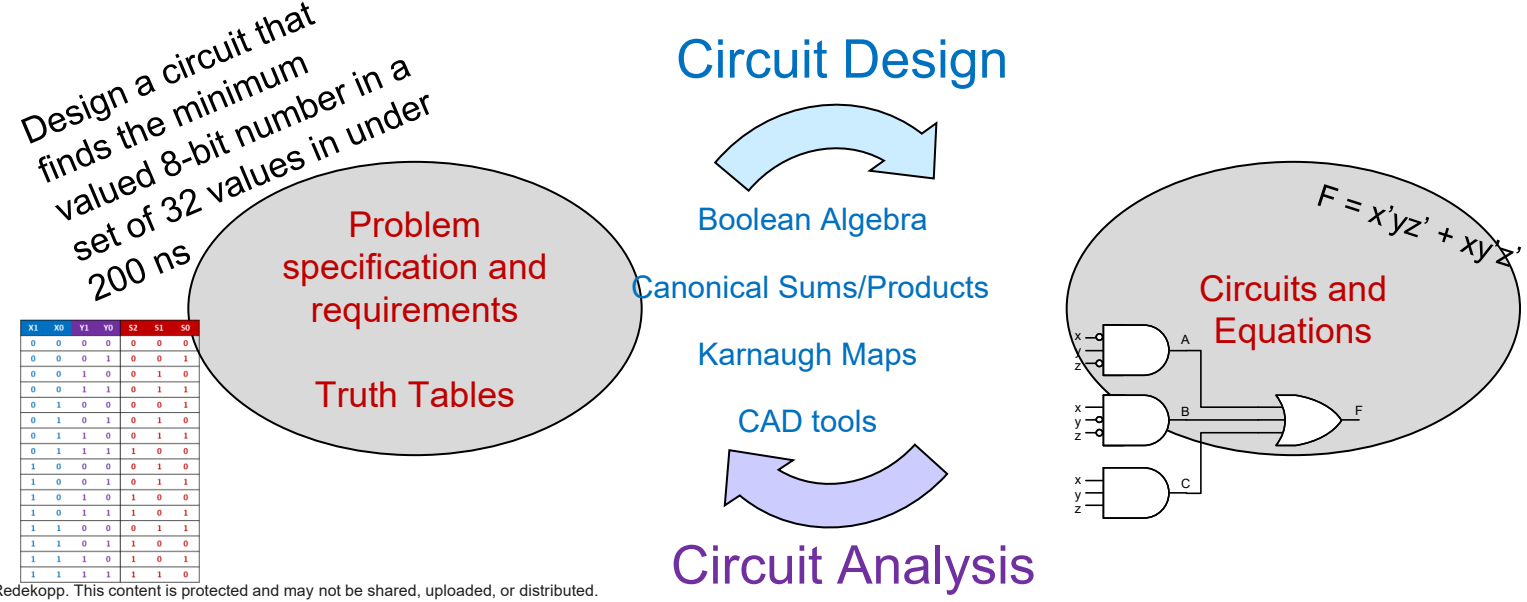
Unit 8b

Minterm and Canonical Sums
2- and 3-Variable Boolean Algebra Theorems
DeMorgan's Theorem
Simplification using Boolean Algebra

Logic Circuit Design and Analysis

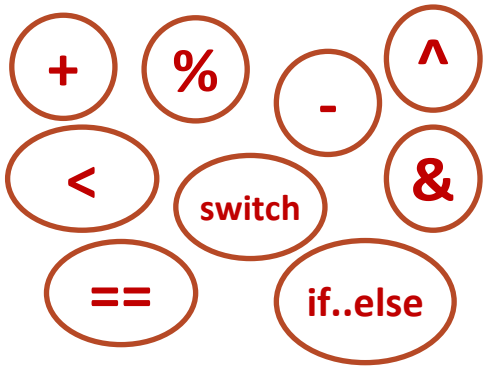
- There are two basic tasks as a digital design engineer…
  - Circuit Design/Synthesis: Take a set of requirements or functional descriptions and arrive at a logic circuit
  - Circuit Analysis: Given a logic circuit, find or verify the logic function it implements



DERIVING TRUTH TABLES

Putting Combinational Logic in Context

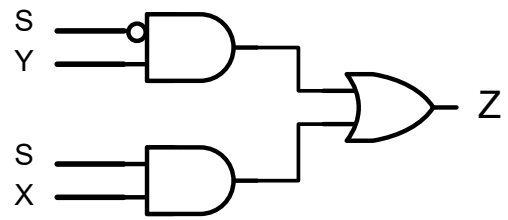
- Recall that **combinational logic…**
  - Refers to logic circuits where the output is function of only the ***present-time*** inputs
  - Can **ALWAYS** be described using a **truth table** then converted to a circuit
  - Can implement common **arithmetic, logic, comparison, and data selection operators** typically used by **software**.



Typical Software Operators

S	X	Y	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

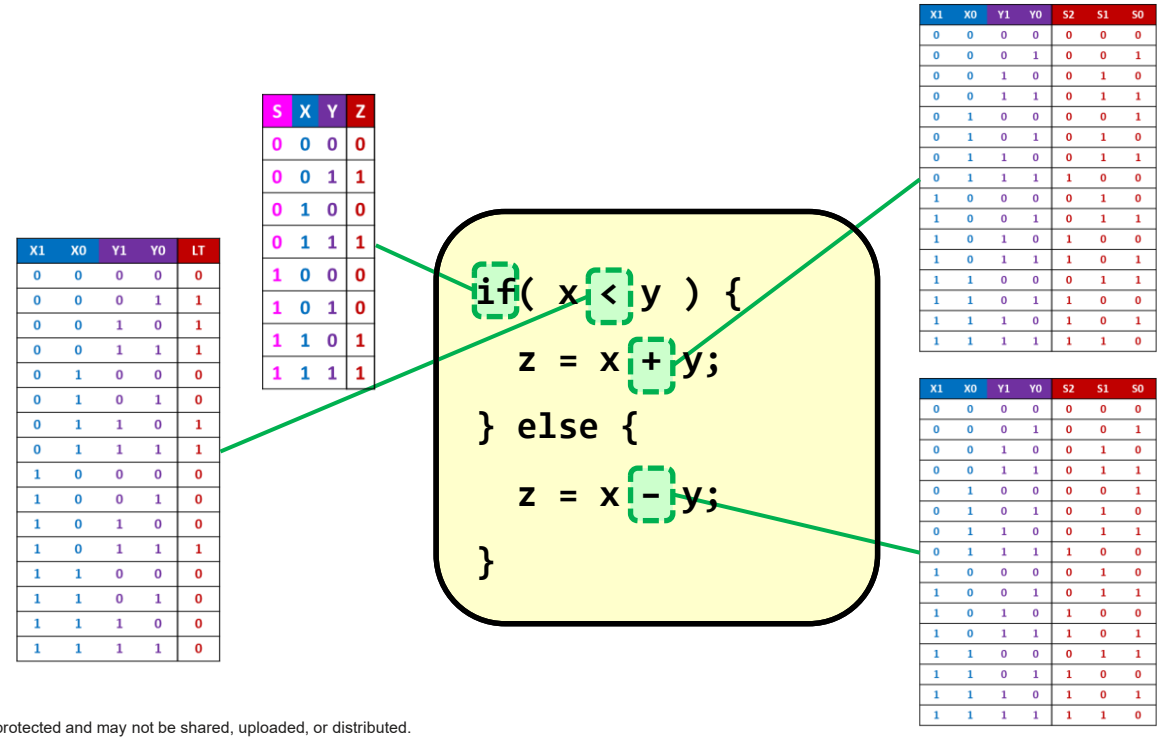
Truth Table



Logic Circuit

Seeing Truth Tables

- **Important skill**: Describe an operation or problem statement with a truth table.

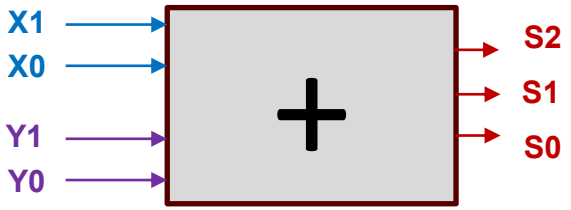


Combinational Logic Design Steps

- **Important skill**: Describe an operation or problem statement with a truth table.
- Steps:
  - Identify the inputs and how many bits each requires [*2-bit unsigned input range: 0-3 decimal*]
  - Determine how many output bits are needed by considering the numeric range of all possible outputs [*Min. Sum = 0+0=0, Max. Sum = 3+3=6. Thus, 3-bit output*]
  - Write a truth table with all 2^n combinations of the n input bits and come up with the desired output on your own [*4 total inputs => 16 combinations*]
  - Convert each output to a logic circuit using one of the synthesis methods we will teach you in the next two units. [*Focus of the next 2 units.*]

Task

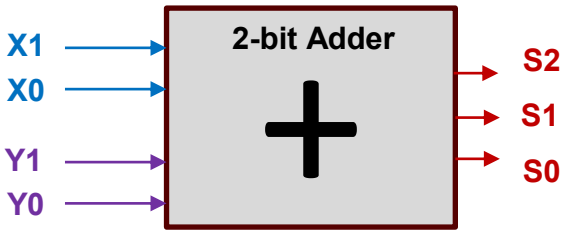
Build an addition operation for 2-bit unsigned binary numbers: X1X0 + Y1Y0



X1	X0	Y1	Y0	S2	S1	S0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
.	.	.	.	.	.	.
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

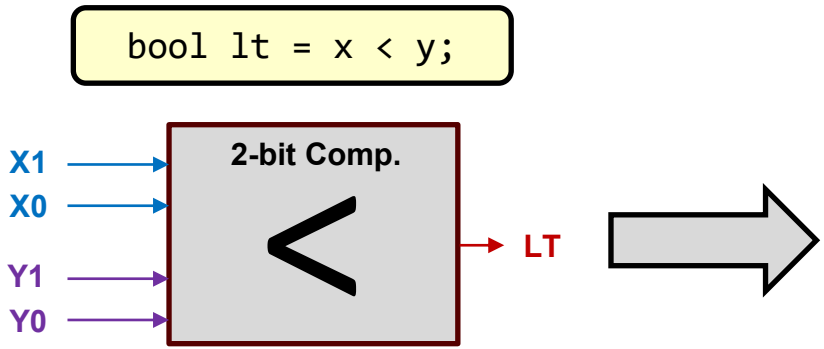
Addition

```
s = x + y;
```



X1	X0	Y1	Y0	S2	S1	S0	Descrip
0	0	0	0	0	0	0	0+0=0
0	0	0	1	0	0	1	0+1=1
0	0	1	0	0	1	0	0+2=2
0	0	1	1	0	1	1	0+3=3
0	1	0	0	0	0	1	1+0=1
0	1	0	1	0	1	0	1+1=2
0	1	1	0	0	1	1	1+2=3
0	1	1	1	1	0	0	1+3=4
1	0	0	0	0	1	0	2+0=2
1	0	0	1	0	1	1	2+1=3
1	0	1	0	1	0	0	2+2=4
1	0	1	1	1	0	1	2+3=5
1	1	0	0	0	1	1	3+0=3
1	1	0	1	1	0	0	3+1=4
1	1	1	0	1	0	1	3+2=5
1	1	1	1	1	1	0	3+3=6

