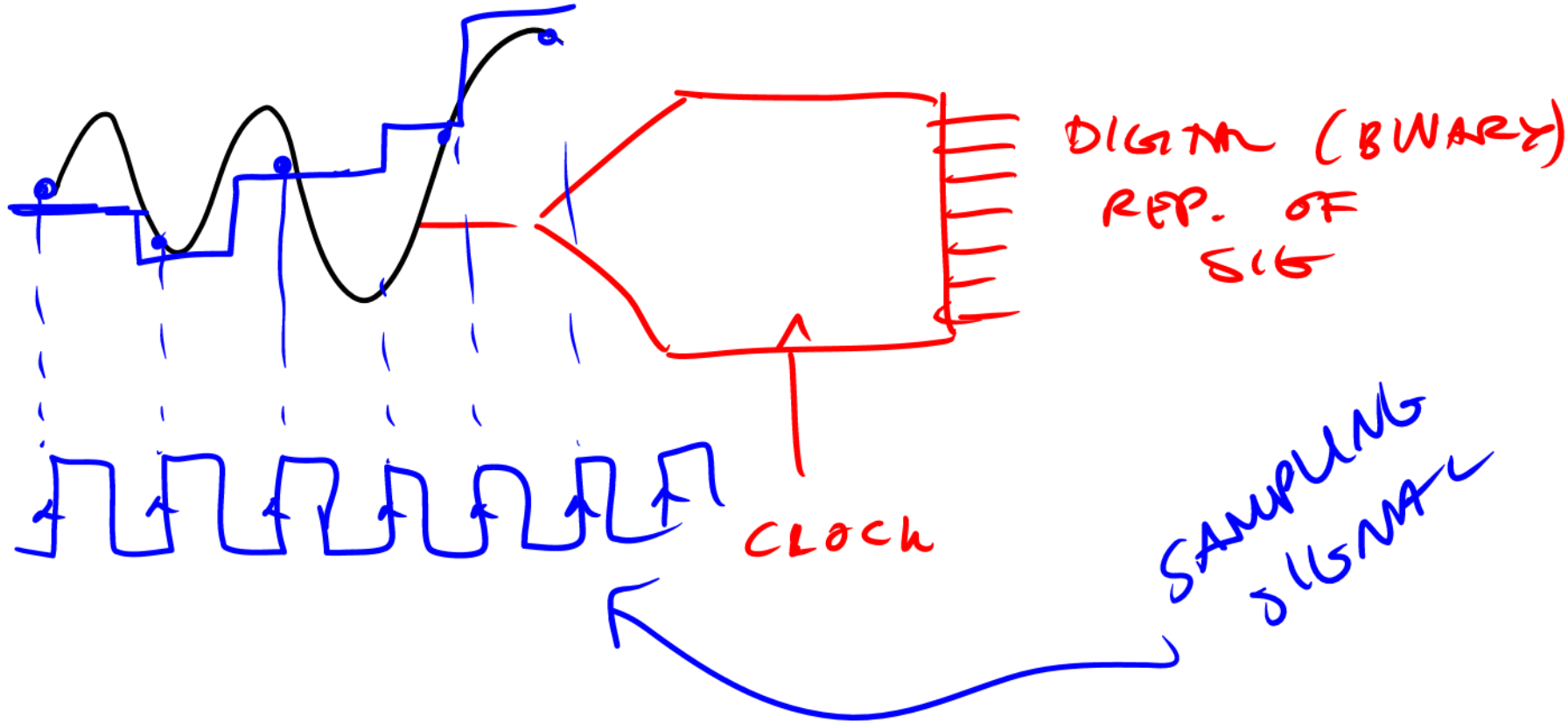
- We can therefore think of the process as introducing "quantization noise" in the signal.

$$\hat{s}(t) = s(t) - n(t)$$

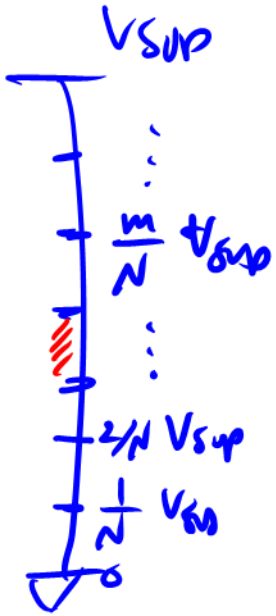
- In other words, if we are insensitive to changes of the signal smaller than the quantization noise, then the approximation is okay.

Analog-to-Digital Converter

- The important specifications for an ADC are the sampling rate (clock rate) of the ADC and its resolution (number of bits).
- Analog signals need to be sampled at a rate of twice the signal bandwidth (audio bandwidth is roughly 5 kHz).
- The resolution determines the smallest discernible signal since the full-scale input voltage (say 5V) is divided by 2^N where N is the number of bits. Any signal smaller than this level is lost in the "quantization noise" (round-off error).



