ECE241
Lec01
2023 (1)

DIGITAL HARDWARE

WHAT IS "HARDWARE"?

- IN EE, IT HAS TWO MEANINGS : ON-CHIP & OFF-CHIP
  - ON-CHIP { is what is integrated; a circuit that is part of a chip and serves a specific purpose, or a circuit that is part of a chip and can be reconfigured to serve different purposes (such as an FPGA)
  - OFF-CHIP are the individual components, not integrated, that can be combined into a circuit to serve a specific purpose

WHAT IS "DIGITAL"?

- EVERY CIRCUIT THAT DEALS WITH DISCRETE VALUES
- IN EE, VALUES CAN BE DISCRETE OR CONTINUOUS
  (A SWITCH) (A DIMMER)

IT IS EASIER TO BUILD CIRCUITS THAT DISTINGUISH TWO VALUES ⇒ HIGH/LOW, ON/OFF
ZERO/ONE

DISCRETE, BINARY

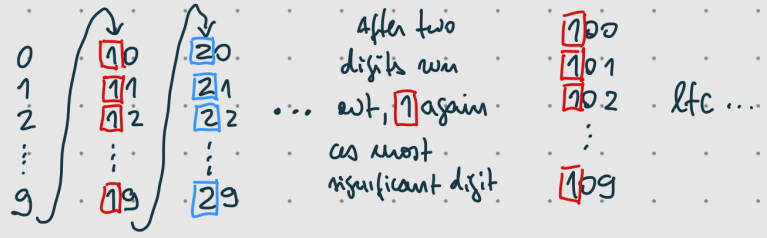
WE MUST FIND A WAY TO REPRESENT VALUES (NUMBERS) IN DIGITAL, BINARY FORMAT

NUMBER REPRESENTATIONS & CONVERSION

WE NEED TO REPRESENT WHAT MAKES SENSE TO US, WITH A FORMAT THAT A MACHINE (A CIRCUIT) CAN PROCESS. LET US START BY LOOKING AT DECIMAL & BINARY NUMBERS.

A DECIMAL NUMBER HAS 10 UNIQUE VALUES 0–9 ("BASE 10")

AFTER SINGLE DIGIT VALUES RUN OUT, DOUBLE DIGITS START
DOUBLE DIGITS START WITH 1 AS THE MOST SIGNIFICANT DIGIT, THEN 2, etc.

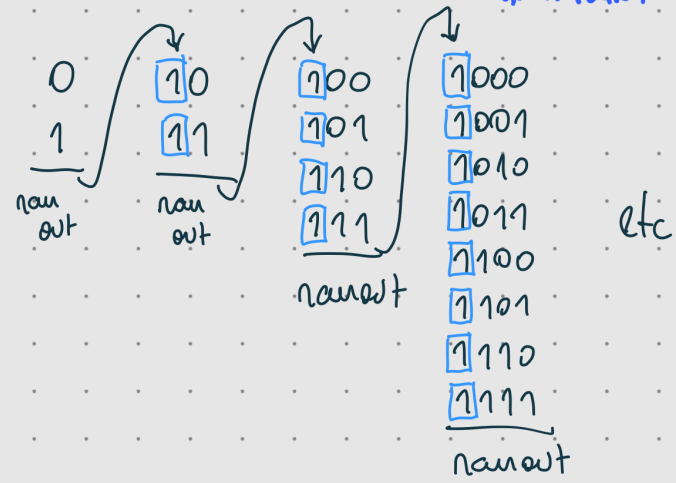


IN BINARY, THERE ARE ONLY TWO UNIQUE VALUES: 0–1. ("BASE 2")

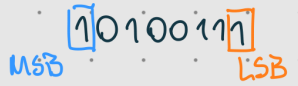
AFTER SINGLE DIGIT VALUES RUN OUT, DOUBLE DIGITS START
DOUBLE DIGITS START WITH 1 AS THE MOST SIGNIFICANT DIGIT

IN BINARY, THE DIGIT IS REFERRED TO AS BIT ("BINARY DIGIT")

WE REFER TO THE LEFT-MOST BIT AS **MOST SIGNIFICANT BIT (MSB)**



WE ALSO REFER TO THE RIGHT-MOST BIT AS THE LEAST SIGNIFICANT BIT. (LSB)



- INTEGER DECIMALS REPRESENT A SUM OF POWERS OF 10

Ex: $1967 = 1\times10^3 + 9\times10^2 + 6\times10^1 + 7\times10^0$

- LIKEWISE, INTEGER BINARY NUMBERS REPRESENT A SUM OF POWERS OF 2

Ex: $1101 = 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

$= 8 + 4 + 0 + 1 = 13$

BINARY IS THE LOWEST LEVEL FORMAT THAT MACHINES UNDERSTAND

LETTERS, ADDRESSES, NUMBERS ARE ALL REPRESENTED IN THIS FORMAT

In ECE241 we will start with integer numbers and their representation in binary. So we need to find a way to convert between formats

⊖ REPRESENTING AND CONVERTING UNSIGNED INTEGERS

THE REPRESENTATION OF DECIMAL INTEGERS IN BINARY IS DONE THROUGH A DIRECT MAPPING BETWEEN THE TWO BASES

BASE 10 DECIMAL	BASE 2 BINARY
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7_d	111_b

Notice a few things:

a) from 0_d to 7_d we have 8 unique values.

We need a minimum of 3 bits to represent all 8 values

$8 \longleftrightarrow 2^{\boxed{3}}$ (# bits)

8: # unique decimal values; 2: numerical base (2)

Q: How many unique integer decimals can one represent with 64 bits?

2^{64}

From an example given above, it follows that the number 111 (base 2) can be interpreted as a decimal (base 10) by an addition of powers of 2

$1\times2^2 + 1\times2^1 + 1\times2^0 = 4 + 2 + 1 = 7$

(111 binary)

Let's formalize it

bit $n-1$ ... bit 1, bit 0

Binary number, n-bits: $B_n = b_{n-1}\ b_{n-2} \cdots b_1\ b_0$

Decimal unsigned integer: $D_n = b_{n-1}\cdot2^{n-1} + b_{n-2}\cdot2^{n-2} + \cdots + b_1\cdot2^1 + b_0\cdot2^0$

Ex: $1101 = 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$

(bit 3, bit 0)

$= 8 + 4 + 0 + 1 = 13$ (base 10 unsigned integer)

If we now have a larger decimal number to convert into binary, we divide successively by 2 & record the remainder

Decimal: $B = (b_{n-1}2^{n-1}) + (b_{n-2}2^{n-2}) + \cdots + (b_1 2^1) + (b_0 2^0)$

MSB ... LSB

$\frac{B}{2} = (b_{n-1}2^{n-2}) + (b_{n-2}2^{n-3}) + \cdots + (b_1 2^0)$ — remainder is b_0

$\frac{B}{4} = (b_{n-1}2^{n-3}) + (b_{n-2}2^{n-4}) + \cdots + (b_2 2^0)$ — remainder is b_1

Let's try it

13 | 2 → 1; 6 | 2 → 0; 3 | 2 → 1; 1 | 2 → 1; 0

LSB ... MSB

$\Rightarrow 1101_b = 13_d$

ECE241 (5) 2023

⊕ A THIRD NUMERICAL REPRESENTATION IS ALSO COMMONLY FOUND IN DIGITAL SYSTEMS: HEXADECIMAL (BASE 16)

- IT HAS 16 UNIQUE SYMBOLS: 0 — 9, A — f (10, 6)

IT IS VERY COMMON TO USE HEX WHEN THE MACHINE ARCHITECTURE MAKES USE OF WIDE, MULTIBIT WORDS → 32, 64 bit

- IT PROVIDES A SHORTER REPRESENTATION

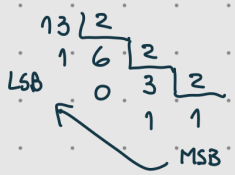
IN BASE 16 WE HAVE 16 UNIQUE SYMBOLS, SO WE NEED 4 BITS TO REPRESENT EACH

0	0000	10	0001 0000		f0	1111 0000
1	0001	11	0001 0001		f1	1111 0001
⋮	⋮	⋮	⋮		⋮	⋮
9	1001	19	0001 1001		f9	1111 1001
A	1010	1A	0001 1010		fA	1111 1010
B	1011	1B	0001 1011	...	fB	1111 1011
C	1100	1C	0001 1100		fC	1111 1100
D	1101	1D	0001 1101		fD	1111 1101
E	1110	1E	0001 1110		fE	1111 1110
F	1111	1F	0001 1111		ff	1111 1111
			1 F			