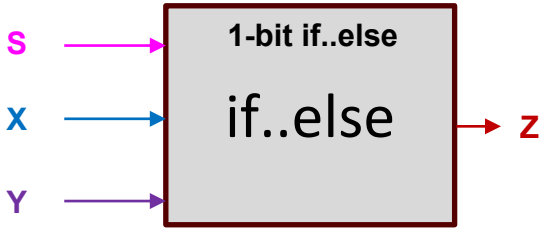
Comparison



X1	X0	Y1	Y0	LT	Descrip
0	0	0	0	0	0<0=F
0	0	0	1	1	0<1=T
0	0	1	0	1	0<2=T
0	0	1	1	1	0<3=T
0	1	0	0	0	1<0=F
0	1	0	1	0	1<1=F
0	1	1	0	1	1<2=T
0	1	1	1	1	1<3=T
1	0	0	0	0	2<0=F
1	0	0	1	0	2<1=F
1	0	1	0	0	2<2=F
1	0	1	1	1	2<3=T
1	1	0	0	0	3<0=F
1	1	0	1	0	3<1=F
1	1	1	0	0	3<2=F
1	1	1	1	0	3<3=F

If..Else (Ternary) Operator (aka "Mux")

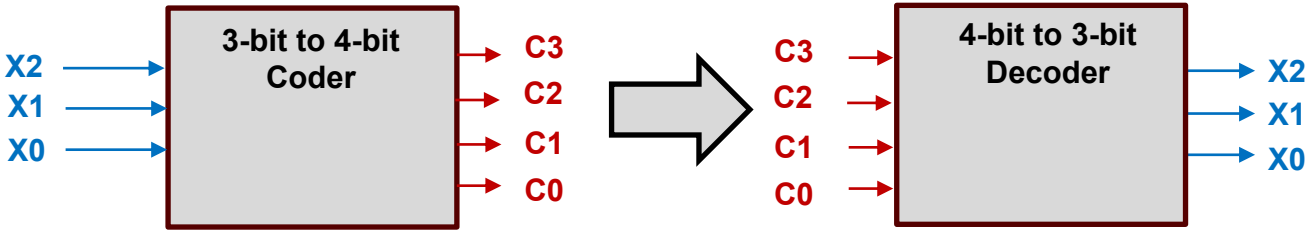
```
z = (s==1) ? x : y;

// if(s==1) { z=x; }
// else     { z=y; }
```



S	X	Y	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Coder / Decoder

- **Example**: Imagine we want to code (or "encrypt") 3-bit numbers and then decrypt/decode on the other side.

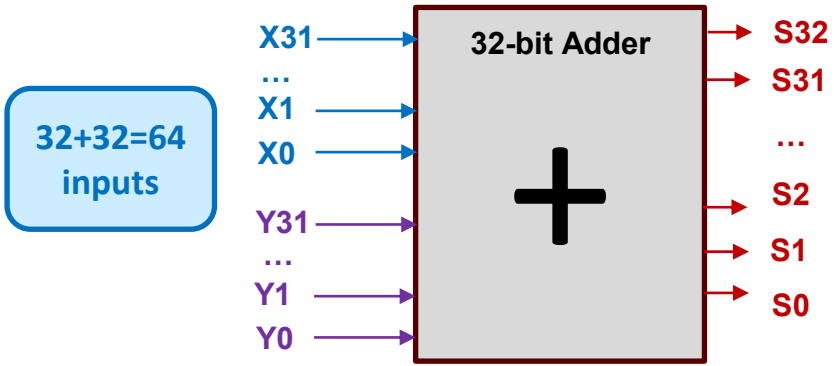


X2	X1	X0	C3	C2	C1	C0
0	0	0	1	0	1	1
0	0	1	0	1	1	0
0	1	0	1	0	1	0
0	1	1	1	1	0	1
1	0	0	0	1	1	1
1	0	1	0	0	0	1
1	1	0	1	1	1	1
1	1	1	0	1	0	1

C3	C2	C1	C0	X2	X1	X0
0	0	0	0	-	-	-
0	0	0	1	1	0	1
0	0	1	0	-	-	-
0	0	1	1	-	-	-
0	1	0	0	-	-	-
0	1	0	1	1	1	1
0	1	1	0	0	0	1
0	1	1	1	1	0	0
1	0	0	0	-	-	-
1	0	0	1	-	-	-
1	0	1	0	0	1	0
1	0	1	1	0	0	0
1	1	0	0	-	-	-
1	1	0	1	0	1	1
1	1	1	0	-	-	-
1	1	1	1	1	1	0

Scaling to Larger Input Sizes

- Typical hardware systems process much larger inputs (32- or 64-bit inputs). How would this process scale?
  - How many rows would a truth table for a 32-bit addition circuit require? $2^{64} \approx 16E18$
  - Clearly, that's not going to scale well for either human or computer-aided algorithms

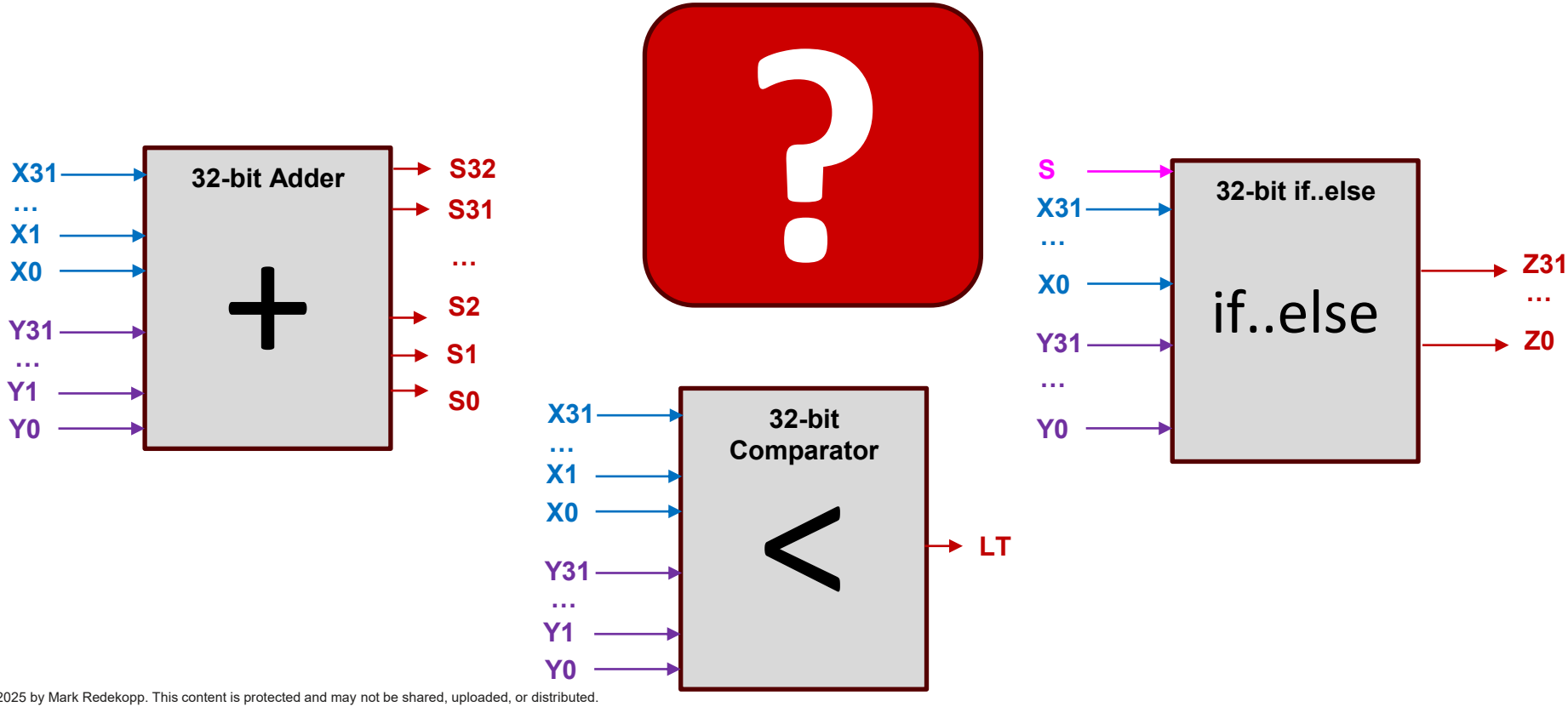


X31	…	…	Y0	S32	…	S0
0			0	0		0
0			1	0		1
0			0	0		0
0			1	0		1
0			0	0		1
0			1			0
0						1
0						0
1				0		0
1			1	0		1
1			0	1		0
1			1	1		1
1			0	0		1
1			1	1		0
1			0	1		1
1			1	1		0

TOO LARGE

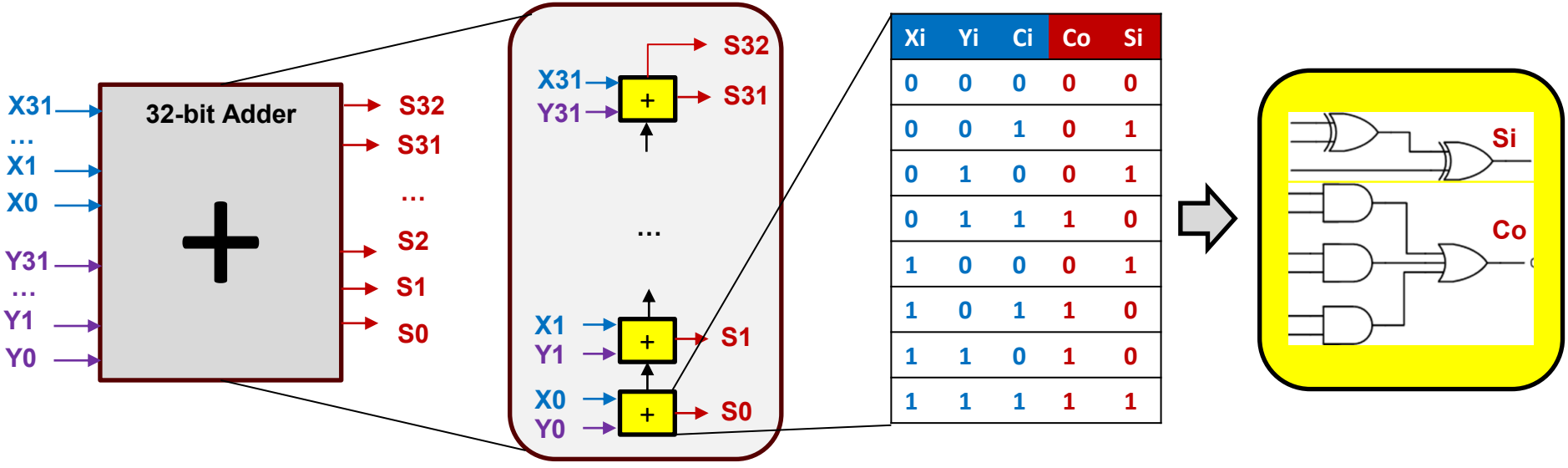
Dealing with Large Input Circuits [1]

- So how do we deal with large-input designs?