A/D CONVERTER

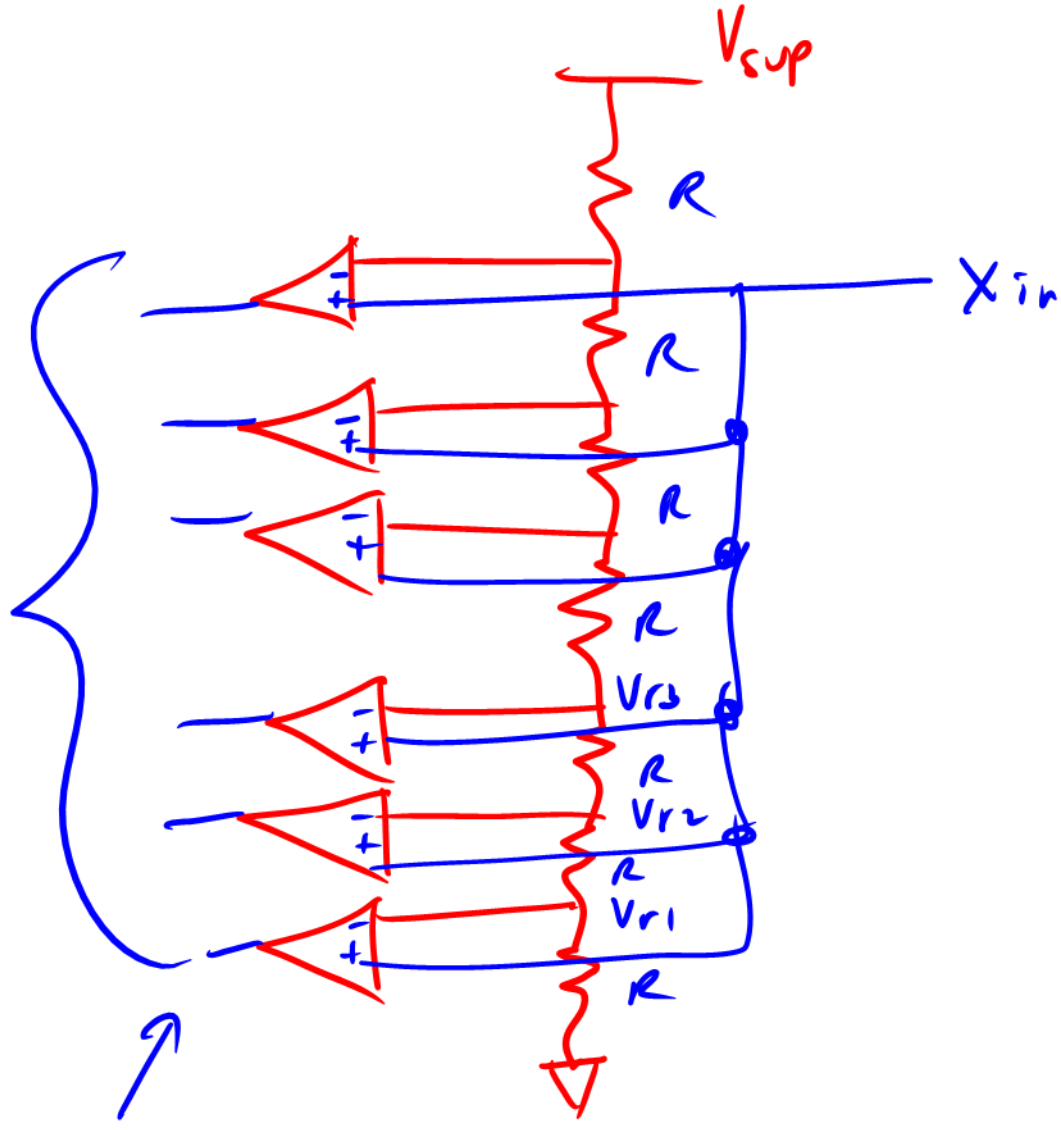


DIG O/P

$$V_{r1} = \frac{R}{\Sigma R} V_{sup}$$

$$V_{r2} = \frac{2R}{\Sigma R} V_{sup}$$

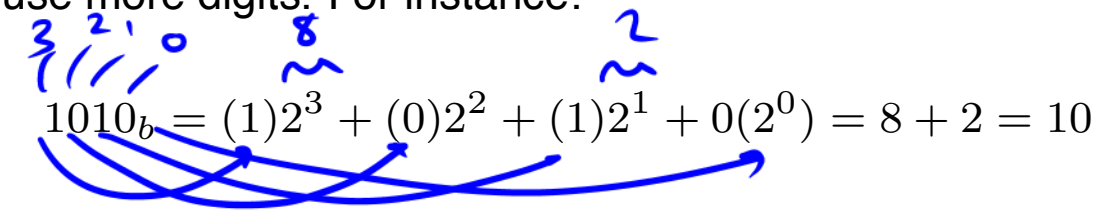
$\vdots$



NOT BINARY

Binary Numbers

- Binary numbers can only take on two values $\{0, 1\}$. To specify a bigger number, you have to use more digits. For instance:



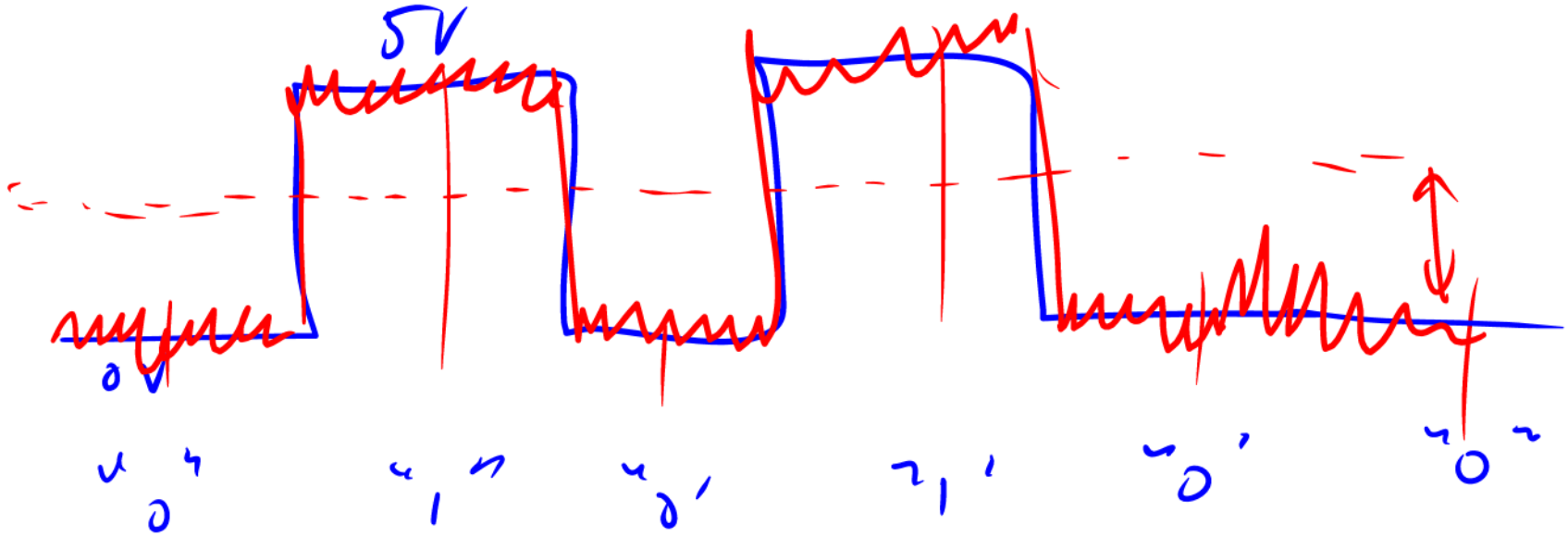
$$1010_b = (1)2^3 + (0)2^2 + (1)2^1 + 0(2^0) = 8 + 2 = 10$$

- Some aliens with only two fingers may naturally use binary numbers but we use them because our computers can only store two states (note: digit means finger).
- To convert a decimal number into binary, you can repeatedly divide by 2 and take the remainder as the next digit.
- Example: 50 in binary is:
  $50/2 = 25r0, 25/2 = 12r1, 12/2 = 6r0, 6/2 = 3r0, 3/2 = 1, r1, 1/2 = 0r1$. This implies the number is given by 110010_b (read it backwards). Verify that $2^4 + 2^1 = 8 + 4 = 12$. Or

$$32 + 16 + 2 = 2^5 + 2^4 + 2^1 = 110010_b$$

Binary Signals

- Binary signals are designed to only take on two values, say $\{0V, 5V\}$ to represent $\{0, 1\}$ (positive or negative logic possible).
- As signals propagate in wires, inevitably noise, distortion, and cross-talk occurs, which corrupts the signal. An analog signal is corrupted directly whereas a digital signal is only corrupted if the noise exceeds the noise margin of the circuit.
- Digital signals are therefore more robust and can be easily regenerated whereas analog signals get more noisy as we process them.