TWO SYMBOLS REPRESENTING 8 bits

⊖ CONVERT BETWEEN BASES

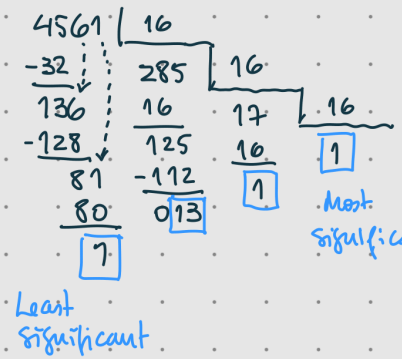
$D_{HEX} = 13_{DEC.}$

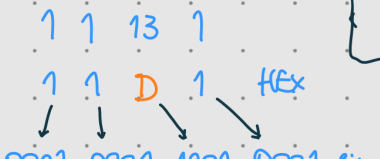


$\Rightarrow D_{HEX} = 13_{DEC} = 1101_{BIN}$

ECE241 (6) 2023

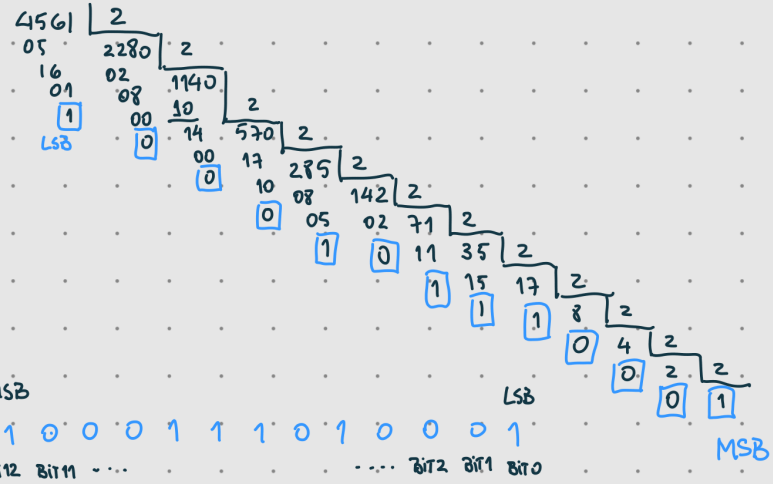
4561 DEC TO HEX



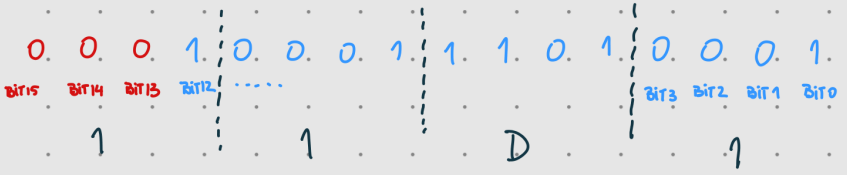


16 BIT representation of 4561 decimal

ONE CAN CONFIRM THE BINARY REPRESENTATION OF 4561 BY CONVERTING DIRECTLY



NOTE THE POSITION OF THE BITS; IF WE ARE TO REPRESENT THIS NUMBER USING 16 BITS, WE HAVE



ECE241
(7)
2023

IN DECIMAL (BASE 10), WHEN WE MULTIPLY BY 10, WE SHIFT THE NUMBER TO THE LEFT & ADD A ZERO.

$$35 \times 10 = 350$$

LIKEWISE, WHEN WE MULTIPLY AN UNSIGNED, INTEGER BINARY NUMBER BY 2, WE SHIFT THE BITS TO THE LEFT BY ONE PLACE & ADD A ZERO

0 0 0 1 | 0 0 0 1 | 1 1 0 1 | 0 0 0 1 = 11D1 (4561)

0 0 1 0 | 0 0 1 1 | 1 0 1 0 | 0 0 1 0

2 3 A 2 (9122)

SIMILARLY, WHEN WE DIVIDE A BASE 10 NUMBER BY 10, WE SHIFT THE NUMBER RIGHT.

TO DIVIDE AN UNSIGNED, INTEGER BINARY NUMBER BY 2, WE SHIFT THE BITS TO THE RIGHT BY ONE PLACE

0 0 0 1 | 0 0 0 1 | 1 1 0 1 | 0 0 0 1 = 11D1 (4561)

0 0 0 0 | 1 0 0 0 | 1 1 1 0 | 1 0 0 0

0 8 E 8 (2280)

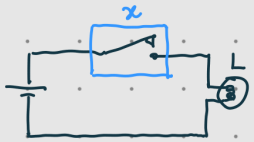
NOTE HERE THAT SOME PRECISION IS LOST! (CAN YOU AFFORD THAT?)

MODULE 2 - LOGIC CIRWITS

ECE241 (1) 2023

SAY WE HAVE A SWITCH x, SUCH THAT WHEN x is OPEN $x=0$ AND WHEN x is CLOSED, $x=1$

x IS OUR INPUT VARIABLE, THAT WHEN CHANGED WILL PRODUCE/CAUSE CHANGES IN OUTPUT



OUTPUT $L = 1$ (LIGHT ON)
OUTPUT $L = 0$ (LIGHT OFF)

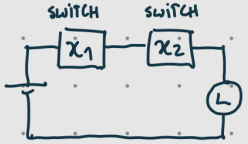
WE CAN WRITE $L = x$, TO SAY WHEN SWITCH IS OPEN ($x=0$), LIGHT IS OFF ($L=0$) AND WHEN SWITCH IS CLOSED ($x=1$), LIGHT IS ON ($L=1$)

THIS IS A LOGIC EXPRESSION, WHERE L IS A LOGIC FUNCTION OF x, THE BOOLEAN VARIABLE

- A BOOLEAN VARIABLE CAN ONLY HAVE THE VALUES 0 & 1

WE CAN NOW CREATE MORE COMPLEX EXPRESSIONS, WITH MULTIPLE VARIABLES

Ex:

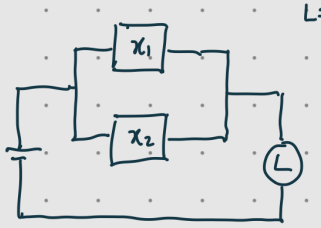


L WILL BE ON WHEN BOTH x_1 AND x_2 ARE CLOSED

$L=1$ IFF $x_1=1$ AND $x_2=1$

WE WRITE
- $L = x_1$ AND x_2
- $L = x_1 \cdot x_2$
- $L = x_1 \,\&\, x_2$
- $L = x_1 x_2$

LIKEWISE

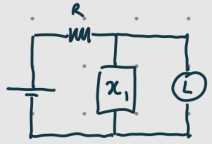


$L=1$ WHEN EITHER x_1 OR x_2 ARE CLOSED

- $L = x_1$ OR x_2
- $L = x_1 + x_2$
- $L = x_1 \mid x_2$

ECE241 (2) 2023

ANOTHER POSSIBILITY



(QUESTION: WHY THE RESISTOR?)

$L = 0$ if $x = 1$
$L = 1$ if $x = 0$

$L =$ NOT x
$L = \sim x$ $\quad L = !x$ $\quad L = x'$ (x PRIME)
$L = \overline{x}$ (x BAR)

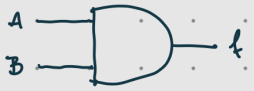
MORE COMPLEX OPERATIONS CAN BE EXPRESSED WITH THIS NOTATION

ex

$$f(x_1, x_2) = x_1 + x_2 \longrightarrow \overline{f}(x_1, x_2) = \overline{x_1 + x_2}$$

- DRAWING CIRCUITS CAN GET COMPLICATED, SO WE'LL USE SYMBOLS TO REPRESENT THE CORRESPONDING CIRCUIT

AND



$f = A \cdot B$

OR

$f = A + B$

NOT

$f = \overline{A}$

AS THE CIRCUITS AND EXPRESSIONS GROW MORE COMPLEX, A TRUTH TABLE IS THE TOOL THAT WILL PROVIDE A VISUAL REPRESENTATION OF HOW THE INPUTS AND OUTPUTS RELATE

INPUT		AND	OR
x_1	x_2	$x_1 \cdot x_2$	$x_1 + x_2$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

NOTE: ALL VALUES & COMBINATIONS OF THE INPUTS ARE LISTED AND ANALYZED

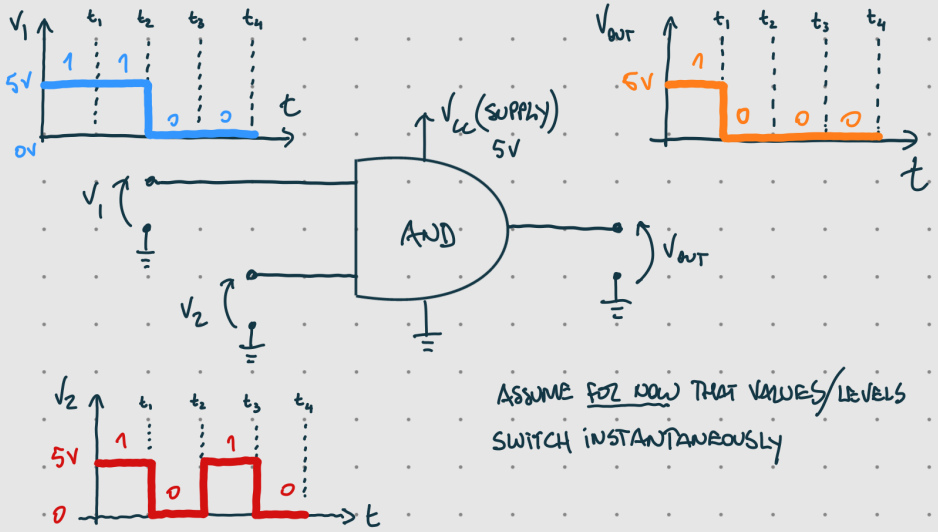
Look @ channel "IN ONE LESSON". YouTube - "HOW TRANSISTORS DO MATH"