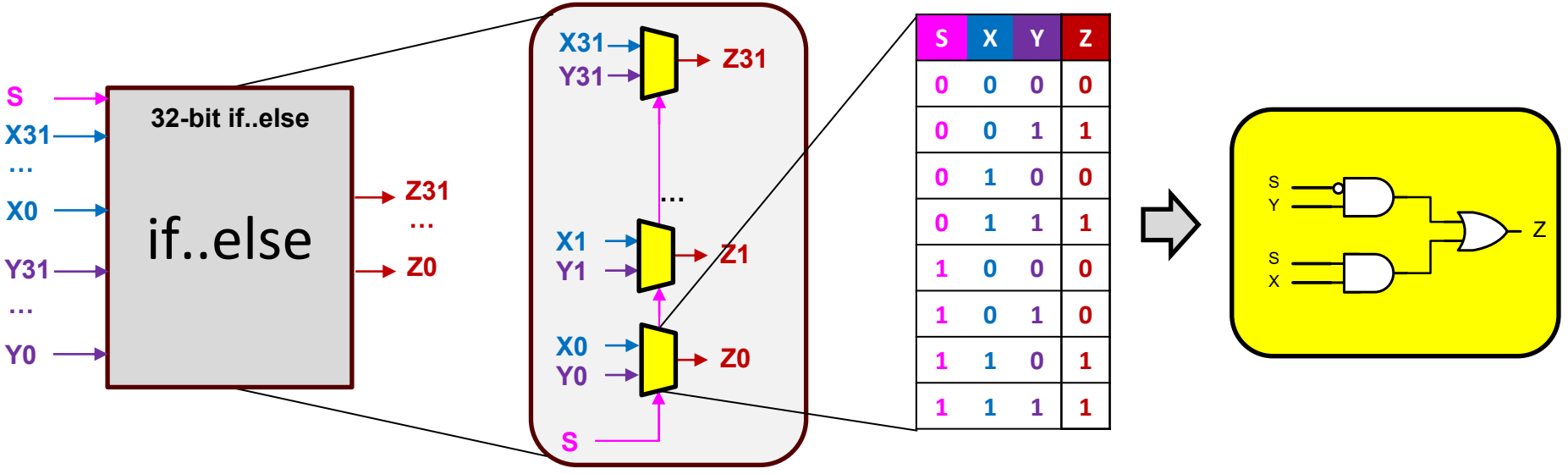
Dealing with Large Input Circuits [2]

- By decomposing the large circuits into **smaller, regularly structured blocks** where we CAN use truth tables and synthesis methods to find arrive at a circuit, we can simply connect these small blocks in a regular way to build the larger structure (like Legos)!



Dealing with Large Input Circuits [3]

- By decomposing the large circuits into **smaller, regularly structured blocks** where we CAN use truth tables and synthesis methods to find arrive at a circuit, we can simply connect these small blocks in a regular way to build the larger structure (like Legos)!

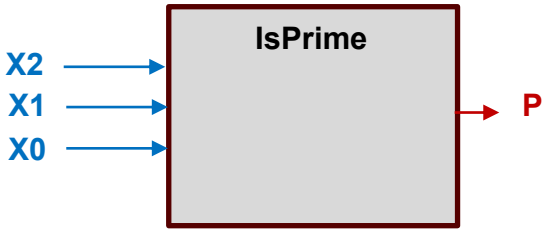


Synthesis: Truth Tables => Circuits

- So, now that we see how operations can be expressed as truth tables…
- …**how do we convert (aka "synthesize") those tables to logic circuits**?
- Let's use the following 2 examples for our discussion
  - 3-bit Prime Checker: Ad-hoc (irregular) design
  - Full Adder: Regular building block for larger adders

3-bit Prime Number Checker

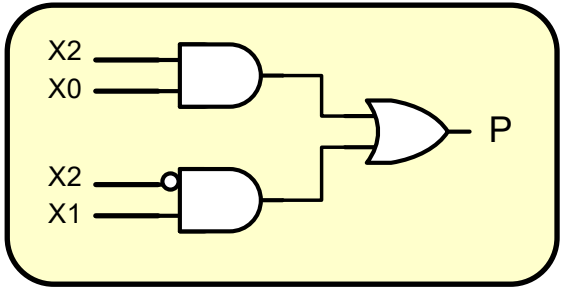
- Given a 3-bit, unsigned number check if it is prime (output 1) or not (output 0).



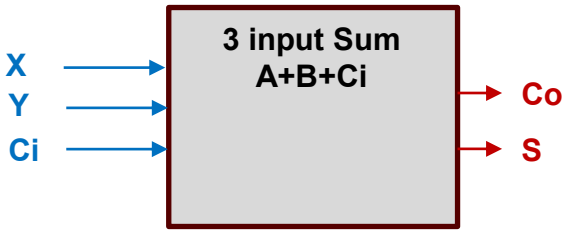
Primes between 0-7

X2	X1	X0	P
0	0	0	**0**
0	0	1	**0**
0	1	0	**1**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**0**
1	1	1	**1**

How can I find a circuit that implements this truth table?



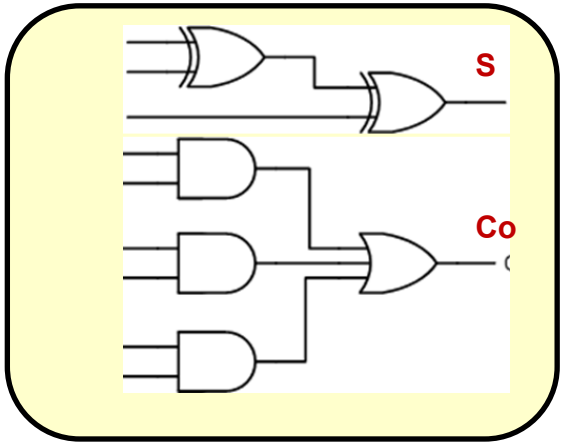
3-input Sum (aka Full Adder)



3 input Sum

X	Y	Ci	Co	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

How can I find a circuit that implements this truth table?



Ad-hoc vs. Regular Design Approach

- For **smaller** circuits or where there is **no regular pattern**, we use an ad-hoc approach of simply describing it with a truth table and trying to synthesize it using the methods we will teach you in the next two units.
  - Example: Prime number checker
- For **larger** circuits where there is a **discernable, regular pattern** we can decompose it into smaller blocks, design those smaller circuits using the techniques we will teach but then build the larger structure from these smaller blocks.
  - Example: Larger arithmetic of data selection circuits

The beginning of synthesis…

CHECKERS / DECODERS

Duality

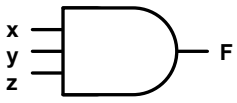
- As we progress in this unit, remember and look for the idea of duality at work
- Duality says: A new, true statement could be found from another by swapping:
  - 0 ⇔ 1 and
  - AND ⇔ OR

$$X + 1 = 1 \quad \Rightarrow \quad X \bullet 0 = 0$$

Original equation **Dual**

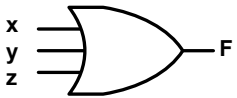
Gates

- Gates can have more than 2 inputs but the operations stay the same
  - AND = output = 1 if ALL inputs are 1
    - Outputs 1 for only 1 input combination
  - OR = output = 1 if ANY input is 1
    - Outputs 0 for only 1 input combination



X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

3-input AND

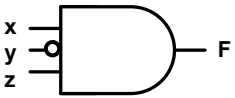


X	Y	Z	F
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

3-input OR

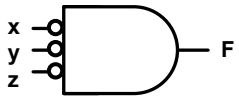
Checkers / Decoders

- An AND gate only outputs ‘1’ for 1 combination
  - That combination can be changed by adding inverters to the inputs
  - We can think of the AND gate as “checking” or “decoding” a specific combination and outputting a ‘1’ when it matches.



Add inverters to create an AND gate decoding (checking for) combination 101

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

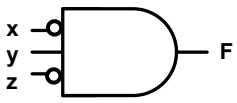


Add inverters to create an AND gate decoding (checking for) combination 000

X	Y	Z	F
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

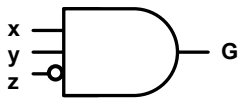
Checkers / Decoders

- Place inverters at the input of the AND gates such that
  - F produces ‘1’ only for input combination {x,y,z} = {010}
  - G produces ‘1’ only for input combination {x,y,z} = {110}



AND gate decoding (checking for) combination 010

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

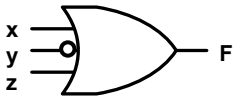


AND gate decoding (checking for) combination 110

X	Y	Z	G
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

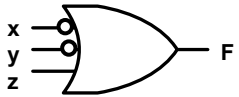
Checkers / Decoders

- An OR gate only outputs ‘0’ for 1 combination
  - That combination can be changed by adding inverters to the inputs
  - We can think of the OR gate as “checking” or “decoding” a specific combination and outputting a ‘0’ when it matches.



OR gate decoding (checking for) combination 010

X	Y	Z	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



OR gate decoding (checking for) combination 110

X	Y	Z	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

SYNTHESIZING LOGIC FUNCTIONS

Two Approaches: Minterms & Maxterms

- Because of duality, there are at least two ways to implement any circuit
- Using AND gate checkers
  (aka **product- or "min-" terms**)
  - Then combining their results with a single OR gate
- Using OR gate checkers
  (aka **sum- or "max-" terms**)
  - Then combining their results with a single AND gate

Using AND Gates (Minterms) to Implement Functions

- Given an any logic function, it can be implemented with the superposition of AND gate decoders/checkers

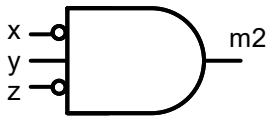
X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

An arbitrary truth table (could be any truth table)

Using AND Gates (Minterms) to Implement Functions

- Generate an **AND** gate checker ("minterm") for each combination *where the output of the logic function evaluates to **1** (i.e. F=1)*

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

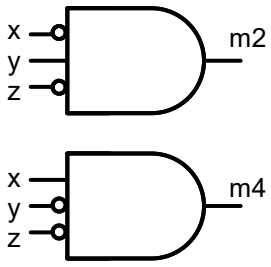


X	Y	Z	m2
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Using AND Gates (Minterms) to Implement Functions

- Generate an **AND** gate checker ("minterm") for each combination *where the output of the logic function evaluates to **1** (i.e. F=1)*

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

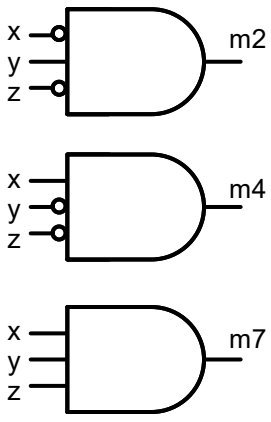


X	Y	Z	m2	m4
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	0	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

Using AND Gates (Minterms) to Implement Functions

- Generate an **AND** gate checker ("minterm") for each combination *where the output of the logic function evaluates to **1** (i.e. F=1)*

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

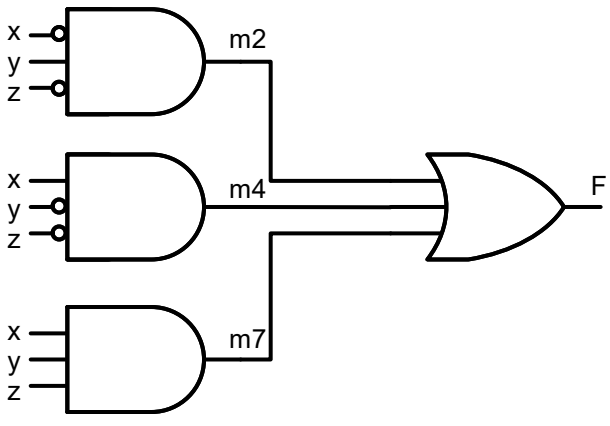


X	Y	Z	m2	m4	m7
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	1	0	0
0	1	1	0	0	0
1	0	0	0	1	0
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	0	1

Using AND Gates (Minterms) to Implement Functions

- Then, OR together all outputs of the AND gate checkers to form the overall function output

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

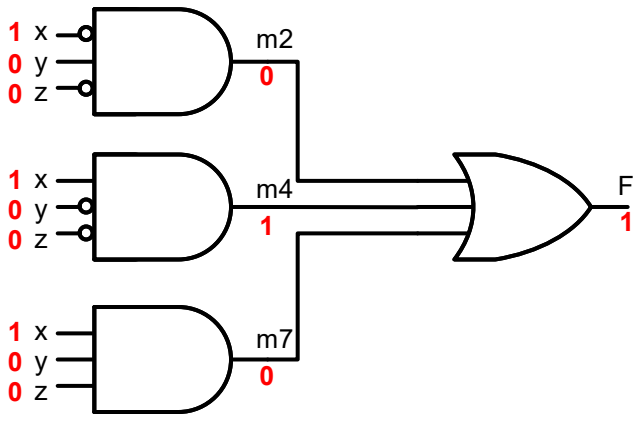


X	Y	Z	m2	m4	m7	F
0	0	0	0	0	0	**0**
0	0	1	0	0	0	**0**
0	1	0	1	0	0	**1**
0	1	1	0	0	0	**0**
1	0	0	0	1	0	**1**
1	0	1	0	0	0	**0**
1	1	0	0	0	0	**0**
1	1	1	0	0	1	**1**