Hexadecimal $= 2^4 = 16$

- A long string of binary numbers if hard to read. We therefore group 4 bits into a *nibble* and group eight bits into a *byte* (early CS people had a good sense of humor!).
- A nibble can take on 16 different distinct values, so a base 16 number system is used to describe it: $\{0, 1, 2, \cdots, A, B, C, D, E, F\}$. Note that we use letters to represent numbers. This takes some getting used to!
- A byte is described by two hexadecimal digitals. For instance, we converted 50 in binary as 110010_b. Let's extend it to 8 bits by zero padding, 00110010_b, or two nibbles of 0011 and 0010, which in hex are 3_{hex} and 2_{hex}, which we write as 32_{hex}. Verify

$156_{hex} = 1 \times 16^2 + 5 \times 16^1 + 6 \times 16^0$

$$32_{hex} = 3 \cdot 16^1 + 2 \cdot 16^0 = 48 + 2 = 50$$

0110 1110 = 8 bits [0-255] = BYTE

NIBBLE NIBBLE

6 E = $6E_{hex}$ = $6 \times 16^1 + 15 \times 16^0$

Binary Arithmetic

- Binary arithmetic is exactly the same as what you learned in grade school, except there are only two digits! For adding single bits

$$0 + 0 = 0$$

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

$$1 + 1 = 0 + \text{carry}$$

1+1 = 2
= 10 binary

- Let's add two nibbles

```
 1 1 1
 0101    (5)
+0011    (3)
----
 1000    (8)
```

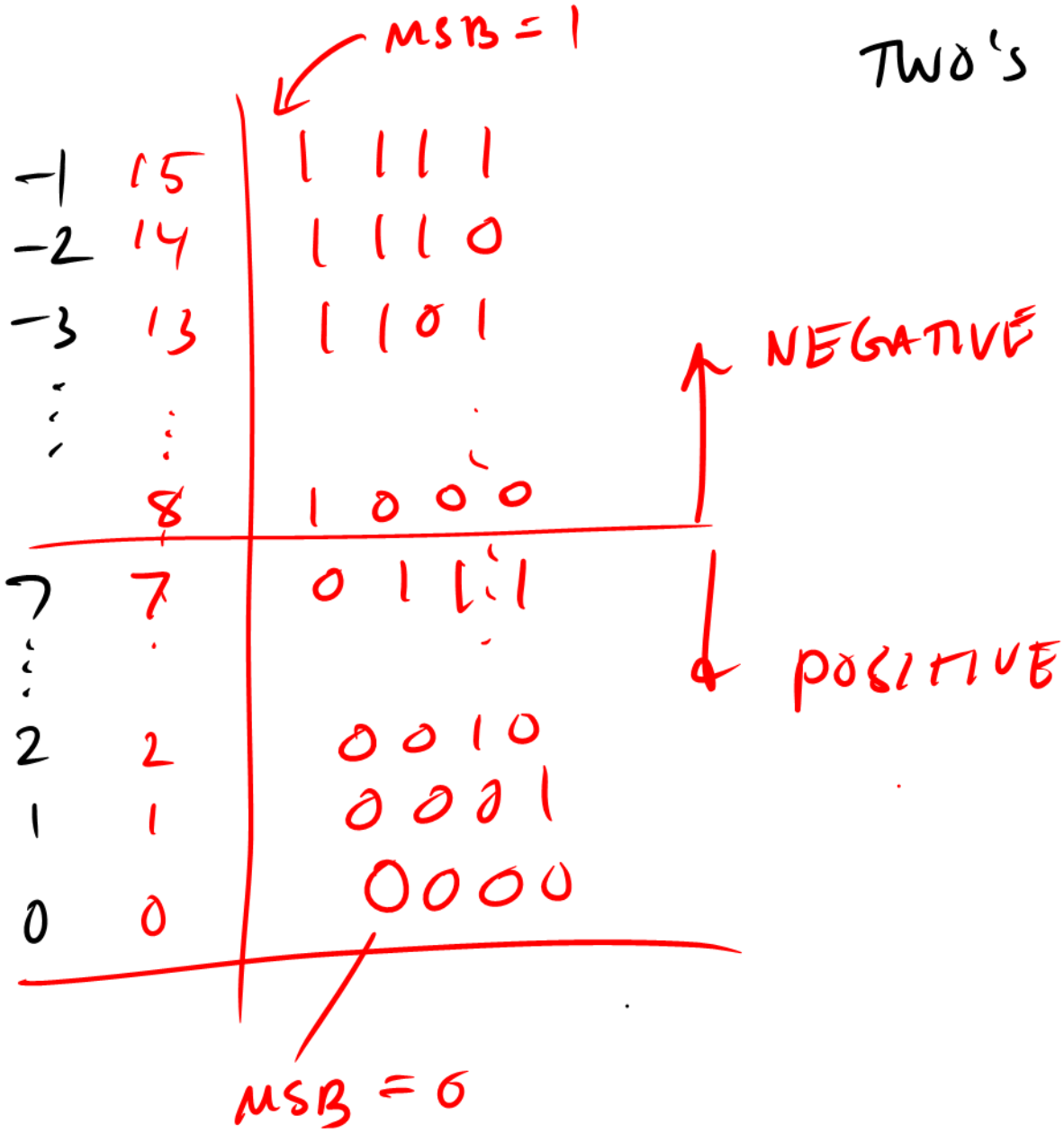
Signed Binary and Two's Complement

- To represent negative numbers, we could add a "sign bit" in front of the number but it turns out that addition no longer works (between positive and negative numbers). A more elegant system that preserves addition (and hence supports subtraction) is the two's complement system.
- In the two's complement system, the first half of the 2^N numbers are assigned to zero and the positive integers and the second half is used for the negative integers. Because addition is modulo 2^N (wraps around after 2^N), when you add a number m and it's negative counterpart, it should sum up to wrap back to zero, which would happen if m were equal to $2^N - m$.
- Take the number 0001. It's 2's complement is $2^4 - 1 = 15$ or 1111. If we add these two numbers in binary, we get

$$\begin{array}{r} 0\,0\,0\,1 \\ +\,1\,1\,1\,1 \\ \hline 1\,0\,0\,0\,0 \end{array}$$

  If we only preserve 4 bits of the result, then we're back to zero.
- Another way to calculate the 2's complement: Invert all the bits and add 1. So for 1 we have $0001 \rightarrow 1110 \rightarrow 1111$, which is the same as above.