AT THIS POINT, LET'S LOOK AT WHAT THIS ALL MEANS ELECTRICALLY

- THERE ARE TWO TYPES OF ELECTRONIC DEVICES: ACTIVE AND PASSIVE
  - ACTIVE DEVICES NEED TO "BE FED", THAT IS, THEY NEED A POWER SUPPLY TO FUNCTION
    - TRANSISTORS ARE ACTIVE DEVICES, SO EVERY DEVICE COMPRISED OF TRANSISTORS IS ALSO ACTIVE; THEY NEED A POWER SUPPLY TO FUNCTION
      - EX: LOGIC GATES, OPERATIONAL AMPLIFIERS, MICROCONTROLLERS, MICROPROCESSORS, FPGAs
  - PASSIVE DEVICES DO NOT NEED A POWER SUPPLY TO MAKE THEM WORK.
    - EX: RESISTORS, CAPACITORS, INDUCTORS, DIODES

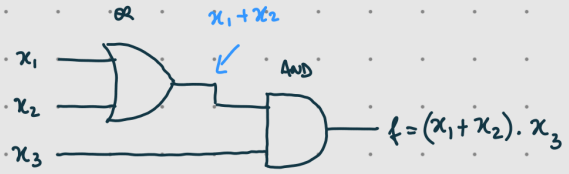
- LOGIC GATES NEED A VOLTAGE SOURCE TO WORK. (ACTIVE). DEPENDING ON THE TECHNOLOGY USED, SUPPLY WILL BE 5V or 3.3V
  - THIS VOLTAGE DETERMINES THE LEVELS AT WHICH THE LOGIC GATES WILL INTERPRET WHAT IS "HIGH" (1) or "LOW" (0)

- CONSIDER THE CIRCUIT, WHICH USES A 5V POWER SUPPLY



ASSUME FOR NOW THAT VALUES/LEVELS SWITCH INSTANTANEOUSLY

WE TYPICALLY ANALYZE A GIVEN LOGIC CIRCUIT USING ONLY THE LOGIC LEVEL REPRESENTATION, AND BUILDING A TRUTH TABLE



x_1	x_2	x_3	f
0	0	0	0
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

ALL POSSIBLE INPUT VALUES ARE LISTED ON THE TRUTH TABLE, AS WELL AS ALL POSSIBLE OUTPUTS PRODUCED

NOTE - f WILL BE 1 ONLY WHEN x_3 IS 1 AND x_1 OR x_2 ARE 1
- IF x_3 IS ZERO THE OUTPUT WILL BE ZERO; IT DOES NOT MATTER WHAT x_1 & x_2 ARE

[MORE COMPLEX EXAMPLE HERE]

LET US ANALYZE A 1-BIT ADDER, A.K.A HALF ADDER (A CIRCUIT THAT ADDS 2 BITS)

HOW ARE TWO BITS ADDED?

$$\begin{array}{r} 0 \\ +\ 0 \\ \hline 0 \end{array} \qquad \begin{array}{r} 0 \\ +\ 1 \\ \hline 1 \end{array} \qquad \begin{array}{r} 1 \\ +\ 0 \\ \hline 1 \end{array} \qquad \begin{array}{r} {\scriptstyle 1}\ 1 \\ +\ 1 \\ \hline 10 \end{array}$$

"ZERO, CARRY ONE"

WE CAN SAY

$$\begin{array}{r} \text{BIT } a \\ +\ \text{BIT } b \\ \hline S_1\ S_0 \end{array}$$

⇒ TWO INPUTS, TWO OUTPUTS

CARRY

WHEN a AS WELL AS b are 1
$S_0 = 0$ and $S_1 = 1$

LET US DESCRIBE THIS BEHAVIOUR WITH A TRUTH TABLE

HALF ADDER

inputs		functions	
a	b	S_1	S_0
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$S_1 = 1$ ONLY WHEN $a=1$ AND $b=1$

$$S_1 = ab$$

OR: $S_1(a,b) = ab$

$S_1(0,1) = 0.1 = 0$

$S_1(1,1) = 1.1 = 1$

SIMILARLY, $S_0 = 1$ WHEN $a=0$ AND $b=1$ OR $a=1$ AND $b=0$

THEREFORE $S_0 = \bar{a}b + a\bar{b}$ OR $S_0(a,b) = \bar{a}b + a\bar{b}$

LET US TEST IF THIS WORKS:

$S_1(0,1) = 0.1 = 0$

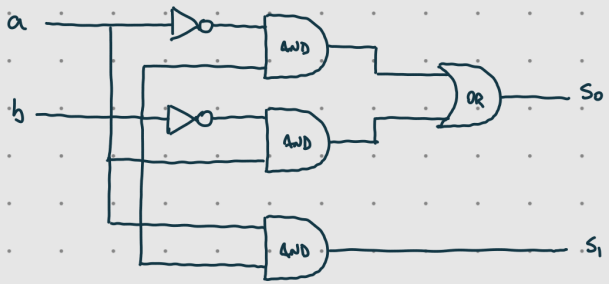
$S_0(0,1) = \bar{0}.1 + 0.\bar{1} = 1.1 + 0.0 = 1$

TRUTH TABLE ✓

$S_1(1,1) = 1.1 = 1$

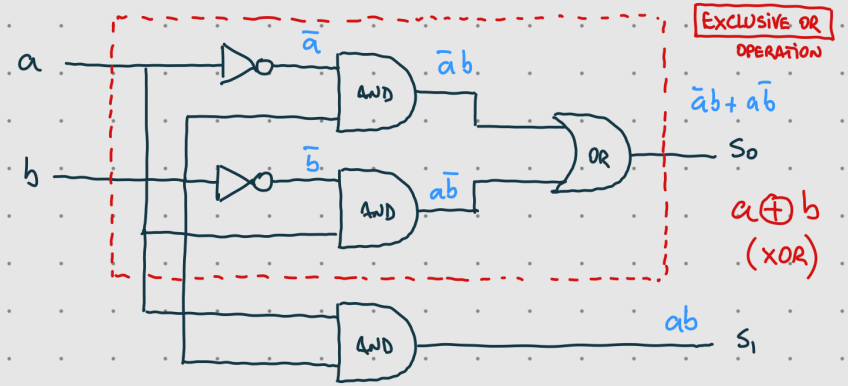
$S_0(1,1) = \bar{1}.1 + 1.\bar{1} = 0.1 + 1.0 = 0$ ✓

WITH THE TRUTH TABLE AND THE TWO DERIVED EQUATIONS, WE CAN BUILD A CIRCUIT TO IMPLEMENT THAT.

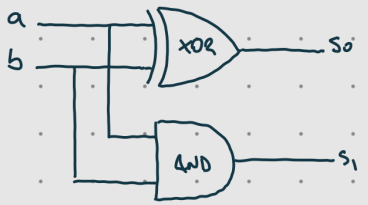


NOTICE FROM THE CIRCUIT THAT
- WHEN a & b ARE DIFFERENT THE OUTPUT IS HIGH
- WHEN a & b ARE THE SAME THE OUTPUT IS LOW

THIS IS AN EXCLUSIVE OR OPERATION



WE CAN REDRAW THE CIRCUIT



THE XOR GATE FUNCTIONS SIMILARLY TO UPSTAIRS/DOWNSTAIRS LIGHT SWITCHES CONTROLLING THE SAME LIGHT.

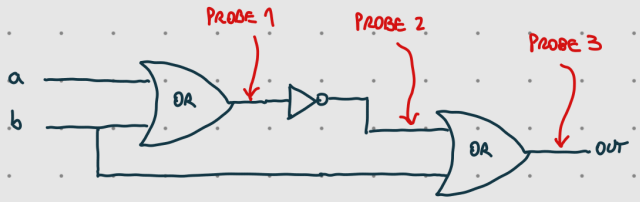
ANOTHER VERY COMMONLY USED GATE IS THE NAND GATE (AN AND OPERATION PLUS A NOT OPERATION)