Using AND Gates (Minterms) to Implement Functions

- Test it by plugging in combinations that should cause F=1
  - As long as one AND gate outputs 1, the output will be 1

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



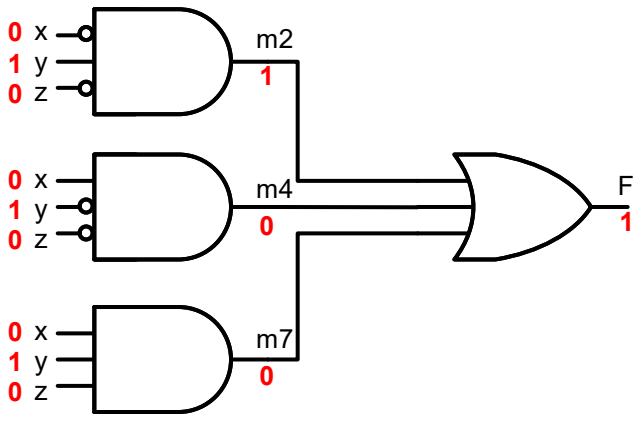
X	Y	Z	m2	m4	m7	F
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	1	0	0	1
0	1	1	0	0	0	0
1	0	0	0	1	0	1
1	0	1	0	0	0	0
1	1	0	0	0	0	0
1	1	1	0	0	1	1

F(1,0,0) = 1

Using AND Gates (Minterms) to Implement Functions

- Test it by plugging in combinations that should cause F=1
  - As long as one AND gate outputs 1, the output will be 1

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1



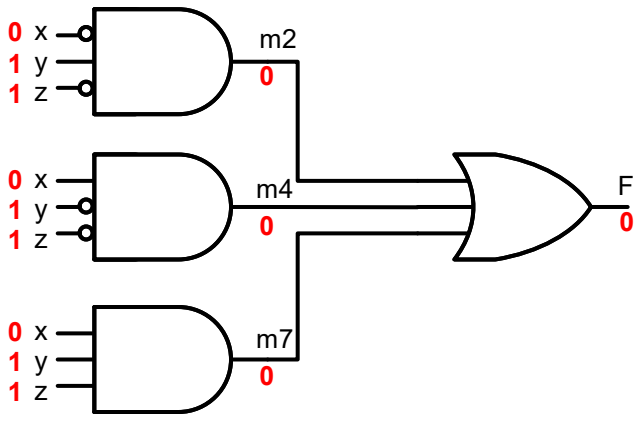
X	Y	Z	m2	m4	m7	F
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	1	0	0	1
0	1	1	0	0	0	0
1	0	0	0	1	0	1
1	0	1	0	0	0	0
1	1	0	0	0	0	0
1	1	1	0	0	1	1

F(0,1,0) = 1

Using AND Gates (Minterms) to Implement Functions

- Test it by plugging in combinations that should cause F=0
  - All AND gates output 0, thus the OR gate will output 0

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

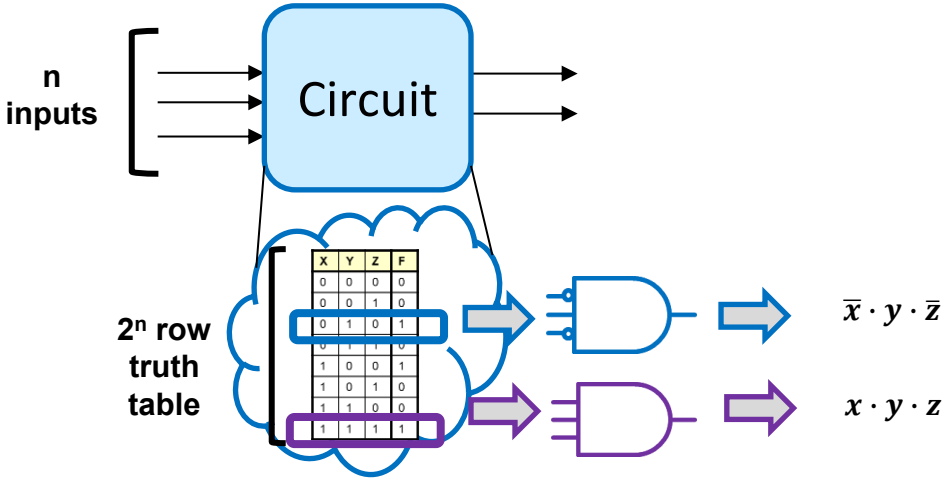


X	Y	Z	m2	m4	m7	F
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	1	0	0	1
0	1	1	0	0	0	0
1	0	0	0	1	0	1
1	0	1	0	0	0	0
1	1	0	0	0	0	0
1	1	1	0	0	1	1

F(0,1,1) = 0

Minterms

- An n-input combinational function can be described with 2^n row truth table
- Each row in the truth table (input combination) has a **unique logic expression** (i.e. an AND gate) that only evaluates to '1' for that combination
  - This logic expression is known as a **minterm**



Applying Minterms to Synthesize a Function

- Each numbered minterm checks whether the inputs are equal to the corresponding combination. When the inputs are equal, the minterm will evaluate to 1 and thus the whole function will evaluate to 1.

Primes between 0-7

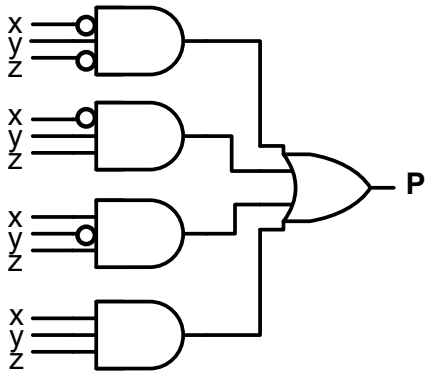
X	Y	Z	P	
0	0	0	**0**	
0	0	1	**0**	Use…
0	1	0	**1**	m2
0	1	1	**1**	m3
1	0	0	**0**	
1	0	1	**1**	m5
1	1	0	**0**	
1	1	1	**1**	m7

$P = m_2 + m_3 + m_5 + m_7$
$= x'yz' + x'yz + xy'z + xyz$

when x,y,z = {0,1,0} = 2 then
$P = 0'\cdot1\cdot0' + 0'\cdot1\cdot0 + 0\cdot1'\cdot0 + 0\cdot1\cdot0$
$= 1 + 0 + 0 + 0 = 1$

when x,y,z = {1,0,1} = 5 then
$P = 1'\cdot0\cdot1' + 1'\cdot0\cdot1 + 1\cdot0'\cdot1 + 1\cdot0\cdot1$
$= 0 + 0 + 1 + 0 = 1$

when x,y,z = {0,0,1} = 1 then
$P = 0'\cdot0\cdot1' + 0'\cdot0\cdot1 + 0\cdot0'\cdot1 + 0\cdot0\cdot1$
$= 0 + 0 + 0 + 0 = 0$

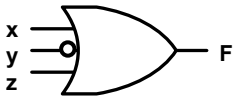


Using OR-gate checkers

AN ALTERNATIVE

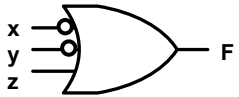
OR-Gate Checkers / Decoders

- An OR gate only outputs ‘0’ for a single combination
  - That combination can be changed by adding inverters to the inputs
  - We can think of the OR gate as “checking” or “decoding” a specific combination and outputting a ‘0’ when it matches.



OR gate decoding (checking for) combination 010

X	Y	Z	F
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1



OR gate decoding (checking for) combination 110

X	Y	Z	F
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

Using OR Checkers to Implement Functions

- Given an any logic function, it can be implemented with the superposition of OR-gate checkers/decoders (aka "sum- or max-terms")

X	Y	Z	G
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Using OR Checkers to Implement Functions

- Generate an **OR** gate checker ("maxterm") for each combination *where the output of the logic function evaluates to **0** (i.e. G=0)*

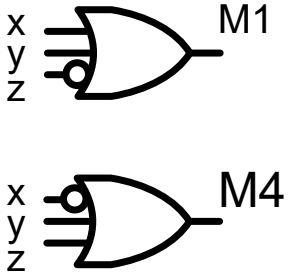
X	Y	Z	G
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

X	Y	Z	M1
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Using OR Checkers to Implement Functions

- Generate an **OR** gate checker ("maxterm") for each combination *where the output of the logic function evaluates to **0** (i.e. G=0)*

X	Y	Z	G
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

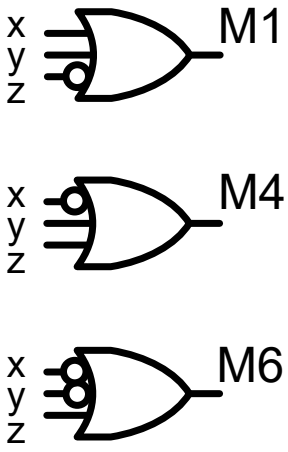


X	Y	Z	M1	M4
0	0	0	1	1
0	0	1	0	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Using OR Checkers to Implement Functions

- Generate an **OR** gate checker ("maxterm") for each combination *where the output of the logic function evaluates to **0** (i.e. G=0)*

X	Y	Z	G
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

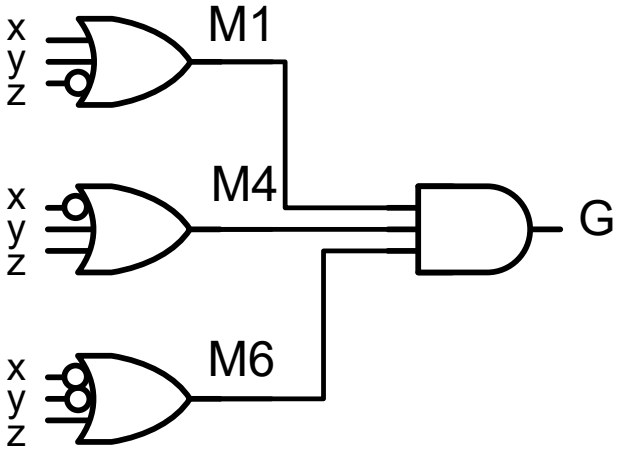


X	Y	Z	M1	M4	M6
0	0	0	1	1	1
0	0	1	0	1	1
0	1	0	1	1	1
0	1	1	1	1	1
1	0	0	1	0	1
1	0	1	1	1	1
1	1	0	1	1	0
1	1	1	1	1	1

Using OR Checkers to Implement Functions

- Then, AND together all outputs of the OR gate checkers to form the overall function output

X	Y	Z	G
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

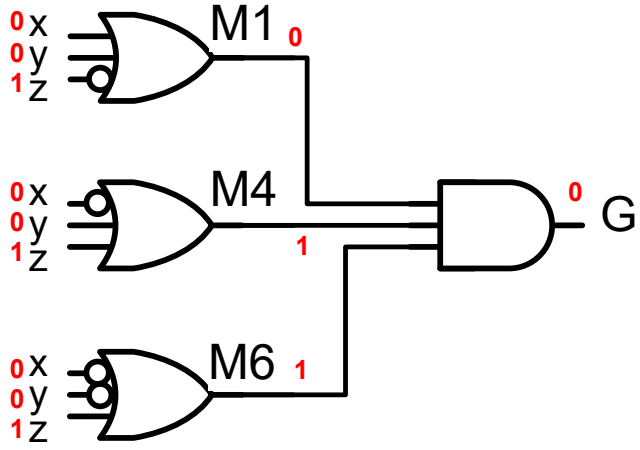


X	Y	Z	M1	M4	M6	G
0	0	0	1	1	1	**1**
0	0	1	0	1	1	**0**
0	1	0	1	1	1	**1**
0	1	1	1	1	1	**1**
1	0	0	1	0	1	**0**
1	0	1	1	1	1	**1**
1	1	0	1	1	0	**0**
1	1	1	1	1	1	**1**