TWO'S COMPLEMENT

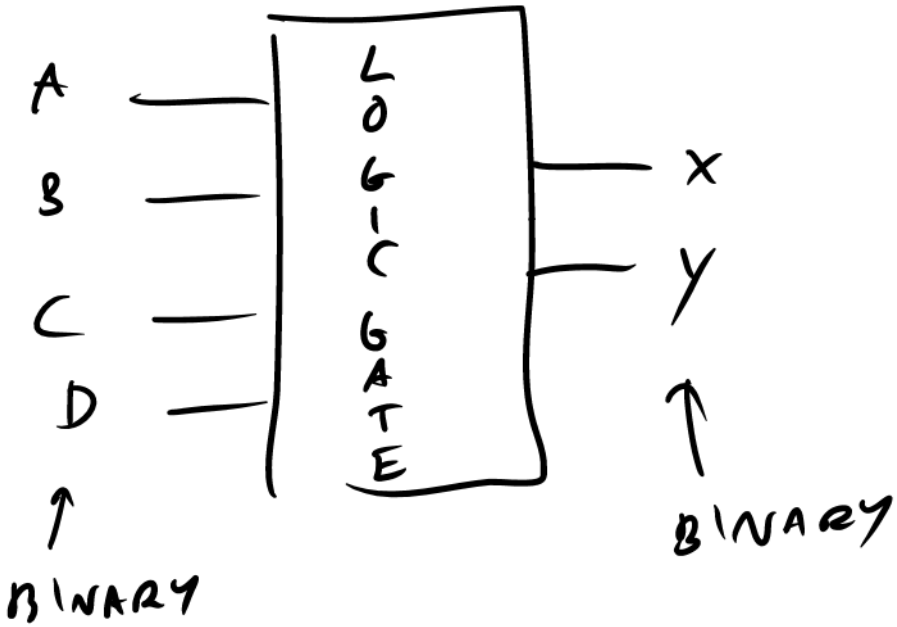


Two's Complement Example

- Example: Add $5 + (-4)$. $5 = 0101$ and -4 is formed by first forming 4, $4 = 0100$, inverting, 1011, and then adding 1, $0001 + 1011 = 1100$. Now if we add $5 + (-4) = 0101 + 1100 = 0001$, which shows that everything works!
- Note that the most significant bit is still 1 for all negative numbers, which is also a kind of sign bit.

What's a logic gate?

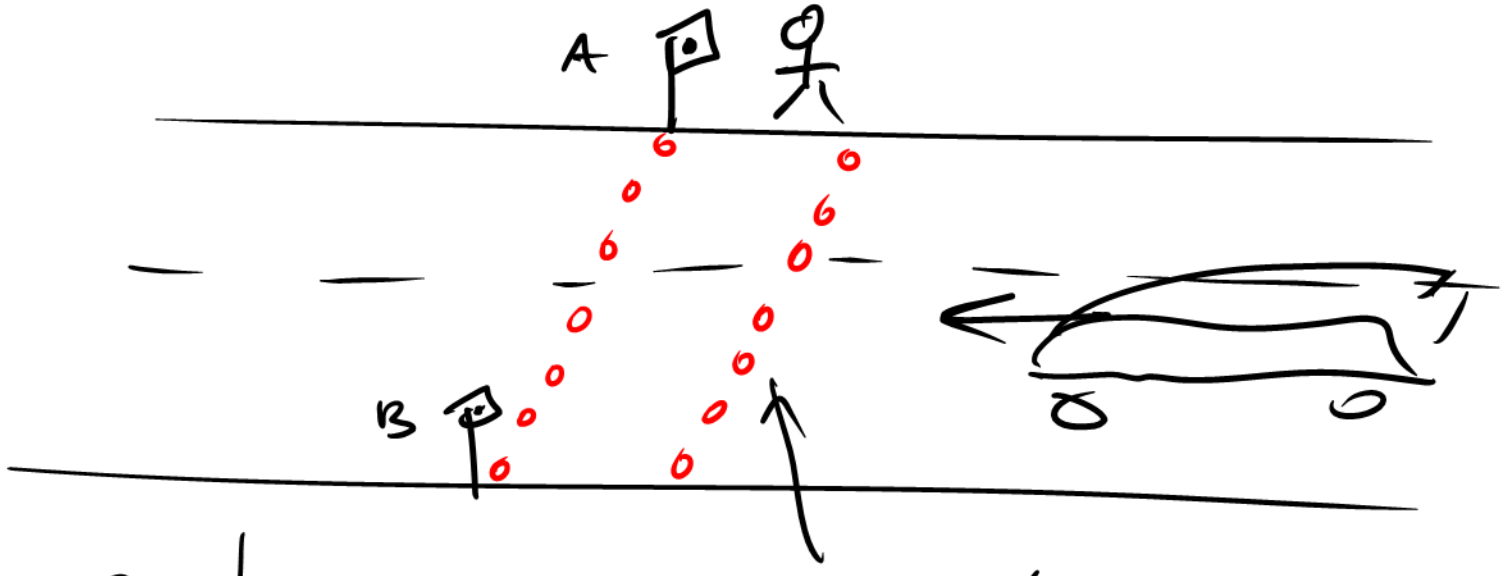
- First let's consider the *Digital Abstraction* in which we build circuits where the inputs and outputs are only interpreted as "high" or "low". Since the circuit only operates on two inputs, it's a binary circuit. In the binary circuit, the two inputs are binary digits "0" and "1".
- A logic gate performs a logic function. It has several inputs and it produces an output. The inputs and outputs can only take on values of {0,1}.
- By convention, all the inputs are put on the left and the output(s) are on the right
- The function is completely described by a truth table



$$x = f_1(A, B, C, D)$$
$$y = f_2(A, B, C, D)$$

A	B	C	D	x	Y
0	0	0	0	0	1
0	0	0	1	0	0
0	0	1	0	1	0
0	0	1	1	0	1
0	1	0	0	1	0
0	1	0	1	1	0
⋮					

LOGIC TO TURN PEDESTRIAN LIGHT GREEN



A	B	L
0	0	0
0	1	1
1	0	1
1	1	1

OR FUNCTION

AND, OR, and XOR

- The AND function is very common. The output is 1 only if both inputs are one. We write this relation as a product

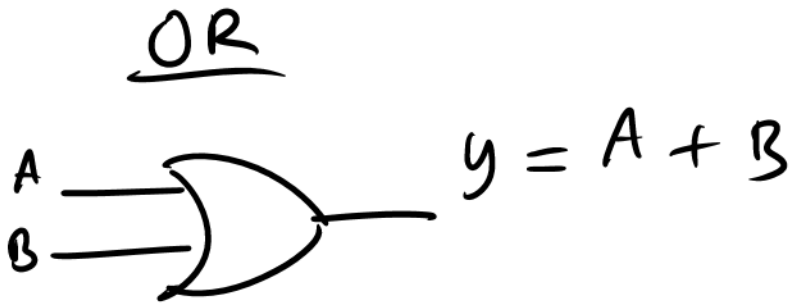
$$y = AB$$

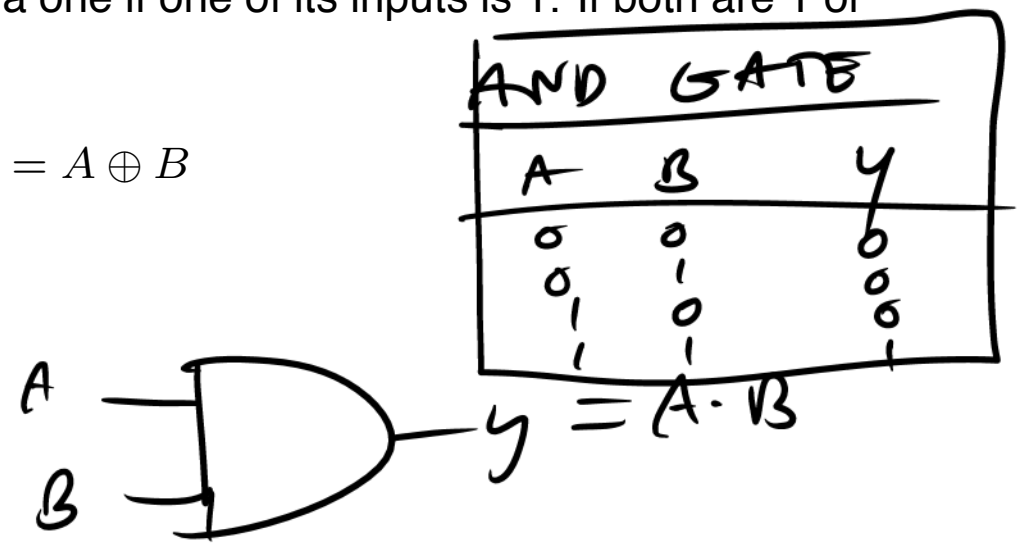
- The OR function produces a 1 as long as one input is 1. We write this relation as