WE WILL SEE NEXT THAT THE NAND GATE CAN BE COMBINED WITH OTHER NAND GATES TO BUILD ANY OTHER LOGIC GATES

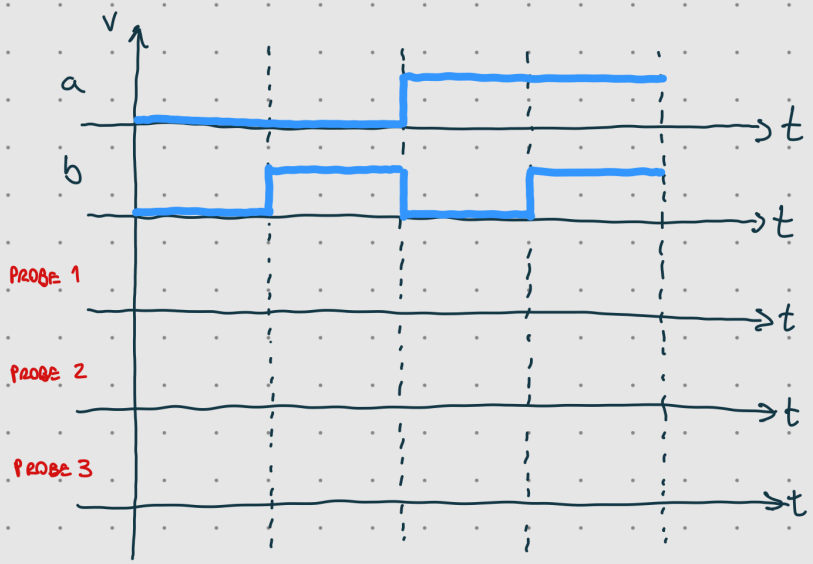
EXERCISE: BUILD AN XOR GATE WITH NAND GATES

ECE241
(7)
2023

EXERCISE: GIVEN THE FOLLOWING CIRCUIT, COMPLETE THE TRUTH TABLE AND THE WAVEFORMS ON THE TIMING DIAGRAM



a	b	PROBE 1	PROBE 2	PROBE 3
0	0			
0	1			
1	0			
1	1			



③ BOOLEAN ALGEBRA & VENN DIAGRAMS

- WE WILL APPLY THE ALGEBRA CREATED BY GEORGE BOOLE TO DESIGN & ANALYZE LOGIC CIRCUITS
  - LOOK UP: "THE LAWS OF THOUGHT" by GEORGE BOOLE

- WE SAW THE OPERATIONS AND, OR, NOT, XOR AS THEY OPERATED ON BINARY VARIABLES
  - WE WILL CALL THESE BOOLEAN VARIABLES NOW
  - TWO VALUES ONLY: 0 – 1, FALSE — TRUE, LOW — HIGH
- IN BOOLEAN ALGEBRA, AS IN EVERY ALGEBRA, THERE ARE BASIC ASSUMPTIONS (AXIOMS) AND RULES THAT MUST BE FOLLOWED.

DUALITY: GIVEN A LOGIC EXPRESSION, IF WE SWAP $1 \leftrightarrow 0$ AND ALSO $\cdot \leftrightarrow +$ THE EXPRESSION STILL WORKS

⊕ AXIOMS

①	$0 \cdot 0 = 0$	$1 + 1 = 1$
②	$1 \cdot 1 = 1$	$0 + 0 = 0$
③	$0 \cdot 1 = 1 \cdot 0 = 0$	$1 + 0 = 0 + 1 = 1$
④	if $x = 0$, $\bar{x} = 1$	if $x = 1$, $\bar{x} = 0$

RECALL A BOOLEAN VARIABLE CAN ONLY TAKE VALUES 0 OR 1

⊕ RULES

⑤	$x \cdot 0 = 0$	$x + 1 = 1$
⑥	$x \cdot 1 = x$	$x + 0 = x$
⑦	$x \cdot x = x$	$x + x = x$
⑧	$x \cdot \bar{x} = 0$	$x + \bar{x} = 1$
⑨	$\bar{\bar{x}} = x$	

AND = LOGICAL PRODUCT

OR = LOGICAL SUM

NOTICE THERE IS MUCH SIMILARITY BETWEEN BOOLEAN OPERATIONS AND ARITHMETIC OPERATIONS THE SAME SYMBOLS ARE BEING USED

$\cdot \equiv$ AND

$+ \equiv$ OR

- BASED ON THE AXIOMS & RULES, WE HAVE IDENTITIES & PROPERTIES

COMMUTATIVE

⑩ $x \cdot y = y \cdot x$ $\qquad x + y = y + x$

ASSOCIATIVE

⑪ $x \cdot (y \cdot z) = (x \cdot y) \cdot z$ $\qquad x + (y + z) = (x + y) + z$

DISTRIBUTIVE

⑫ $x \cdot (y + z) = x \cdot y + x \cdot z$ $\qquad x + (y \cdot z) = (x + y) \cdot (x + z)$

x	y	z	$(y \cdot z)$	LHS	$x + y$	$x + z$	RHS
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

TRY EXAMPLE 2.4 PP 36.–37

REDUCE # OF LOGIC GATES THROUGH ALGEBRAIC MANIPULATION

⑬ $x + x \cdot y = x$ $\qquad x \cdot (x + y) = x$ COVERING

⑭ $x \cdot y + x \cdot \bar{y} = x$ $\qquad (x + y)(x + \bar{y}) = x$ COMBINING

$x \cdot (y + \bar{y})$

⑮ $\overline{x \cdot y} = \bar{x} + \bar{y}$ $\qquad \overline{x + y} = \bar{x} \cdot \bar{y}$ DE MORGAN'S THEOREM

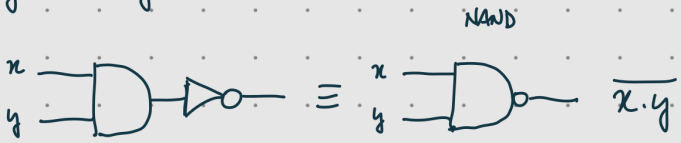
IN BOOLEAN ALGEBRA, THE PRECEDENCE IS NOT, AND, OR

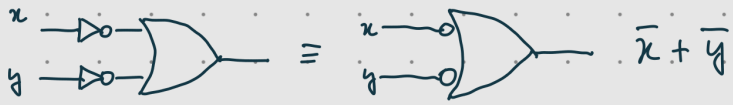
PARENTHESES PRECEDE ALL, BUT BE CAREFUL WITH USING TOO MANY (CONFUSING!)

ECE 241
(3)
2023

A NOTE ON DE MORGAN'S THEOREM

$\overline{x \cdot y} = \bar{x} + \bar{y}$





"BUBBLE PUSH"

BUBBLE GOES FROM INPUT TO OUTPUT, GATE CHANGES FROM OR ↔ AND

DUAL: $\overline{x + y} = \bar{x} \cdot \bar{y}$

NOR GATE

WE WILL SHOW LATER THAT ANY LOGIC GATE CAN BE MADE OF NAND GATES

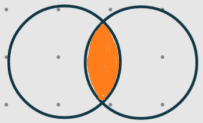
NOW LET'S SEE HOW TO REPRESENT VARIABLES WITH VENN DIAGRAMS

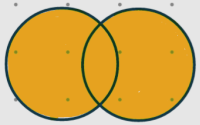
ECE 241
(4)
2023

VENN DIAGRAMS

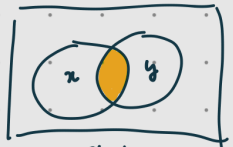
WE USE THEM TO REPRESENT BOOLEAN VARIABLES AND OPERATIONS

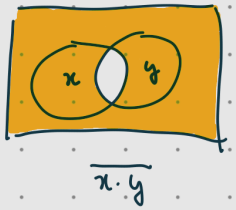
x is INSIDE

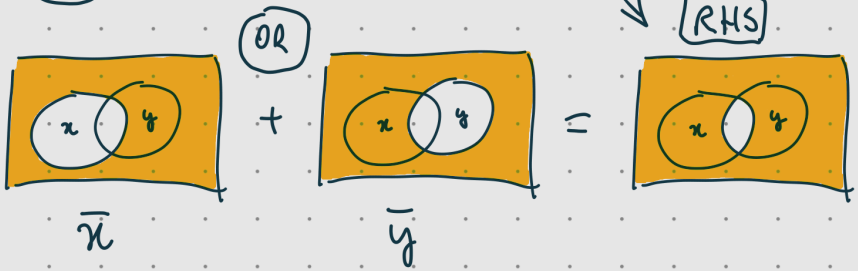




LET'S REVISIT DE MORGAN'S THEOREM: $\overline{x \cdot y} = \bar{x} + \bar{y}$







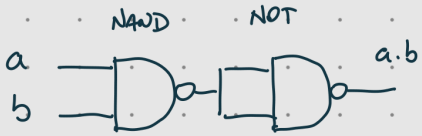
ECE241 ⑤ 2023

⊕ USING NAND GATES TO BUILD OTHER GATES

NOT $a \cdot a = a$

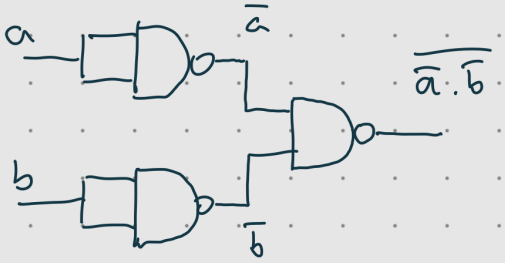
$\overline{a \cdot a} = \bar{a}$

AND ≡ "NAND NOT"