Using OR Checkers to Implement Functions

- Test it by plugging in combinations that should cause G=0
  - As long as one OR gate outputs 0, the output will be 0

X	Y	Z	G
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



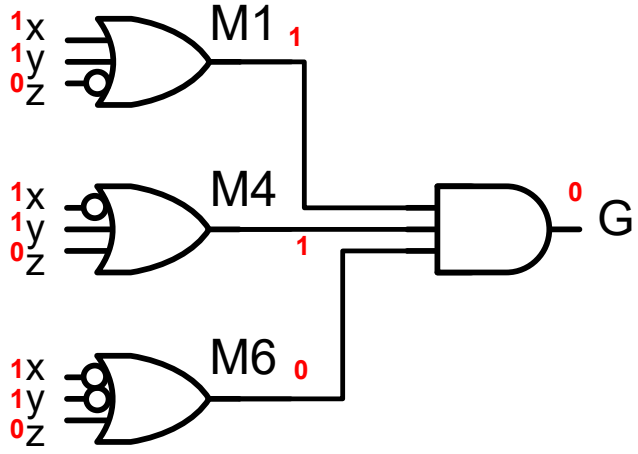
X	Y	Z	M1	M4	M6	G
0	0	0	1	1	1	**1**
0	0	1	0	1	1	**0**
0	1	0	1	1	1	**1**
0	1	1	1	1	1	**1**
1	0	0	1	0	1	**0**
1	0	1	1	1	1	**1**
1	1	0	1	1	0	**0**
1	1	1	1	1	1	**1**

F(0,0,1) = 0

Using OR Checkers to Implement Functions

- Test it by plugging in combinations that should cause G=0
  - As long as one OR gate outputs 0, the output will be 0

X	Y	Z	G
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



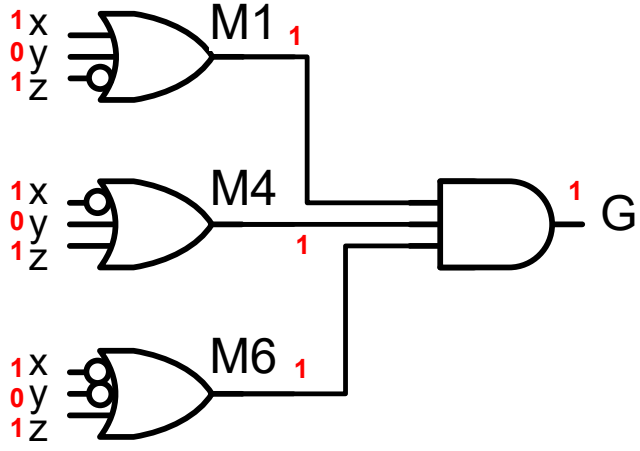
F(1,1,0) = 0

X	Y	Z	M1	M4	M6	G
0	0	0	1	1	1	**1**
0	0	1	0	1	1	**0**
0	1	0	1	1	1	**1**
0	1	1	1	1	1	**1**
1	0	0	1	0	1	**0**
1	0	1	1	1	1	**1**
1	1	0	1	1	0	**0**
1	1	1	1	1	1	**1**

Using OR Checkers to Implement Functions

- Test it by plugging in combinations that should cause G=1
  - All OR gates output 1, thus the AND gate will output 1

X	Y	Z	G
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



X	Y	Z	M1	M4	M6	G
0	0	0	1	1	1	**1**
0	0	1	0	1	1	**0**
0	1	0	1	1	1	**1**
0	1	1	1	1	1	**1**
1	0	0	1	0	1	**0**
1	0	1	1	1	1	**1**
1	1	0	1	1	0	**0**
1	1	1	1	1	1	**1**

F(1,0,1) = 1

Applying Maxterms to Synthesize a Function

- Each output that should produce a '0' can be checked-for with an OR gate
  - We refer to that OR-gate checker as a Maxterm of the function (M_i) where i represents the decimal value of the binary combination being checked
- We then AND together the maxterms

Primes between 0-7

X	Y	Z	P	Use…
0	0	0	**0**	M0
0	0	1	**0**	M1
0	1	0	**1**	
0	1	1	**1**	
1	0	0	**0**	M4
1	0	1	**1**	
1	1	0	**0**	M6
1	1	1	**1**	

$P = M_0 \cdot M_1 \cdot M_4 \cdot M_6$

$= (x+y+z) \cdot (x+y+z') \cdot (x'+y+z) \cdot (x'+y'+z)$

when x,y,z = {0,0,1} = 1 then

$P = (0+0+1) \cdot \mathbf{(0+0+1')} \cdot (0'+0+1) \cdot (0'+0'+1)$

$= 1 \cdot \mathbf{0} \cdot 1 \cdot 1 = \mathbf{0}$

when x,y,z = {1,1,0} = 6 then

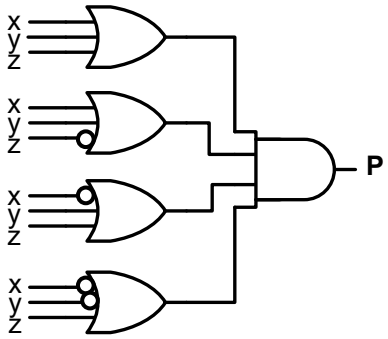
$P = (1+1+0) \cdot (1+1+0') \cdot (1'+1+0) \cdot \mathbf{(1'+1'+0)}$

$= 1 \cdot 1 \cdot 1 \cdot \mathbf{0} = \mathbf{0}$

when x,y,z = {1,1,1} = 7 then

$P = (1+1+1) \cdot (1+1+1') \cdot (1'+1+1) \cdot (1'+1'+1)$

$= 1 \cdot 1 \cdot 1 \cdot 1 = 1$



Defining Min-/Max-terms

- Below are the min-/max-terms for a function of 3-inputs: x,y,z
- Given a desired output, the designer could choose to include the appropriate set of min-/max-terms

			Inputs			Min-/Max-term Outputs							
Row #	Abbrev	Min-/Max-term Expression	x	y	z	m_0/ M_0	m_1/ M_1	m_2/ M_2	m_3/ M_3	m_4/ M_4	m_5/ M_5	m_6/ M_6	m_7/ M_7
0	m_0/M_0	x'•y'•z' / x+y+z	0	0	0	1/0	0/1	0/1	0/1	0/1	0/1	0/1	0/1
1	m_1/M_1	x'•y'•z / x+y+z'	0	0	1	0/1	1/0	0/1	0/1	0/1	0/1	0/1	0/1
2	m_2/M_2	x'•y•z' / x+y'+z	0	1	0	0/1	0/1	1/0	0/1	0/1	0/1	0/1	0/1
3	m_3/M_3	x'•y•z / x+y'+z'	0	1	1	0/1	0/1	0/1	1/0	0/1	0/1	0/1	0/1
4	m_4/M_4	x•y'•z' / x'+y+z	1	0	0	0/1	0/1	0/1	0/1	1/0	0/1	0/1	0/1
5	m_5/M_5	x•y'•z / x'+y+z'	1	0	1	0/1	0/1	0/1	0/1	0/1	1/0	0/1	0/1
6	m_6/M_6	x•y•z' / x'+y'+z	1	1	0	0/1	0/1	0/1	0/1	0/1	0/1	1/0	0/1
7	m_7/M_7	x•y•z / x'+y'+z'	1	1	1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	1/0

Finding simplified equations and circuits

2- AND 3-VARIABLE THEOREMS

Why Boolean Algebra

- We can now convert ***any truth table*** into an equation and circuit by using **minterms or maxterms**
- But minterms/maxterms yield the **LARGEST** equation/circuit
- By starting with sum of minterm (product of maxterm) form and then using **Boolean algebra** to simplify, we can arrive and smaller (even minimal) circuits

2 & 3 Variable Theorems

T6	X+Y = Y+X	**T6'**	X•Y = Y•X	Commutativity
T7	(X+Y)+Z = X+(Y+Z)	**T7'**	(X•Y)•Z = X•(Y•Z)	Associativity
T8	XY+XZ = X(Y+Z)	**T8'**	(X+Y)(X+Z) = X+YZ	Distribution & Factoring
T9	X + XY = X	**T9'**	X(X+Y) = X	Covering
T10	XY + XY' = X	**T10'**	(X+Y)(X+Y') = X	Combining
T11	XY + X'Z + YZ = XY + X'Z	**T11'**	(X+Y)(X'+Z)(Y+Z) = (X+Y)(X'+Z)	Consensus
DM	(X+Y)' = X'•Y'	**DM'**	(X•Y)'=X'+Y'	DeMorgan's

Proofs Through Other Theorems

T8	XY+XZ = X(Y+Z)	**T8'**	(X+Y)(X+Z) = X+YZ
T9	X + XY = X	**T9'**	X(X+Y) = X
T10	XY + XY' = X	**T10'**	(X+Y)(X+Y') = X
T11	XY + X'Z + YZ = XY + X'Z	**T11'**	(X+Y)(X'+Z)(Y+Z) = (X+Y)(X'+Z)

- Prove T9: X + XY = X
  - X(1 + Y) = X [T8]
  - X(1) = X [T2]
  - X = X [T1']
- Prove T10: XY + XY' = X
  - X(Y + Y') = X [T8]
  - X(1) = X [T2]
  - X = X [T1']
- Prove T10': (X+Y)(X+Y')=X
  - XX + XY' + XY + YY' = X [T8 / FOIL]
  - X + XY' + XY + 0 = X [T3 and T5']
  - X(1 + Y' + Y) = X [T1, T8]
  - X = X [T2, T1']

OR

X + YY' = X [T8']
X + 0 = X [T5']
X = X [T1]

Logic Synthesis

- Describe the function
  - Usually with a truth table
- Find the sum of minterm (or product of maxterm) expression
- Use Boolean Algebra (T8-T11) to find a simplified expression

Synthesize/Simplify Exercise 1

- Synthesize this function
  - First generate the canonical sum
  - Then use theorems to simplify

T8	AB+AC = A(B+C)	**T8'**	(A+B)(A+C) = A+BC	Distribution & Factoring
T9	A + AB = A	**T9'**	A(A+B) = A	Covering
T10	AB + AB' = A	**T10'**	(A+B)(A+B') = A	Combining
T11	AB + A'C + BC = AB + A'C	**T11'**	(A+B)(A'+C)(B+C) = (A+B)(A'+C)	Consensus

- P = X'YZ' + X'YZ + XY'Z + XYZ
  - X'Y(Z' + Z) + XZ(Y'+Y) [T8]
  - X'Y + XZ [T5, T1']

Primes between 0-7

X	Y	Z	P	
0	0	0	**0**	
0	0	1	**0**	Use…
0	1	0	**1**	m2
0	1	1	**1**	m3
1	0	0	**0**	
1	0	1	**1**	m5
1	1	0	**0**	
1	1	1	**1**	m7

Synthesize/Simplify Exercise 2a

- Synthesize each output separately
  - First generate the canonical prod.
  - Then use theorems to simplify

T8	XY+XZ = X(Y+Z)	**T8'**	(X+Y)(X+Z) = X+YZ	Distribution & Factoring
T9	X + XY = X	**T9'**	X(X+Y) = X	Covering
T10	XY + XY' = X	**T10'**	(X+Y)(X+Y') = X	Combining
T11	XY + X'Z + YZ = XY + X'Z	**T11'**	(X+Y)(X'+Z)(Y+Z) = (X+Y)(X'+Z)	Consensus