$$y = A + B$$

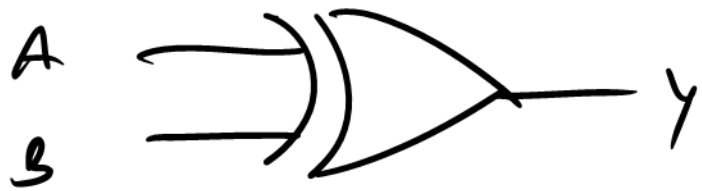
- Exclusive OR, or XOR, only produce a one if one of its inputs is 1. If both are 1 or 0, the output is 0. This is written as

$$y = A \oplus B$$





XOR



A	B	y
0	0	0
0	1	1
1	0	1
1	1	0

"OR"

"AND"

NEED AN INVERSE

NOT Gate

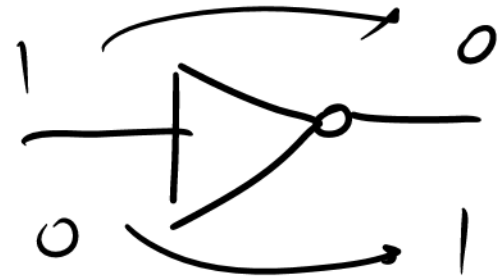
- The NOT gate inverts its input and it's not surprising that it's also known as an inverter. As we'll see, this is a very crucial function. The notation for NOT is

$$y = A'$$

$$y = \sim A$$

$$y = !A$$

$$y = \overline{A}$$



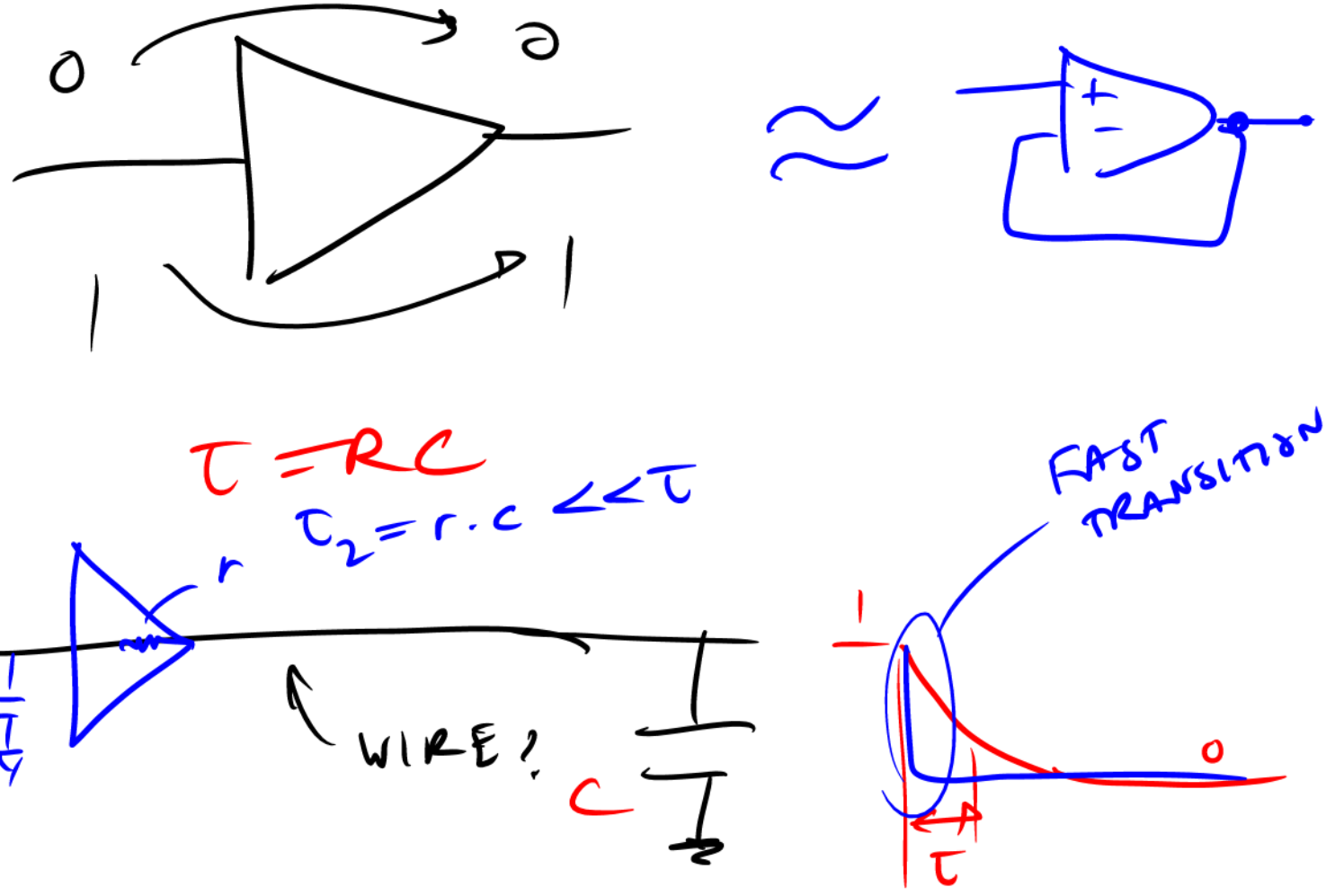
- Notice that the OR operation is the same as the AND operation if we NOT its inputs.

NOT, COMPLEMENT, INVERSE

A	B	OR	AND	$\overline{OR(\bar{A},\bar{B})}$	$\bar{A}$	$\bar{B}$	$OR(\bar{A},\bar{B})$
0	0	0	0	0	1	1	1
0	1	1	0	0	1	0	1
1	0	1	0	0	0	1	1
1	1	1	1	1	0	0	0

Buffer is more than a wire

- A buffer has the same output as its input. It seems rather trivial!
- From the perspective of logic, the buffer is the same as a wire, but in a real electrical circuit, a buffer is an amplifier that can be used to reduce the capacitive loading of a stage and help drive large capacitance. Thinks of the voltage follower.