OR DE MORGAN

$\overline{\bar{a} \cdot \bar{b}} = \bar{\bar{a}} + \bar{\bar{b}} = a + b$

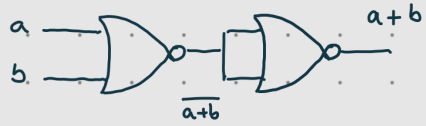


USING NOR GATES TO BUILD OTHER GATES

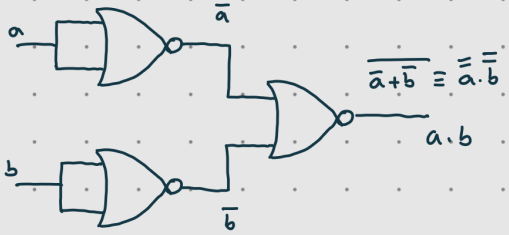
NOT $a + a = a$

$\bar{a} + \bar{a} = \overline{a \cdot a} = \bar{a}$

OR "NOR NOT"



AND DE MORGAN



ECE241 6 2023

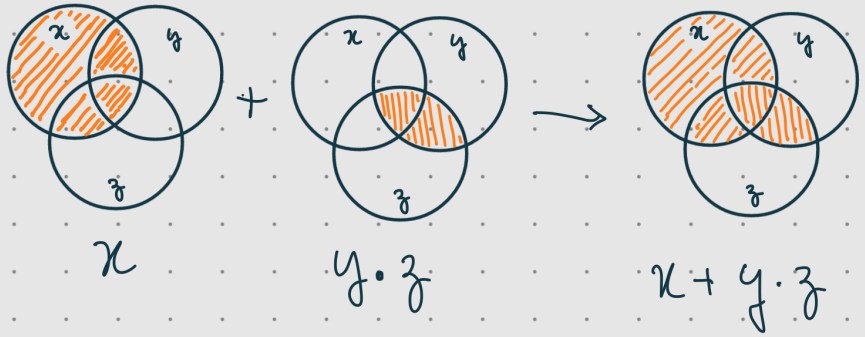
AN EXAMPLE WITH VENN DIAGRAMS

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

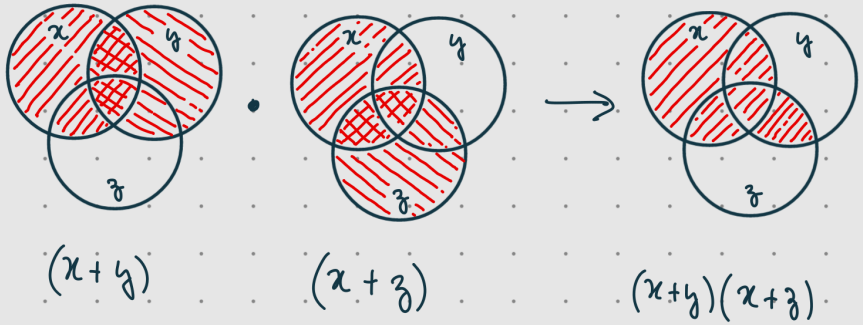
PROVE THIS WITH VENN DIAGRAMS

NOTE THAT THIS IS DIFFERENT THAN THE ALGEBRA WE ARE USED TO.

LHS



RHS



(4) SUM OF PRODUCTS / PRODUCT OF SUMS

ECE241 (1) 2023

LET US DESIGN A SYSTEM WITH WHAT WE KNOW SO FAR

I NEED TO DETECT CARS IN A ROW. IF I DETERMINE I HAVE TWO OR MORE CARS, I WILL SWITCH ON A LEFT TURN LIGHT. I HAVE THREE SENSORS AVAILABLE TO PROVIDE ME WITH THE SIGNALLING TO MAKE MY DECISION

OPERATION: NO LIGHT IF 1 OR OR LESS
LIGHT ON IF 2 OR MORE CARS

BUILD A TRUTH TABLE

S1	S2	S3	LIGHT
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

LIGHT L IS ON WHEN $L=1$

THEREFORE

$$L = \bar{S_1} S_2 S_3 + S_1 \bar{S_2} S_3 + S_1 S_2 \bar{S_3} + S_1 S_2 S_3$$
$$0\,1\,1 \quad 1\,0\,1 \quad 1\,1\,0 \quad 1\,1\,1$$

OR

CAN WE DO BETTER? RECALL THAT $\boxed{x + x = x}$

$$L = \bar{S_1} S_2 S_3 + S_1 \bar{S_2} S_3 + S_1 S_2 \bar{S_3} + S_1 S_2 S_3 + S_1 S_2 S_3 + S_1 S_2 S_3$$

$$L = (\bar{S_1} + S_1) S_2 S_3 + (\bar{S_2} + S_2) S_1 S_3 + (\bar{S_3} + S_3) S_1 S_2$$

THEREFORE $\boxed{L = S_2 S_3 + S_1 S_3 + S_1 S_2}$ WHY IS THIS BETTER?

⊕ SUM OF PRODUCTS & PRODUCT OF SUMS

- FROM THE TRUTH TABLE WE BUILT, WE USED THE ROWS WHERE $\boxed{f(\text{OUTPUT}) = 1}$
- WE COULD HAVE USED $f(\text{output}) = 0$ AS WELL
- NOTE THAT THIS IS EQUIVALENT TO WRITING CODE BASED ON THE TRUTH TABLE USING BOOLEAN VARIABLES S_1, S_2, S_3

```
bool S1, S2, S3;
   ⋮
bool LeftTurn(S1, S2, S3){
   if (!S1 & S2 & S3) return TRUE;
   else if (S1 & !S2 & S3) return TRUE;
      ⋮
   else
      return FALSE;}
```

S_1	S_2	S_3	out
0	1	1	1
1	0	1	1
1	1	0	1
1	1	1	1

ECE241 (2) 2023

BEFORE WE PROCEED TO ANOTHER EXAMPLE, LET US SEE THE APPROPRIATE TERMINOLOGY

- PRODUCT TERM - IS ANY TERM WITH AN AND OPERATION
- MINTERM - IS A PRODUCT TERM THAT INCLUDES ALL INPUTS TO A FUNCTION

SO, $f(\underbrace{x, y, z}_{\text{INPUTS}})$ HAS 8 MINTERMS

x	y	z	MINTERM	
0	0	0	$\bar{x} \cdot \bar{y} \cdot \bar{z}$	m_0
0	0	1	$\bar{x}\ \bar{y}\ z$	m_1
0	1	0	$\bar{x}\ y\ \bar{z}$	m_2
0	1	1	$\bar{x}\ y\ z$	m_3
⋮				
1	1	1	$x\ y\ z$	m_7

FOR OUR PREVIOUS TRUTH TABLE ...

x	y	z	$f(x,y,z)$	
0	0	0	0	m_0
0	0	1	0	m_1
0	1	0	0	m_2
0	1	1	1	m_3
1	0	0	0	m_4
1	0	1	1	m_5
1	1	0	1	m_6
1	1	1	1	m_7

FROM THE TRUTH TABLE, I CAN SAY $f = m_0 \cdot 0 + m_1 \cdot 0 + m_2 \cdot 0 + m_3 \cdot 1 + m_4 \cdot 0 + m_5 \cdot 1 + m_6 \cdot 1 + m_7 \cdot 1$

$$f = \bar{x} y z + x \bar{y} z + x y \bar{z} + x y z$$

$$f(x y z) = \sum m(3, 5, 6, 7)$$

- SUM OF PRODUCTS (SOP) IS A SUM OF PRODUCT TERMS
  ↑ THAT IS OR

- **CANONICAL FORM SOP** – IT IS A UNIQUE EXPRESSION OF A FUNCTION WHERE EACH PRODUCT TERM IS A MINTERM

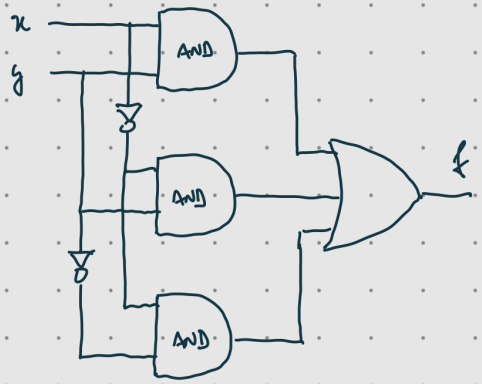
So, $f = \bar{x}yz + x\bar{y}z + xy\bar{z} + xyz$ IS IN CANONICAL FORM

EXAMPLE CONSIDER THIS TRUTH TABLE

x	y	$f(x,y)$	
0	0	1	m_0
0	1	1	m_1
1	0	0	m_2
1	1	1	m_3

$f(x,y) = 1 \cdot m_0 + 1 \cdot m_1 + 0 \cdot m_2 + 1 \cdot m_3$

$f(x,y) = \bar{x}\bar{y} + \bar{x}y + xy$ (CANONICAL → ALL MINTERMS)



THIS CIRCUIT IMPLEMENTS THE FUNCTION (EXPRESSED FROM THE TRUTH TABLE)

CAN WE MAKE THIS "BETTER"?