- G1 = M0•M1
  - (A+B+C)(A+B+C')
  - (A+B)+C•C' = (A+B) [T8'/T5' or T10']
  - **(A+B)** [Final Answer]
- G0 = M0•M2•M3
  - (A+B+C)(A+B'+C)(A+B'+C')
  - (A+B+C)(A+B'+C) (A+B'+C)(A+B'+C') [Use T3 to replicate]
  - [(A+C)+B•B')][(A+B')+C•C'] = (A+C)(A+B') [T8'T5' or T10']
  - **(A+C)(A+B')** [Final Answer]

A	B	C	G1	G0
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**1**	**0**
0	1	1	**1**	**0**
1	0	0	**1**	**1**
1	0	1	**1**	**1**
1	1	0	**1**	**1**
1	1	1	**1**	**1**

Encode the highest input ID (ie. 3, 2, or 1) that is ON (=1)

Synthesize/Simplify Exercise 2b

- Synthesize each output separately
  - First generate the canonical sum
  - Then use theorems to simplify

T8	XY+XZ = X(Y+Z)	**T8'**	(X+Y)(X+Z) = X+YZ	Distribution & Factoring
T9	X + XY = X	**T9'**	X(X+Y) = X	Covering
T10	XY + XY' = X	**T10'**	(X+Y)(X+Y') = X	Combining
T11	XY + X'Z + YZ = XY + X'Z	**T11'**	(X+Y)(X'+Z)(Y+Z) = (X+Y)(X'+Z)	Consensus

- G1 = m2 + m3 + m4 + m5 + m6 + m7
  - A'BC' + A'BC + AB'C' + AB'C + ABC' + ABC
  - [T3 (A = A + A) allows us to replicate m_6 and m_7 (ABC' + ABC)]
  - A'BC' + A'BC + ABC' + ABC + AB'C' + AB'C + ABC' + ABC
  - B(A'C' +A'C + AC' + AC) + A(B'C' + B'C + BC' + BC) [T8]
  - B(A'(C'+C) + A(C'+C)) + A(B'(C+C') + B(C' + C)) [T8/T5 or T10]
  - B(A'+ A) + A(B' + B) [T8/T5] = **B + A** [Final Answer]
- G0 = m1 + m4 + m5 + m6 + m7
  - A'B'C + AB'C' + AB'C + ABC' + ABC
  - A'B'C + AB'C + AB'C' + AB'C + ABC' + ABC [Use T3 to replicate m_5]
  - B'C(A' + A) + A(B'C' + B'C + BC' + BC) [T8]
  - B'C + A [T8/T5 or T10] = **B'C + A** [Final Answer]

A	B	C	G1	G0
0	0	0	**0**	**0**
0	0	1	**0**	**1**
0	1	0	**1**	**0**
0	1	1	**1**	**0**
1	0	0	**1**	**1**
1	0	1	**1**	**1**
1	1	0	**1**	**1**
1	1	1	**1**	**1**

Encode the highest input ID (ie. 3, 2, or 1) that is ON (=1)

Synthesize/Simplify Exercise 3 (Optional)

- Synthesize each output separately
  - First generate the canonical sum
  - Then use theorems to simplify

T8	AB+AC = A(B+C)	**T8'**	(A+B)(A+C) = A+BC	Distribution & Factoring
T9	A + AB = A	**T9'**	A(A+B) = A	Covering
T10	AB + AB' = A	**T10'**	(A+B)(A+B') = A	Combining
T11	AB + A'C + BC = AB + A'C	**T11'**	(A+B)(A'+C)(B+C) = (A+B)(A'+C)	Consensus

- Co = m3 + m5 + m6 + m7
  - X'YCi + XY'Ci + XYCi' + XYCi
  - [Replicate m_7 twice so we can simplify the others: m3 + m7 + m5 + m7 + m6 + m7]
  - X'BCi + XYCi + XY'Ci + XYCi + XYCi' + XYCi
  - YCi(X'+X) + XCi(Y'+Y) + XY(Ci'+Ci) = **YCi + XCi + XY [Final Answer]**
- S = m1 + m2 + m4 + m7
  - X'Y'Ci + X'YCi' + XY'Ci' + XYCi [Not much to factor that will cause simplification]
  - X'(Y'Ci + YCi') + X(Y'Ci' + YCi) [But write the truth tables of Y'Ci+YCi' and Y'Ci'+YCi]
  - X'(Y⊕Ci) + X(Y⊕Ci)' [But if we let W=Y⊕Ci then we have X'W + XW'…write its TT]
  - **X ⊕ Y ⊕ Ci [Final Answer]**

3-bit Sum (Full Adder)

X	Y	Ci	Co	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

How to make faster circuits…

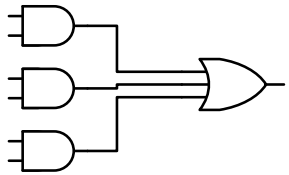
DEFINITIONS, EXPRESSION FORMS, SPEED, AND DEMORGAN'S

Definitions

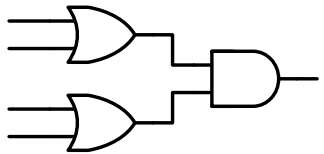
- **Literal:** A single bit variable or its inverse
  - **Good:** `x, y', SLEEPING'`
  - **Bad:** `(x+y)`
- **Product Term:** A single literal by itself or an AND'ing (not NAND'ing) of literals
  - **Good:** `z, x•y, AWAKE•LISTENING•THINKING`
  - **BAD:** `(x•y)', AWAKE•(LISTENING+THINKING)`
  - **The minterms we defined earlier are product terms where EACH input variable of a function is 1 literal in the product term**
- **Sum Term:** A single literal by itself or an OR'ing (not NOR'ing) of literals
  - **Good:** `z, x'+y, CURIOUS+PERSISTENT`
  - **BAD:** `(x+y)', TIRED•(BORED+SLEEPY)`
  - **The maxterms we defined earlier are sum terms where EACH input variable of a function is 1 literal in the sum term**

Expression/Circuit Forms

- **SOP (Sum of Products) Form**: An **SOP** expression is a logical sum (OR) of product terms
  - Correct Examples: [x'•y'•z + w + a'•b•c], [w + x'•z•y + y'z]
  - Incorrect Examples: [x'•y•z + w•(a+b)], [x•y + (y'•z)']
- SOP equations yield **2-level circuits** with AND gates in the 1st level with an OR gate in the 2nd (aka **AND-OR** circuits)
- **POS (Product of Sums) Form**: A **POS** expression is a logical product (AND) of sum terms.
  - Correct Examples: [(x+y'+z) • (w'+z) • (a)], [z'•(x+y)•(w'+y)]
  - Incorrect Examples: [x' + y•(x+w)], [(x+y)•(x+z)']
- POS equations yield **2-level circuits** with OR gates in the 1st level with an AND gate in the 2nd (aka **OR-AND** circuits)
- 1 level circuits (i.e. a single gate) are generally BOTH SOP and POS



SOP ⇔ AND-OR
(Sum of products yields AND-OR circuits)



POS ⇔ OR-AND
(Product of sums yields OR-AND circuits)

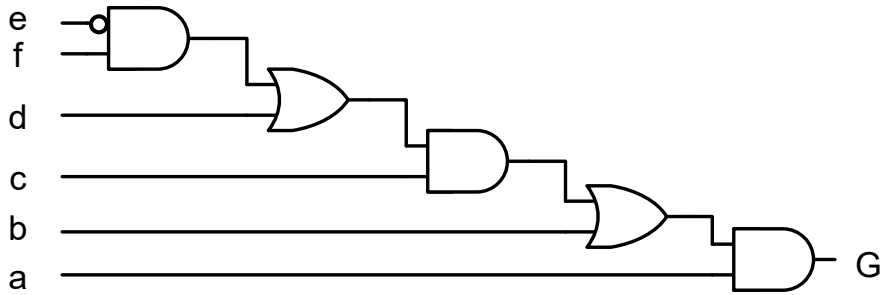
Check Yourself

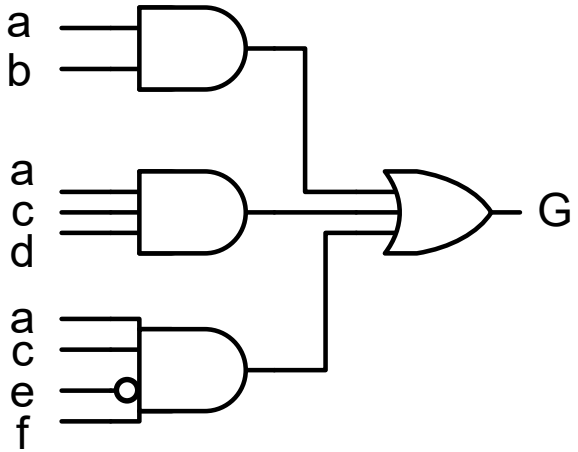
Expression	SOP / POS / Both / Neither
w•x•(y•z)’ + xy’z + w	Neither (Can’t have complements of sub-expressions…only literal)
xy+xz+(w’yz)	SOP (parentheses are unnecessary)
(w+y’+z)(w+x)	POS
(w+y)x(w’+z)	POS (a single literal is a sum term)
wy + wy + xy’	SOP (redundancy doesn’t matter)
w+x+y	Both (individual literals are both a product and sum term)

Factoring and Distributing (Size vs. Speed)

- Factoring decreases size
- Distributing decreases levels of logic (delay)

$$G = a \cdot (b + (c \cdot (d + \bar{e}f))) = ab + acd + ac\bar{e}f$$





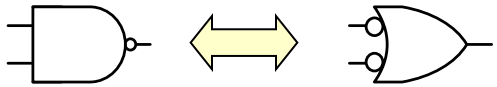
DeMorgan's Theorem

- Inverting output of an AND gate = inverting the inputs of an OR gate
- Inverting output of an OR gate = inverting the inputs of an AND gate

A function's inverse is equivalent to inverting all the inputs and changing AND to OR and vice versa

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

$\overline{A \bullet B}$

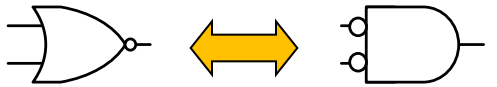


$\overline{A} + \overline{B}$

A	B	Out
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

$\overline{A + B}$



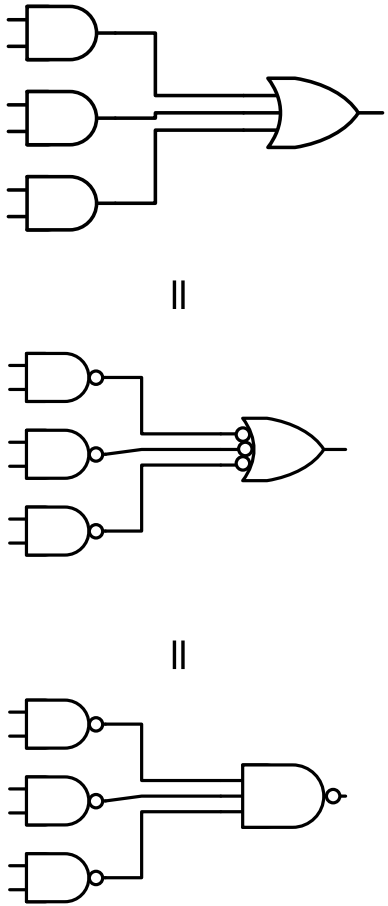
$\overline{A} \bullet \overline{B}$

A	B	Out
0	0	1
0	1	0
1	0	0
1	1	0

Analogy: Turning a gate "inside-out" (like your socks).