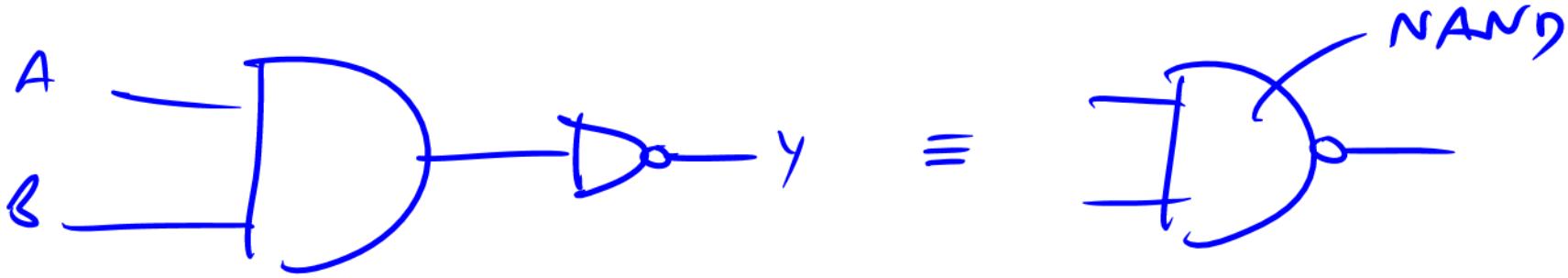


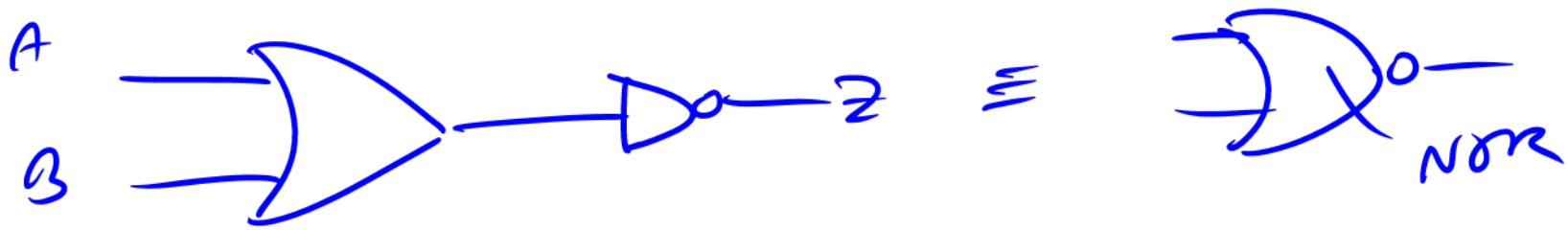
NAND and NOR

- The NAND and NOR gates are the same as AND and OR except the outputs are inverted. In real circuit implementation, they are often easier to realize than AND and OR.

NAND(A,B) = NOT(AND(A,B)) = Y

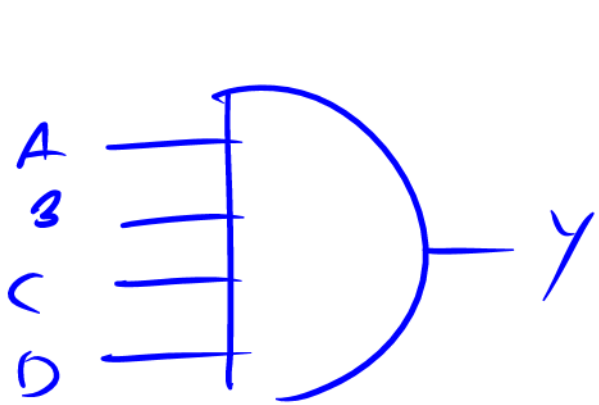
NOR(A,B) = NOT(OR(A,B)) = Z



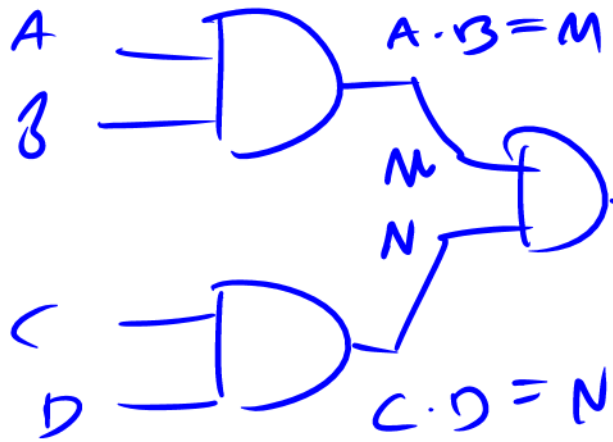


Multi Input Logic

- So far we have only dealt with two input logic gates. The concept generalizes to any number of inputs and outputs.
- Multi-input logic gates can be synthesized from two-input gates.



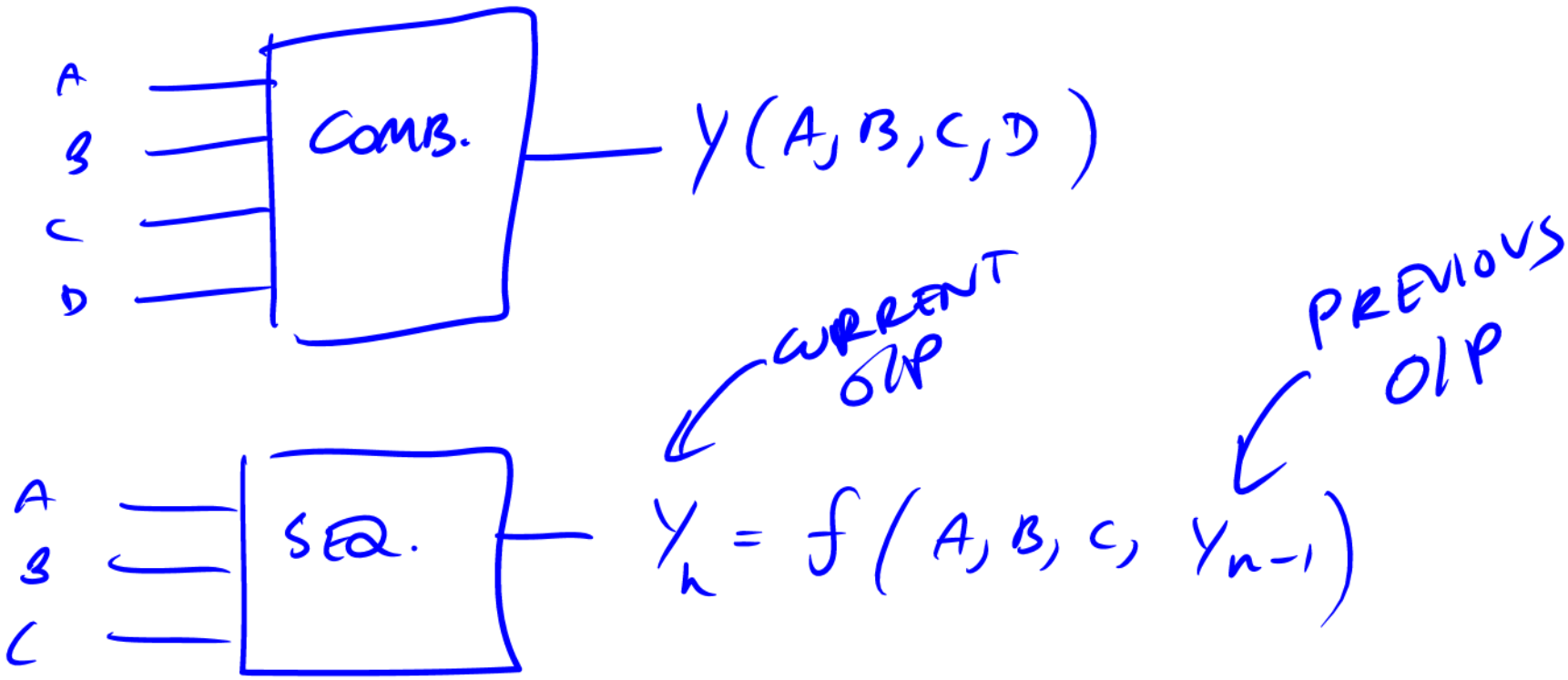
$$Y = \begin{cases} 1 & \text{IFF } A=B=C=D=1 \\ 0 & \text{otherwise} \end{cases}$$