$f(x,y) = xy + \bar{x}\bar{y} + \bar{x}y$

$f(x,y) = xy + \bar{x}\bar{y} + \underbrace{\bar{x}y + \bar{x}y}_{x+x=x}$ (7)

THEN... $f(x,y) = y(x + \bar{x}) + \bar{x}(\bar{y} + y)$

$\Rightarrow f(x,y) = y + \bar{x}$

$\boxed{f(x,y) = y + \bar{x}}$ IS THE MINIMAL COST REALIZATION ("SYNTHESIS")

THE IDEA IS ALWAYS TO MINIMIZE THE NUMBER OF GATES USED TO IMPLEMENT A PARTICULAR FUNCTION.

So "BETTER" ⇒ "LESS COSTLY"

AND $\boxed{\text{COST} = \#\text{GATES} + \#\text{INPUTS}}$

COST OF CANONICAL SOP

COST_c = 3 AND + 6 INPUT
+ 2 NOT + 2 INPUT
+ 1 OR + 3 INPUT

COST_c = 6 GATES + 11 INPUT

(17)

COST OF SYNTHESIZED CIRCUIT

COST_s = 1 OR + 2 INPUT
1 NOT + 1 INPUT

2 GATES + 3 INPUT

(5)

THIS TERMINOLOGY HAS A DUAL

- SUM TERM $x + y$, $\bar{x} + y + \bar{z}$ (ANY OR TERM)
- MAXTERM – A SUM TERM INCLUDING ALL INPUTS

x	y	z	MAXTERM	
0	0	0	$x + y + z$	M_0
0	0	1	$x + y + \bar{z}$	M_1
0	1	0	$x + \bar{y} + z$	M_2
0	1	1	$x + \bar{y} + \bar{z}$	M_3
⋮			⋯	
1	1	1	$\bar{x} + \bar{y} + \bar{z}$	M_7

CAPITAL

FOR OUR PREVIOUS TRUTH TABLE...

x	y	z	$f(x,y,z)$	
0	0	0	0	M_0
0	0	1	0	M_1
0	1	0	0	M_2
0	1	1	1	M_3
1	0	0	0	M_4
1	0	1	1	M_5
1	1	0	1	M_6
1	1	1	1	M_7

$\boxed{f(x,y,z) = \prod M(0,1,2,4)}$

PRODUCT OF SUMS (POS) → AND OF ALL OR TERMS

CANONICAL POS – UNIQUE EXPRESSION WHERE EACH TERM IS A MAXTERM

EXAMPLE: is this SOP or POS?

EXPRESS $f(x,y,z) = (x+y)(y+z)$ in CANONICAL POS form

↓ NEED TO SHOW ALL INPUTS ARE IN USE

RECALL $a+bc = (a+b)(a+c)$

so $f = [(x+y)+z\bar{z}][x\bar{x}+(y+z)] = (x+y+z)(x+y+\bar{z})(x+y+z)(\bar{x}+y+z)$

(REPEATED!)

$(x+y)$ OR 0 — 0 OR $(y+z)$

$f = (x+y+z)(x+y+\bar{z})(\bar{x}+y+z)$

$z=0 \rightarrow out=0$

$z=1 \rightarrow out=0$

EXAMPLE

EXPRESS $f(a,b,c) = \bar{a}b + c$ in CANONICAL SOP form

↳ NEED TO MAKE ALL INPUTS SHOW UP

$f = \bar{a}b(c+\bar{c}) + (\bar{a}+a)(\bar{b}+b)c$

1 1 1

$c=0 \rightarrow out=1$

$c=1 \rightarrow OUT=1$

EXAMPLE: GIVEN THE TRUTH TABLE, EXPRESS f in SOP & POS

x	y	f
0	0	0
0	1	1
1	0	1
1	1	1

SOP → $\sum$ MINTERMS WHERE $f=1$

POS → $\prod$ MAXTERMS WHERE $f=0$

SOP → $f = \bar{x}y + x\bar{y} + xy$

$\bar{x}y + xy + x\bar{y} + xy$

$y(\bar{x}+x) + x(\bar{y}+y)$

1 1

$\boxed{f = y + x}$ ← EITHER SOP ($f=1$) OR POS ($f=0$) WILL FULLY REPRESENT THE SYSTEM FUNCTIONALITY

POS → $\boxed{f = x + y}$

MAXTERM M0 ONLY!

TO SUMMARIZE:

- SUM OF PRODUCTS (SOP)
  - → IT IS THE OR OF AND TERMS, FOR WHICH $f=1$
  - → USES MINTERMS → $f = \sum$ (MINTERMS WHERE $f=1$)
  - (AND) OR (AND) ⟶ NAND NAND

RECALL

DE MORGAN

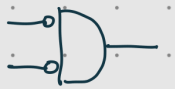
$\overline{xy} = \bar{x} + \bar{y}$

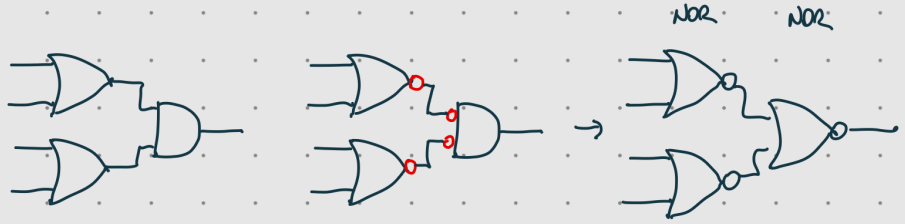
- PRODUCT OF SUMS (POS)
  - IT IS THE AND OF OR TERMS, FOR WHICH $f=0$
  - USES MAXTERMS → $f = \prod$ (MAXTERMS WHERE $f=0$)
  - (OR) AND (OR) ⟶ NOR NOR

RECALL

DE MORGAN

$\overline{x+y} = \bar{x}\cdot\bar{y}$ →





EXAMPLE: GIVEN THE LOGIC FUNCTION

$$f(a,b,c) = \sum m(1,3,5,6,7)$$

1) BUILD THE TRUTH TABLE THAT CHARACTERIZES THE SYSTEM

CLUES: $a, b, c \Rightarrow$ 3 INPUTS

$\sum m \Rightarrow$ SUM OF MINITERMS, WHERE $f = 1$

a	b	c	f	
0	0	0	0	m_0
0	0	1	1	m_1
0	1	0	0	m_2
0	1	1	1	m_3
1	0	0	0	m_4
1	0	1	1	m_5
1	1	0	1	m_6
1	1	1	1	m_7

2) PRESENT A SIMPLIFIED SOP FUNCTION

$$f = \bar{a}\bar{b}c + \bar{a}bc + a\bar{b}c + ab\bar{c} + abc$$

$(a+\bar{a})bc$, $(\bar{a}+a)\bar{b}c$

$$f = \bar{a}\bar{b}c + \bar{a}bc + a\bar{b}c + ab\bar{c} + abc + abc$$

$(a+\bar{a})bc$, $(\bar{a}+a)\bar{b}c$, $(\bar{c}+c)ab$

$$f = \bar{b}c + bc + ab$$

$$= c(\bar{b}+b) + ab \Rightarrow \boxed{f = c + ab}$$ SOP

3) PRESENT A SIMPLIFIED POS FUNCTION

$$f = (a+b+c)(a+\bar{b}+c)(\bar{a}+b+c)$$

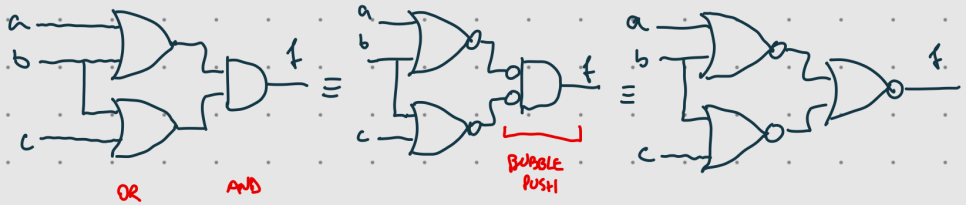
$$\left[(a+c)+b\bar{b}\right]\left[(b+c)+a\bar{a}\right]$$

$$f = (a+c)(b+c) \Rightarrow \boxed{f = c + ab}$$ POS

SAME

4) PRESENT A CIRCUIT FOR POS, USING ONLY NOR GATES

$$f = (a+c)(b+c)$$



EXAMPLE EXPRESS $f(x,y,z) = (x+y)(y+z)$ IN POS CANONICAL FORM

CLUE: POS → ΠM MAXTERM

CANONICAL → ALL INPUTS MUST BE REPRESENTED IN EVERY FACTOR OF THE EQUATION

WE KNOW IT IS $f(x,y,z) \Rightarrow$ 3 inputs

WE CAN DO $f = (x+y+0)(0+y+z)$ — NO CHANGE

$$f = (x+y+z\bar{z})(x\bar{x}+y+z)$$

NOW RECALL $a+bc = (a+b)(a+c)$ (12)

$$f = (\underbrace{x+y}_{a} + \underbrace{z\bar{z}}_{bc})(\underbrace{x\bar{x}}_{bc} + \underbrace{y+z}_{a})$$