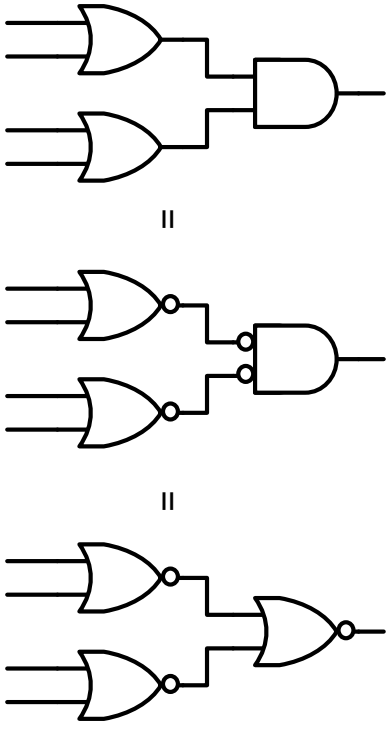
AND-OR / NAND-NAND

- Canonical Sums yield
  - AND-OR Implementation
  - NAND-NAND Implementation
- Recall inverting gates such as NAND gates may be "faster" or have desirable properties vs. typical AND/OR gates



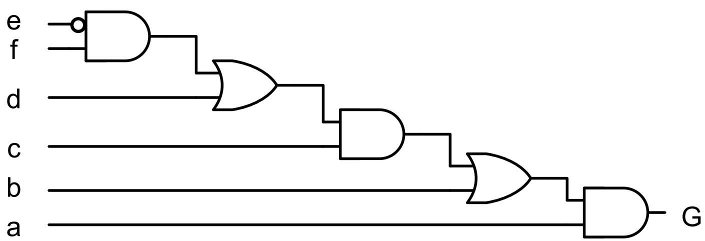
OR-AND / NOR-NOR

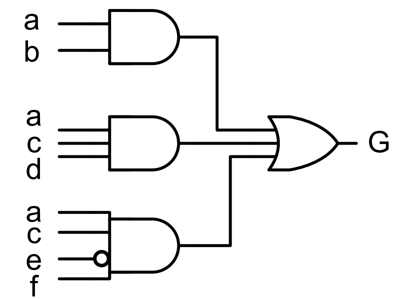
- Canonical Products yield
  - OR-AND Implementation
  - NOR-NOR Implementation
- Recall inverting gates such as NOR gates may be "faster" or have desirable properties vs. typical AND/OR gates

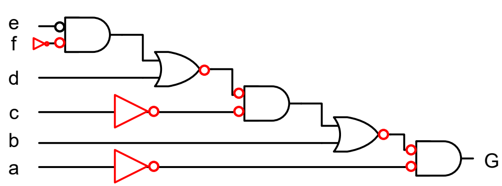


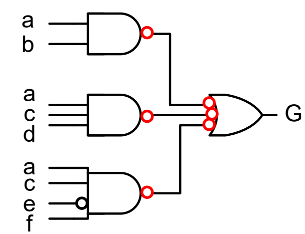
DeMorgan's Practice

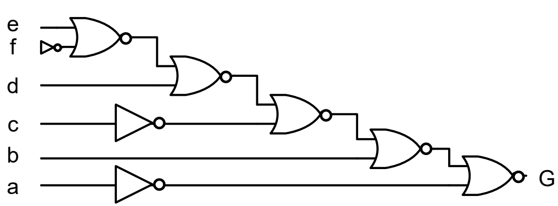
- Convert the circuits shown below to use only NAND or NOR gates?

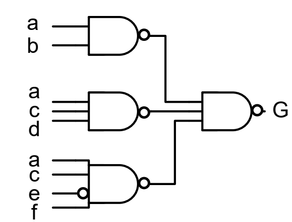






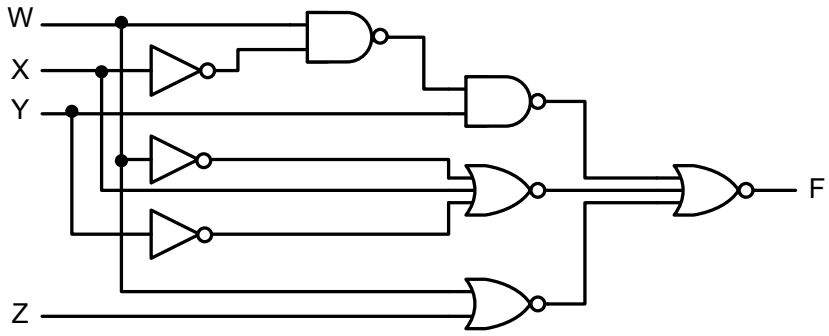


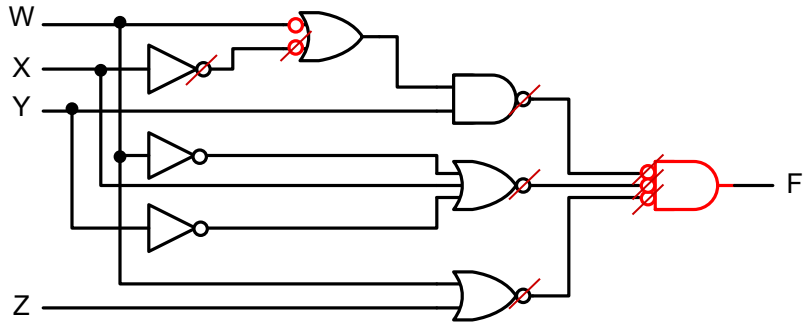




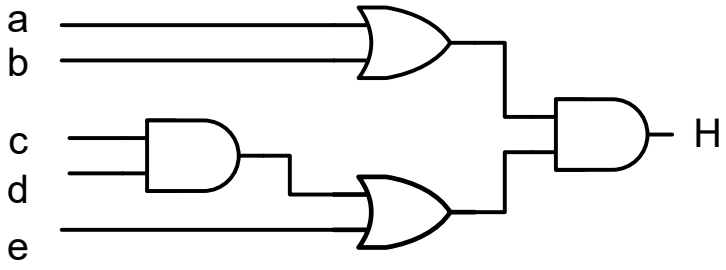
DeMorgan's Theorem Example

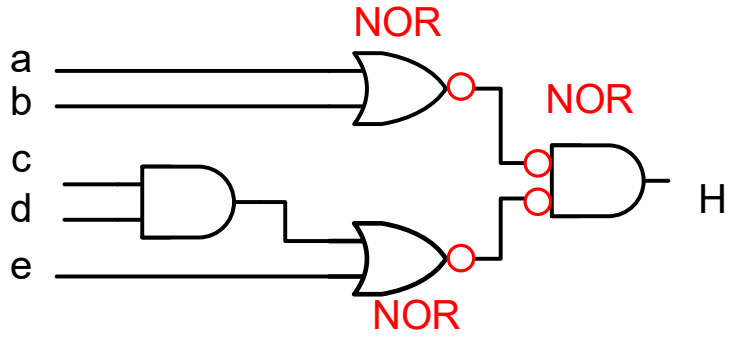
- Cancel as many bubbles as you can using DeMorgan's theorem.





- Convert as many gates as possible to NOR gates. You are allowed to add additional inverters





DeMorgan's Theorem

- DeMorgan's let's us break large inversions (of whole expressions) into smaller inversions (of individual literals).
  - This is necessary to arrive at SOP or POS (which can only have inversions of literals)
- Recursively find the last (lowest precedence operation) and apply DeMorgan's theorem by flipping the operation and inverting the inputs

$$F = \overline{(\overline{X}+Y) + \overline{Z} \bullet (Y+W)}$$

$$F = \overline{(\overline{X}+Y) + \overline{Z} \bullet (Y+W)}$$

$$F = \overline{(\overline{X}+Y)} \bullet \overline{(\overline{Z} \bullet (Y+W))}$$

$$F = (\overline{\overline{X}} \bullet \overline{Y}) \bullet (\overline{\overline{Z}} + \overline{(Y+W)})$$

$$F = (X \bullet \overline{Y}) \bullet (Z + (\overline{Y} \bullet \overline{W}))$$

Use DeMorgan's theorem to simplify "move" inversions (either to break-up "big bars" or join "small bars"

Generalized DeMorgan's Theorem

$$F'(X_1,\ldots,X_n,+,\bullet) = F(X_1',\ldots,X_n',\bullet,+)$$

To find F', swap AND's and OR's and complement each literal. However, you must maintain the original order of operations.

Note: This parentheses doesn't matter (we are just OR'ing X', Y, and the following subexpression)

$$F = (\overline{X}+Y) + \overline{Z} \bullet (Y+W)$$

$$F = (\overline{X}+Y) + \overline{Z} \bullet (Y+W)$$

$$\overline{F} = \overline{(\overline{X}+Y) + \overline{Z} \bullet (Y+W)}$$

$$\overline{F} = \overline{X}+Y + (\overline{Z} \bullet (Y+W))$$

Fully parenthesized to show original order of ops.

$$\overline{F} = X \bullet \overline{Y} \bullet Z + (\overline{Y} \bullet \overline{W})$$

$$\overline{F} = X \bullet \overline{Y} \bullet (Z + (\overline{Y} \bullet \overline{W}))$$

AND's & OR's swapped
Each literal is inverted

Generalized DeMorgan's Theorem

$$F'(X_1,\ldots,X_n,+,\bullet) = F(X_1',\ldots,X_n',\bullet,+)$$

To find F', swap AND's and OR's and complement each literal. However, you must maintain the original order of operations.

Note: This parentheses doesn't matter (we are just OR'ing X', Y, and the following subexpression)

$$F = (\overline{X}+Y) + \overline{Z} \bullet (Y+W)$$

$$F = \overline{X}+Y + (\overline{Z} \bullet (Y+W))$$

Fully parenthesized to show original order of ops.

$$\overline{F} = X \bullet \overline{Y} \bullet (Z + (\overline{Y} \bullet \overline{W}))$$

AND's & OR's swapped
Each literal is inverted

Additional Content

Not Tested

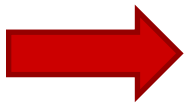
Canonical Sums and Products

LOGIC FUNCTION NOTATION

Canonical Sums and Products

- Truth tables require us to list all 2^n combinations of the n inputs
- A shorthand for a truth table is to describe the function using the canonical sum (sigma, Σ) or product (pi, Π) notation
- These forms of expressing a function have all the information in the truth table but can be written more compactly
  - Though still may require listing 2^n input values
- We'll often use these shorthand notations in assignments/exams

	X	Y	Z	F
m_0	0	0	0	0
m_1	0	0	1	0
m_2	0	1	0	1
m_3	0	1	1	1
m_4	1	0	0	0
m_5	1	0	1	1
m_6	1	1	0	0
m_7	1	1	1	1



$$F = \Sigma_{xyz}(2,3,5,7)$$

Canonical Sums

- Given a T.T., use the minterms where F=1 and SUM them together
  - (Σ = SUM or OR of all the minterms)

	X	Y	Z	F
m_0	0	0	0	0
m_1	0	0	1	0
m_2	0	1	0	1
m_3	0	1	1	1
m_4	1	0	0	0
m_5	1	0	1	1
m_6	1	1	0	0
m_7	1	1	1	1

$$F = m_2 + m_3 + m_5 + m_7$$
$$= (X'YZ') + (X'YZ) + (XY'Z) + (XYZ)$$

Canonical Sum:

$$F = \Sigma_{xyz}(2,3,5,7)$$

List the variables in the order they would appear in the truth table and that you'd use to find the decimal values