$$Y = M \cdot N$$
$$= (A \cdot B) \cdot (C \cdot D)$$
$$= A \cdot B \cdot C \cdot D$$

Combinatorial versus Sequential

- The kinds of gates that we have been describing can be interconnected to build a combinatorial logic circuit. This is as opposed to a "sequential" logic circuit. The difference is that a combinatorial logic circuit is only a function of the current inputs. In other words, it has no memory. A sequential logic circuit, on the other hand, has memory and so the current output is a function of the current input and past inputs/outputs.

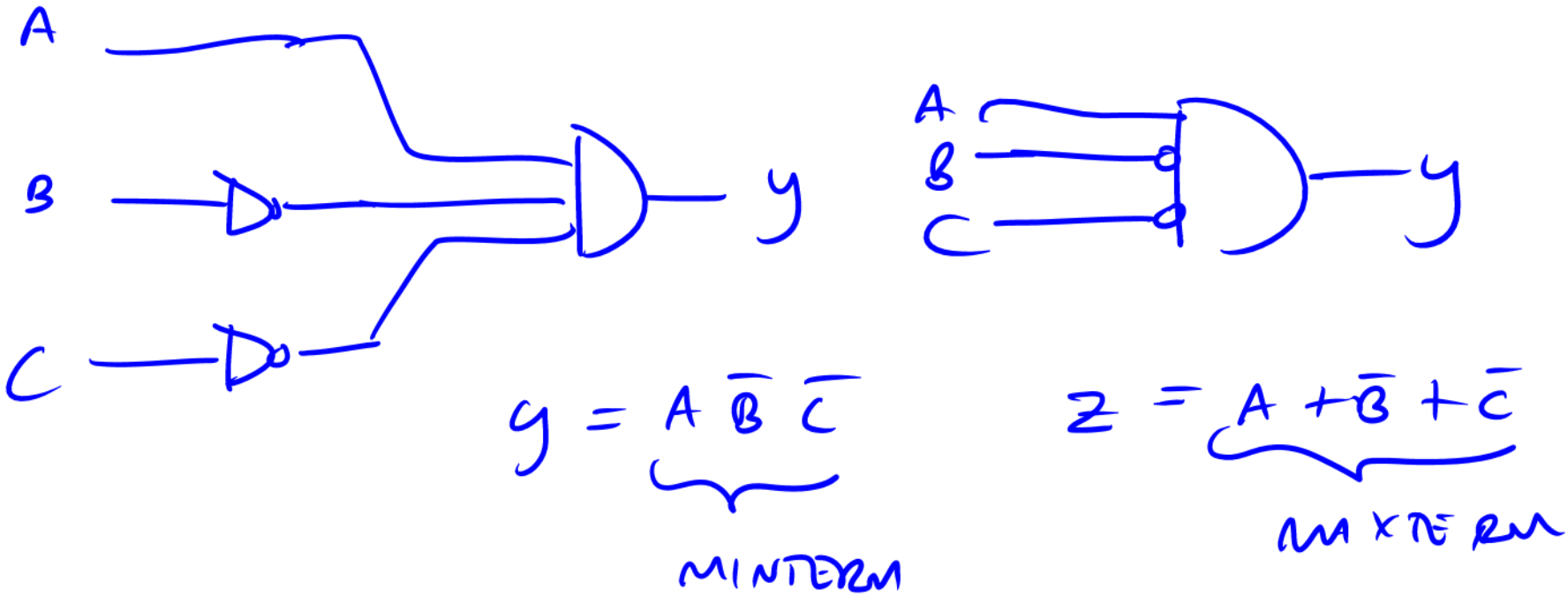


Boolean Equations

- The algebra of binary digits is known as Boolean algebra. It's very simple because there are only two possible inputs. Typical expressions are of the following form:

$$y = A\overline{B}\,\overline{C}$$

- When an expression is a product of all of it's inputs, it's known as a minterm. When a term is a sum of all of it's inputs, it's called a maxterm.



Order of Precedence

- How do you interpret an expression such as $y = A + BC$?

$$y = A + BC \stackrel{?}{=} (A\ OR\ B)\ AND\ C \stackrel{?}{=} A\ OR\ (B\ AND\ C)$$

- The rules of precedence determine the order of evaluation. The NOT operator has the highest precedence, followed by AND, and then OR.

$A + (BC)$

$(A + B) \cdot C$

NOT

AND

OR

ORDER

Sum of Products Form

- Any function can be written in a sum of products form. Simply examine the truth table and for each '1' output, write the corresponding minterm:

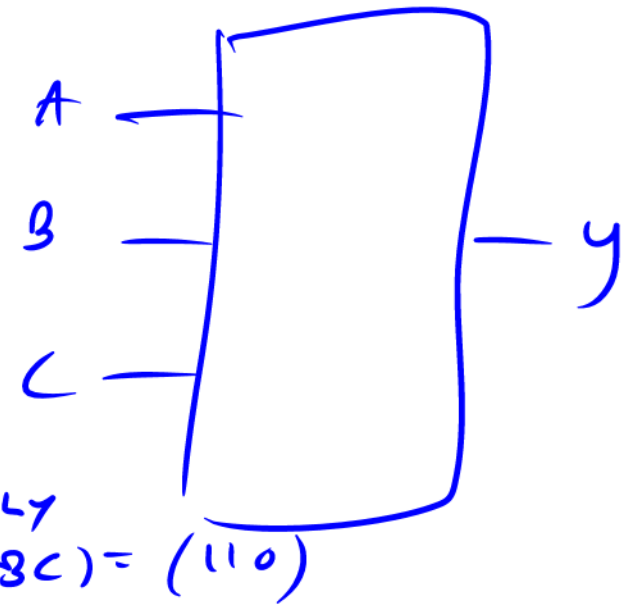
$$y = \overline{A}\,\overline{B}\,C + A\overline{B}C + AB\overline{C}$$

AND

+ $\overline{A}$ B C

A	B	C	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

← $\overline{A}$ B C



1 ONLY
IF (ABC) = (110)

Product of Sums

- Any function can also be written in the product of sums form. Simply examine the truth table and for each '0' input, write the corresponding maxterm:

$$y = (A + B + \overline{C})(A + \overline{B} + \overline{C})(\overline{A} + B + \overline{C})(\overline{A} + \overline{B} + C)$$

- When is the product of sums form better? When it produces a shorter expression. This happens of course when there are fewer zeros than ones in the truth table output.