THEN $f = (x+y+z)(x+y+\bar{z})(x+y+z)(\bar{x}+y+z)$

ONLY NEED ONE

$x \cdot x = x$ (8)

$$\boxed{f(x,y,z) = (x+y+z)(x+y+\bar{z})(\bar{x}+y+z)}$$ POS CANONICAL

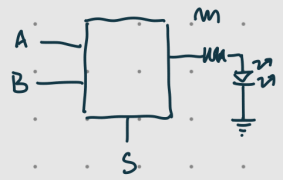
05. MULTIPLEXERS

ECE241 ① 2023 bkje

LET US DESIGN A CIRCUIT TO CHECK THE LOGIC STATE OF TWO SIGNALS, A & B. THE SIGNALS TO BE CHECKED ARE SELECTED BY A SWITCH S

NOTE: 3 INPUTS - A, B, S
1 OUTPUT - m

GENERALIZED BLOCK DIAGRAM



TRUTH TABLE

SELECTOR ↓

	S	A	B	m
SELECT A	0	0	0	0
	0	0	1	0
	0	1	0	1
	0	1	1	1
SELECT B	1	0	0	0
	1	0	1	1
	1	1	0	0
	1	1	1	1

IN WORDS:
IF A IS SELECTED AND HIGH, LED ON
B IS SELECTED AND HIGH, LED ON

EQUATION

$$m = \bar{S}A\bar{B} + \bar{S}AB + S\bar{A}B + SAB$$
$$= \bar{S}\left(A\bar{B} + AB\right) + S\left(\bar{A}B + AB\right)$$
$$= \bar{S}\left(A\left(\underbrace{\bar{B}+B}_{1}\right)\right) + S\left(B\left(\underbrace{\bar{A}+A}_{1}\right)\right)$$

$$\boxed{m = \bar{S}A + SB}$$ "SYNTHESIZED"

THEN I CAN CREATE A TABLE TO EXPRESS THAT

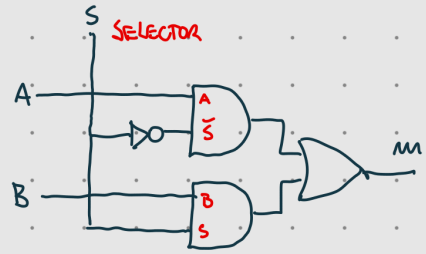
S	m
0	A
1	B

if S
then m = B
else m = A

THIS IS A **2:1 MULTIPLEXER**

ECE241 ② 2023 bkje

LET'S IMPLEMENT $m = \bar{S}A + SB$ WITH LOGIC GATES



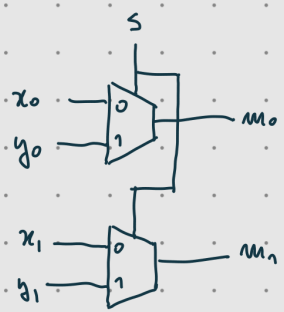
- NOTE THAT A & B ARE ONE BIT INPUTS
- WE CAN USE THE SAME IDEA TO SELECT INPUTS WITH MULTIPLE BITS

② SELECTING 2 BIT INPUTS (TWO BITS OF SELECTED INPUT WILL GO TO OUTPUT)

INPUT $X = (x_1, x_0)$
$Y = (y_1, y_0)$
$M = (m_1, m_0)$

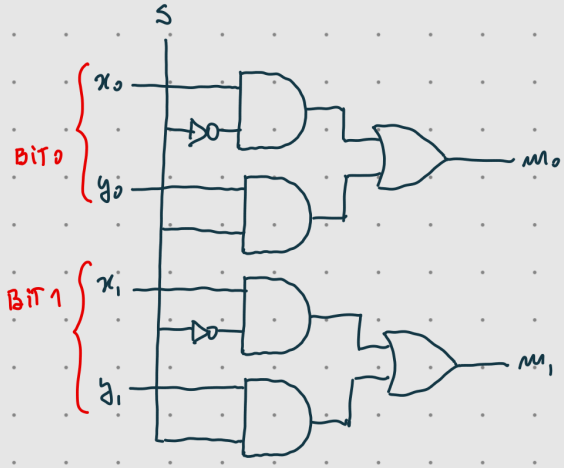
NOTE: THE SELECTOR IS STILL ONE BIT (SELECTING BETWEEN 2 INPUTS)

S	M	
0	x_1	x_0
1	y_1	y_0
	BIT 1	BIT 0



NOTE THAT S SELECTS BOTH BITS OF EACH INPUT

S=0 → x_1, x_0
S=1 → y_1, y_0



FOR THOUGHT:

- WHAT IF WE WANTED TO SELECT FROM 3 DIFFERENT INPUTS?

  ANS: THE SELECTOR WOULD NEED 2 BITS

- FOR $4 < \text{inputs} \leq 8$ → 3 BIT SELECTOR

  etc.

ECE241 ③ 2023 bkje

LET US TAKE A BREAK FROM DRAWING CIRCUITS AND SEE HOW TO DESCRIBE THEM IN CODE

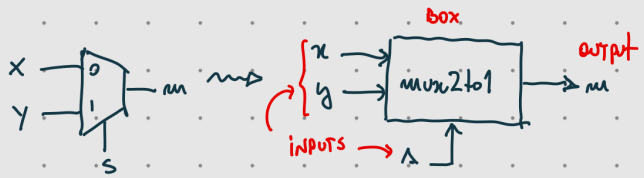
VERILOG

- IT IS A HARDWARE DESCRIPTION LANGUAGE (HDL) → VERILOG / VHDL / SYSTEM VERILOG
- IT IS USED TO DESCRIBE THE CIRCUITS TO BE SYNTHESIZED
  "SYNTHESIS" ⇒ A COMPILATION FROM DESCRIPTION IN CODE TO ACTUAL CIRCUIT
  - IT TAKES INTO ACCOUNT THE STRUCTURE OF THE CIRCUIT AND THE DESCRIPTION OF ITS BEHAVIOUR

- **STRUCTURAL VERILOG** – DESCRIBES CIRCUITS GATE BY GATE & HOW THEY CONNECT
  "OUTPUT OF A TO INPUT OF B"

- **BEHAVIOURAL VERILOG** – IT IS A HIGHER LEVEL OF ABSTRACTION IN WHICH LOGIC EQUATIONS ARE USED
  - THEN THE TOOL (e.g. QUARTUS) OPTIMIZES THE CIRCUIT (WHICH GATES TO USE & HOW)

EXAMPLE 2:1 MUX (ONE BIT)



"BOX"

```
module mux2to1 (x, y, s, m)        PORTS (inputs & outputs)
    input  x, y, s;
    output m;                      } who is who (LOOK AT THE ARROWS!)
    assign m = (~s & x) | (s & y); } WHAT THE "BOX" DOES
end module
```

SECT 4.6.5 of book Operators

ECE241 ④ 2023 bkje

"ASSIGN" MEANS: WHEN x, y & s CHANGE, m is RECALCULATED

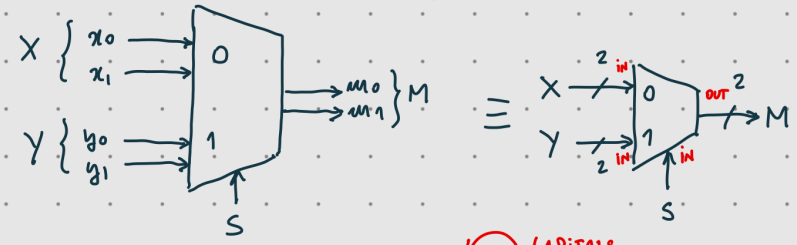
- IT IS A "CONTINUOUS" ASSIGNMENT (p. 73 BOOK)
- IT IS A BEHAVIOURAL SPECIFICATION, WHICH DEFINES HOW THE MUX OPERATES: m (AN OUTPUT) COMES FROM (s̄ AND x) OR (s AND y)

- ONCE THE MODULE IS DESIGNED, IT CAN BE → SIMULATED / REUSED OVER & OVER / MODIFIED

- THE PROCESS:
  1) DRAW THE SCHEMATIC DIAGRAM OF THE CIRCUIT
     i.e. IDENTIFY INPUTS, OUTPUTS, FUNCTIONALITY
  2) DESCRIBE THE CIRCUIT USING VERILOG
     ATTENTION TO HOW THE ARROWS IN YOUR DIAGRAM CONNECT

LET US CREATE A MUX MODULE WITH 2 INPUTS, AND EACH INPUT BEING 2 BITS WIDE

- 2 INPUTS → 1 BIT SELECTOR
- 2 BITS WIDE ⇒ OUTPUT ALSO 2 BITS WIDE




```
module mux2to1_2bit (X, Y, S, M);        CAPITALS
    input [1:0] X, Y;
    input S;                             [a:b] DEFINES WIDTH, INDEX VALUES, BIT ORDER
    output [1:0] M;
    assign M[0] = (~S & X[0]) | (S & Y[0]);   } NOW OUTPUT IS 2 BITS,
    assign M[1] = (~S & X[1]) | (S & Y[1]);   } EACH BIT CALCULATED SEPARATELY
end module
```

WE CAN DO THIS ↑ OR WE CAN USE OUR mux2to1 MODULE

SIMILAR TO THE IDEA OF WRITING A FUNCTION IN C, I CAN CREATE A SMALL WORKING MODULE AND ASSEMBLE LARGER MODULES WITH IT.