List the minterms where F is 1, and just list their decimal number equivalent

Canonical Products

- Given a T.T., AND together all the maxterms where F = 0

	X	Y	Z	F
M_0	0	0	0	0
M_1	0	0	1	0
M_2	0	1	0	1
M_3	0	1	1	1
M_4	1	0	0	0
M_5	1	0	1	1
M_6	1	1	0	0
M_7	1	1	1	1

$$F = M_0 \bullet M_1 \bullet M_4 \bullet M_6$$
$$= (X+Y+Z) \bullet (X+Y+Z') \bullet (X'+Y+Z) \bullet (X'+Y'+Z)$$

Canonical Product:

$$F = \Pi_{xyz}(0,1,4,6)$$

List the variables in the order they would appear in the truth table and that you'd use to find the decimal values

List the maxterms where F is 0, and just list their decimal number equivalent

Canonical Sums & Products

- **Canonical Sum**: An SOP expression where all the product terms are minterms (i.e. have each literal in each product term)
- **Canonical Product**: A POS expression where all the sum terms are maxterms (i.e. each literal in each sum term)

Canonical Sum:

$F = \Sigma_{xyz}(2,3,5,7)$

$F = m_2 + m_3 + m_5 + m_7$

$= (X'YZ') + (X'YZ) + (XY'Z) + (XYZ)$

Canonical Product:

$F = \Pi_{xyz}(0,1,4,6)$

$F = M_0 \bullet M_1 \bullet M_4 \bullet M_6 =$

$(X+Y+Z) \bullet (X+Y+Z') \bullet (X'+Y+Z) \bullet (X'+Y'+Z)$

X	Y	Z	F
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Definitions

- **Minterm**: A **product** term where all the input variables of a function appear as exactly one literal

f(x,y,z)	Yes (m_i) / No	g(w,x,y,z)	Yes (m_i) / No
x•y	No	w'•x•y'•z'	Yes, m4
x'•y•z'	Yes, m2	w•x'•z'	No
x+y'•z'	No (not prod. term)	w•x'•y•z	Yes, m11
(x•y'•z)'	No (not prod. term)	w'•x•y•z'	Yes, m6

- **Maxterm**: A **sum** term where each input variable of a function appears as exactly one literal

f(a,b,c)	Yes (M_i) / No	g(v,w,x,y,z)	Yes (M_i) / No
a+b'+c	Yes, M2	v+w'+x'+y+z	Yes, M12
a+b'	No	v+z	No
a'•(b'+c)	No (not sum term)	v'+w+x+y'+z'	Yes, M19
(a'+b+c)'	No (not sum term)		

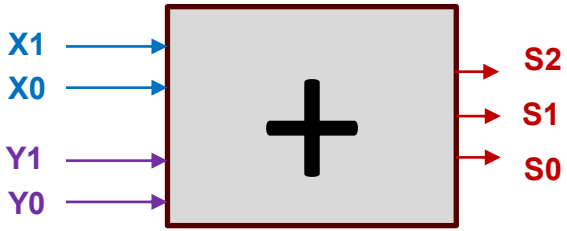
BACKUP

Key: Truth tables

- **Important skill**: Describe an operation or problem statement with a truth table.
- Steps:
  - Identify the inputs and how many bits each requires
  - Determine how many output bits are needed by considering the numeric range of all possible outputs
  - Write a truth table with all 2^n combinations of the n input bits and come up with the desired input on your own
  - Convert each output to a logic circuit using one of the synthesis methods we will teach you in the next two units.

Task

Build an addition operation for 2-bit unsigned binary numbers: X1X0 + Y1Y0



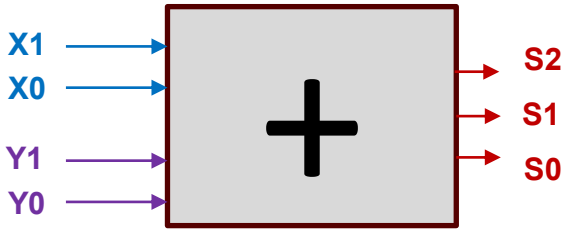
X1	X0	Y1	Y0	S2	S1	S0	Descrip
0	0	0	0	0	0	0	0+0=0
0	0	0	1	0	0	1	0+1=1
0	0	1	0	0	1	0	0+2=2
0	0	1	1	0	1	1	0+3=3
0	1	0	0	0	0	1	1+0=1
0	1	0	1	0	1	0	1+1=2
0	1	1	0	0	1	1	1+2=3
0	1	1	1	1	0	0	1+3=4
1	0	0	0	0	1	0	2+0=2
1	0	0	1	0	1	1	2+1=3
1	0	1	0	1	0	0	2+2=4
1	0	1	1	1	0	1	2+3=5
1	1	0	0	0	1	1	3+0=3
1	1	0	1	1	0	0	3+1=4
1	1	1	0	1	0	1	3+2=5
1	1	1	1	1	1	0	3+3=6

Key: Truth tables

- **Important skill**: Describe an operation or problem statement with a truth table.
- Steps:
  - Identify the inputs and how many bits each requires
  - Determine how many output bits are needed by considering the numeric range of all possible outputs
  - Write a truth table with all 2^n combinations of the n input bits and come up with the desired input on your own
  - Convert each output to a logic circuit using one of the synthesis methods we will teach you in the next two units.

Task

Build an addition operation for 2-bit unsigned binary numbers: X1X0 + Y1Y0



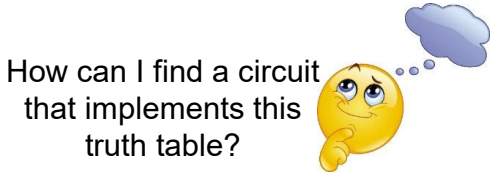
X1	X0	Y1	Y0	S2	S1	S0
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	0	0	1	0
0	0	1	1	0	1	1
.	.	.	.	.	.	.
1	1	0	1	1	0	0
1	1	1	0	1	0	1
1	1	1	1	1	1	0

S	X	Y	Z
0	0	0	**0**
0	0	1	**1**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**0**
1	1	0	**1**
1	1	1	**1**

X	Y	F
0	0	**1**
0	1	**0**
1	0	**1**
1	1	**1**

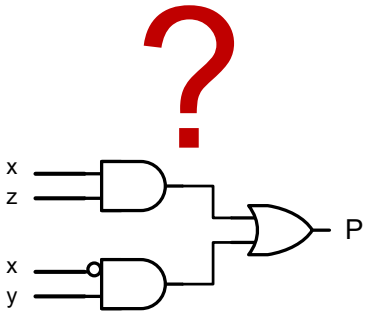
The Problem

- The goal of this unit is to teach you how you can take ***ANY*** logic function expressed as a truth table and design a digital circuit to implement that logic function



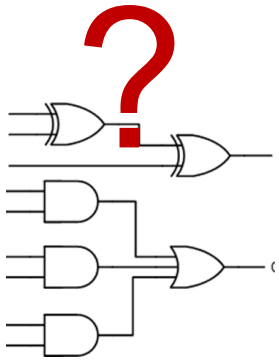
Primes between 0-7

X	Y	Z	P
0	0	0	**0**
0	0	1	**0**
0	1	0	**1**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**0**
1	1	1	**1**

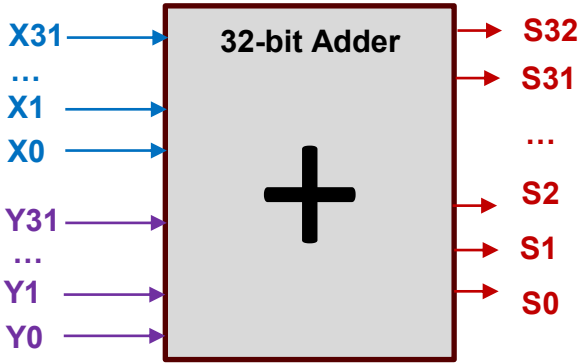


1's Count (Addition) of Inputs

I3	I2	I1	C1	C0
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**



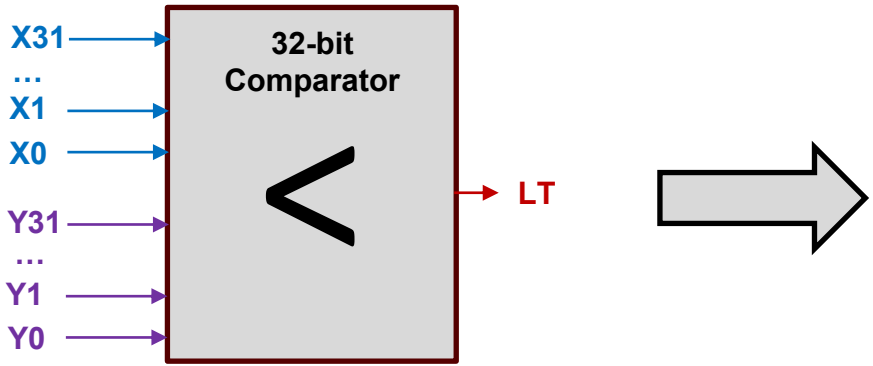
Addition



X31 …	… Y0	S32 …	S0
0	0	0	0
0	1	0	1
0	0	0	0
0	1	0	1
0	0	0	1
0	1	0	0
0	0	0	1
0	1	1	0
1	0	0	0
1	1	0	1
1	0	1	0
1	1	1	1
1	0	0	1
1	1	1	0
1	0	1	1
1	1	1	0

Comparison

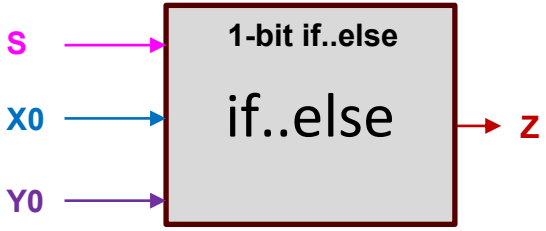
```
bool lt = x < y;
```



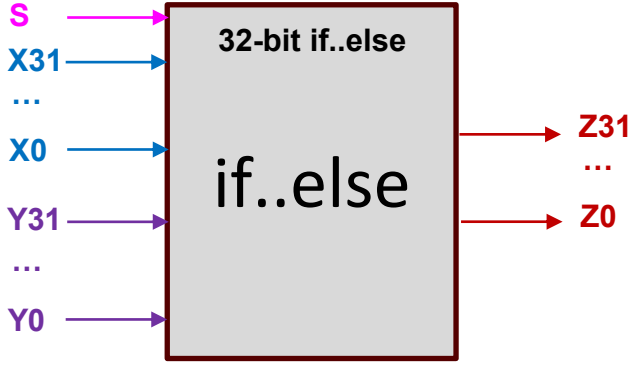
X31	…	…	Y0	LT
0			0	
0			1	
0			0	
0			1	
0			0	
0			1	
0			0	
0			1	
1			0	
1			1	
1			0	
1			1	
1			0	
1			1	
1			0	
1			1	

If..Else (Ternary) Operator (aka "Mux")

```
z = (s==1) ? x : y;

// if(s==1) { z=x; }
// else     { z=y; }
```



S	X	Y	Z
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



?

T8	XY+XZ = X(Y+Z)	**T8'**	(X+Y)(X+Z) = X+YZ	Distribution & Factoring
T9	X + XY = X	**T9'**	X(X+Y) = X	Covering
T10	XY + XY' = X	**T10'**	(X+Y)(X+Y') = X	Combining
T11	XY + X'Z + YZ = XY + X'Z	**T11'**	(X+Y)(X'+Z)(Y+Z) = (X+Y)(X'+Z)	Consensus

T8	AB+AC = A(B+C)	**T8'**	(A+B)(A+C) = A+BC	Distribution & Factoring
T9	A + AB = A	**T9'**	A(A+B) = A	Covering
T10	AB + AB' = A	**T10'**	(A+B)(A+B') = A	Combining
T11	AB + A'C + BC =	**T11'**	(A+B)(A'+C)(B+C) =	Consensus