$y = (A+B+\overline{C})(A+\overline{B}+C)(\overline{A}+B+C)$.
$(\overline{A}+\overline{B}+C)$

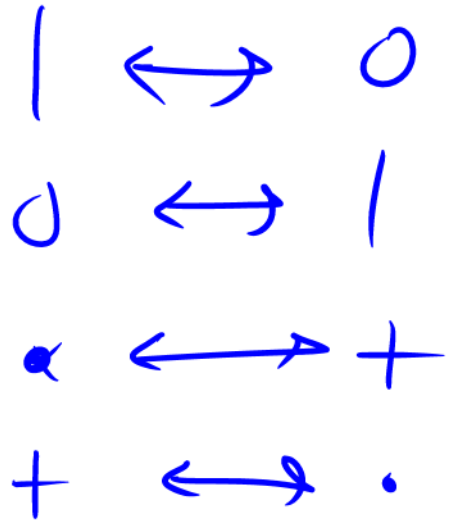
A	B	C	y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$(A + B + \overline{C})$ $(ABC) = 001$
$0 + 0 + 1 = 0$
$(A + \overline{B} + C)$
$(\overline{A} + B + C)$
$(\overline{A} + \overline{B} + C)$

Boolean Algebra Axioms

A1	$B = 0$ if $B \neq 1$	A1'	$B = 1$ if $B \neq 0$
A2	$\overline{0} = 1$	A2'	$\overline{1} = 0$
A3	$0 \cdot 0 = 0$	A3'	$1 + 1 = 1$
A4	$1 \cdot 1 = 1$	A4'	$0 + 0 = 0$
A5	$0 \cdot 1 = 1 \cdot 0 = 0$	A5'	$1 + 0 = 0 + 1 = 1$

- By definition, axioms cannot be proven. We use them as self-consistent definitions that can be used as givens when deriving theorems.



Boolean Algebra Theorems

T1	$B \cdot 1 = B$	T1'	$B + 0 = B$
T2	$B \cdot 0 = 0$	T2'	$B + 1 = 1$
T3	$B \cdot B = B$	T3'	$B + B = B$
T4	$\overline{\overline{B}} = B$	T4'	$\overline{\overline{B}} = B$
T5	$B \cdot \overline{B} = 0$	T5'	$B + \overline{B} = 1$

- Theorems can be proven from Axioms. Most of these are pretty obvious but you can also always check by putting in all possible inputs and verifying that both sides are equal.
- These theorems are very useful for simplifying results.

T1 $B = 0$ $0 \cdot 1 = B = 0$ ✓

$B = 1$ $1 \cdot 1 = 1 = B$ ✓

T5 $B \cdot \overline{B} = 0$ $B = 0 \Rightarrow 0 \cdot 1 = 0$ ✓

$B = 1 \Rightarrow 1 \cdot 0 = 0$ ✓

More Theorems

T6	$B \cdot C = C \cdot B$	
T6'	$B + C = C + B$	Commutativity
T7	$(B \cdot C) \cdot D = B \cdot (C \cdot D)$	
T7'	$(B + C) + D = B + (C + D)$	Associativity
T8	$(B \cdot C) + (B \cdot D) = B \cdot (C + D)$	
T8'	$(B + C) \cdot (B + D) = B + (C \cdot D)$	Distributivity
T9	$B \cdot (B + C) = B$	
T9'	$B + (B \cdot C) = B$	Covering
T10	$(B \cdot C) + (B \cdot \overline{C}) = B$	
T10'	$(B + C) \cdot (B + \overline{C}) = B$	Combining
T11	$(B \cdot C) + (\overline{B} \cdot D) + (C \cdot D) = B \cdot C + \overline{B} \cdot D$	
T11'	$(B + C) \cdot (\overline{B} + D) \cdot (C + D) = (B + C) \cdot (\overline{B} + D)$	Consensus
T12	$\overline{B_0 \cdot B_1 \cdot B_2 \ldots} = (\overline{B_0} + \overline{B_1} + \overline{B_2} + \cdots)$	
T12'	$\overline{B_0 + B_1 + B_2 + \cdots} = (\overline{B_0} \cdot \overline{B_1} \cdot \overline{B_2} \cdots)$	De Morgan's Theorem

- The theorems are the usual things we're familiar with from algebra, including commutativity, associativity, and distributivity. There's also covering, combining, and consensus, which are new.
- De Morgan's Theorem is extremely useful and should be studied carefully.

$\overline{A\,B} = \overline{A} + \overline{B}$

Equation Minimization

- Using the Axioms and Theorems of Boolean Algebra, we can simplify expressions. For example

$$y = \overline{A}\,\overline{B}\,\overline{C} + A\overline{B}\,\overline{C} + A\overline{B}C$$

- Can be simplified using distribution in the first two terms

$$y = \overline{B}\,\overline{C}(A + \overline{A}) + A\overline{B}C$$

- But $A + \overline{A} = 1$ and anything times 1 is itself

$$y = \overline{B}\,\overline{C}(1) + A\overline{B}C = \overline{B}\,\overline{C} + A\overline{B}C$$

- Can we do better? Notice the simplification resulted because the first two terms only differed by 1 bit, A. But the last two terms also differ in only 1 bit, C. We could have combined these two terms as well, giving us the same complexity in the minimized term.
- But we can also be clever and duplicate the middle term! That's because $A + A = A$. Show that we then can get down to

$$y = \overline{B}\,\overline{C} + A\overline{B}$$

From Logic to Gates

- It's now very easy to see how we can synthesize any Boolean expression using logic gates. For instance, to realize an expression in sum of products terms, we first form the miterms using AND gates and then we OR the outputs.

Other Possible Ouputs

- When an illegal output occurs in a logic gate, we denote that as ‘X’. Suppose we have a circuit where there is a contention between two logic gates. The output will then be an illegal output (not a well defined ‘0’ or ‘1’) and the circuit may not function correctly or as expected.
- The same notation is used for a “don’t care” output, in other words if we don’t care about a particular output, so be careful.
- Many logic gates have an “enable” input that can put the output into a third state, known as a floating or high impedance (high-‘Z’) state. This is known as a ‘Z’ state for this reason.
- A common block with this functionality is a tristate buffer. This is very useful in busses and other circuits where multiple gates drive the same output. Now a contention is not longer present as long as only one gate is enabled.

MUX

- A MUX or multiplexer simply selects one of its inputs based on the “select” input. You can think of it as a three input logic gate with the truth table shown below.
- A MUX can be implemented using a two-level logic, or using tri-state buffers.

4:1 MUX

- Higher order MUX circuits are also handy and can be realized in different ways.
- Notice that any truth table with 2 inputs can be realized by connecting the inputs of the 4:1 MUX to either supply or gnd !

Decoders

- A decoder is very handy when building a memory. There are N inputs and 2^N outputs. Only one output is selected at a time.