- MAKES IT EASIER TO DEBUG.

⊛ HIERARCHY IN VERILOG

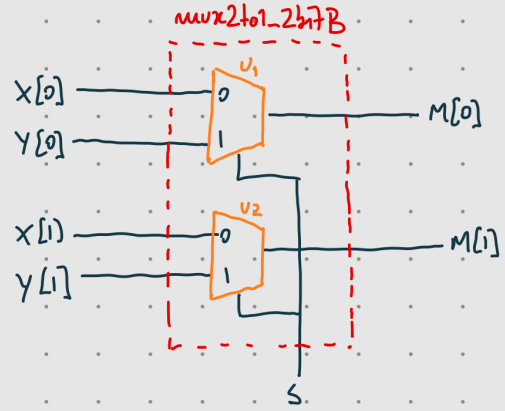
LET US CALL UPON (INSTANTIATE) THE 2:1 MUX (1 BIT) TO CREATE A 2-BIT 2:1 MUX

```
module mux2to1_2bitB (x, y, s, M);
    input [1:0] x, y;
    input s;
    output [1:0] M;
    mux2to1 U1 (x[0], y[0], s, M[0]);
    mux2to1 U2 (x[1], y[1], s, M[1]);
end module
```

SAME ORDER AS IN THE ORIGINAL mux2to1 MODULE

INPUTS — OUTPUT

CALL INSTANCES OF mux2to1 INSTEAD OF THE ASSIGN STATEMENT



NOTE THAT IF WE WISH TO INSTANTIATE A MODULE WITH 100 PORTS, AND WE HAVE TO FOLLOW THE ORDER IN WHICH THE PORTS WERE DECLARED, THIS MAKES INSTANTIATING VERY DIFFICULT. WE CAN DO, INSTEAD:

```
mux2to1 U1 (.x(x[0]), .s(s), .y(y[0]), .m(M[0]));
```

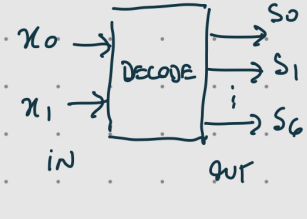
THIS IS BETTER STYLE, AND ORDER-INDEPENDENT.

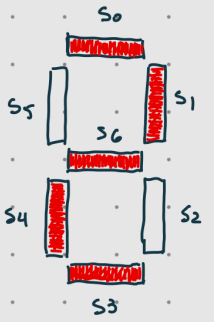
⊛ DESIGN EXAMPLE (book, pg 63)

- WE HAVE A 2 bit NUMBER $X(x_1, x_0)$. DISPLAY THE DECIMAL NUMBER EQUIVALENT TO THE 2 BIT BINARY ON A 7 SEGMENT DISPLAY.

(IN OTHER WORDS: DECODE THE 2 bit BINARY INTO DECIMAL & LIGHT UP THE LEDs)

1) DRAW THE DIAGRAM





- NUMBER 0
  ALL LIT EXCEPT S6
- NUMBER 1
  S1 & S2 LIT, OTHERS OFF

HOW ABOUT #4?

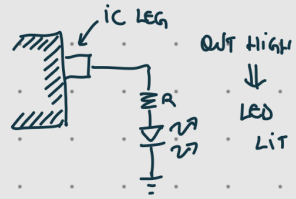
NEED MORE BITS...

2) CREATE THE TRUTH TABLE

#2

DEC	x_1	x_0	S_6	S_5	S_4	S_3	S_2	S_1	S_0
0	0	0	0	1	1	1	1	1	1
1	0	1	0	0	0	0	1	1	0
2	1	0	1	0	1	1	0	1	1
3	1	1	1	0	0	1	1	1	1

OUTPUT 1 WHEN LED IS ON



3) WRITE ONE BOOLEAN EQUATION FOR EACH OUTPUT ($S_0 \rightarrow S_6$)

ASK WHICH ONE IS SIMPLER: USING MAXTERMS & MINTERMS?)

HAVE 10 ZEROS & 18 ONES ... LET'S USE THE ZEROS → MAXTERMS

$S_0 = x_1 + \overline{x_0}$

$S_1 = 1$

$S_2 = \overline{x_1} + x_0$

$S_3 = S_0$

IF WE'RE USING MAXTERMS,

- WE USE THE ZEROS FROM THE T.T.
- THE ONES ARE REPRESENTED WITH BAR → $\overline{a}$
- THE OPERATION IS PRODUCT (AND) OF SUMS (OR)

CONTINUING...

$s_4 = (x_1 + \bar{x}_0)(\bar{x}_1 + \bar{x}_0) = \bar{x}_0(x_1 + \bar{x}_1)$ (14) $\Rightarrow s_4 = \bar{x}_0$

$s_5 = (x_1 + \bar{x}_0)(\bar{x}_1 + \bar{x}_0)(\bar{x}_1 + x_0) = \bar{x}_0(\bar{x}_1 + x_0)$

$= \bar{x}_0 \bar{x}_1$ (16) $\Rightarrow s_5 = \bar{x}_0 \bar{x}_1$

$s_6 = (x_1 + x_0)(x_1 + \bar{x}_0) \Rightarrow$ (14) $s_6 = x_1$

NOTE
BOOK PG77
USES MINTERMS

- WE COULD DRAW THE CIRCUIT TO IMPLEMENT THE DECODER WITH DISCRETE LOGIC GATES NOW, OR... WE DESCRIBE THE DECODER IN HDL

4) WRITE THE MODULE IN VERILOG

```
module display2bit (input x1, x0, output [6:0] s);
    assign s[0] = x1 | ~x0;
    assign s[1] = 1'b1;
    assign s[2] = ~x1 | x0;
    assign s[3] = s[0];
    assign s[4] = ~x0;
    assign s[5] = ~x1 & ~x0;
    assign s[6] = x1;
end module
```

CAN DESCRIBE I/O HERE

→ "BINARY VALUE IS ONE"

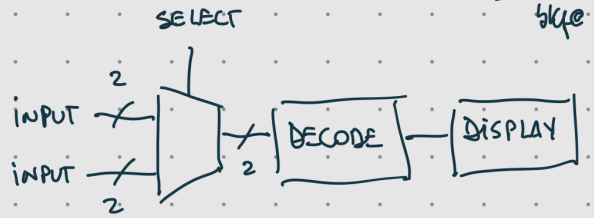
NOTE IN THE LAB ⇒ HEX NUMBER IS DISPLAYED: [0−9, A−f]

- WILL HAVE MORE BITS TO BE ENCODED
- DISPLAY CAN BE CONFUSING

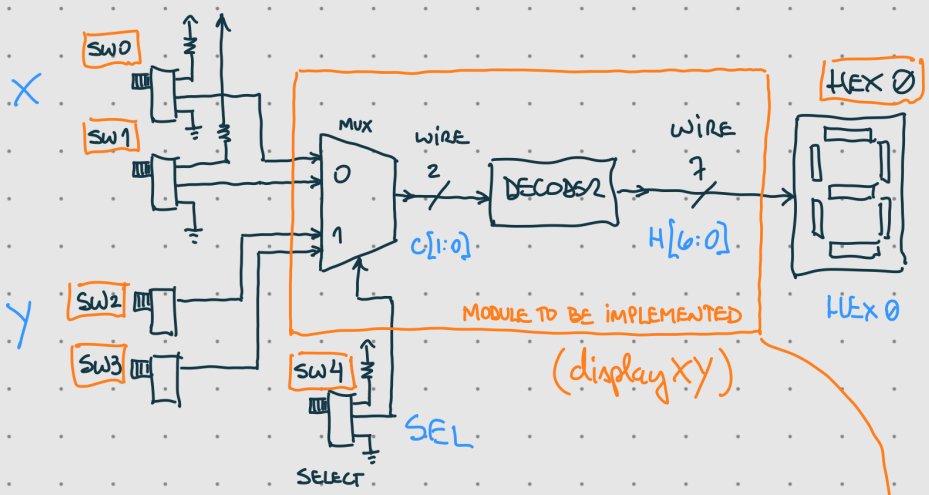
FOR D USE  FOR B USE 

NOW WE HAVE A MUX, WHICH SELECTS 2-BIT INPUTS AND WE HAVE A DECODER THAT INTERPRETS TWO BITS IN ORDER TO DISPLAY THE DECIMAL (OR HEX) EQUIVALENT

DRAW A DIAGRAM



DRAW A MORE DETAILED DIAGRAM



DESCRIBE THE MODULES AND HOW THEY CONNECT IN HDL

```
module displayXY (input [1:0] X,Y, input SEL, output [6:0] HEX0);
    wire [1,0] C;
    wire [6:0] H;
    mux2to1_2bit U1 (.X(X), .Y(Y), .S(SEL), .M(C));
    display2bit U2 (.x1(C[1]), .x0(C[0]), .s(H));
    assign HEX0 = H;
end module
```

FPGA

- FIELD PROGRAMMABLE GATE ARRAY
  - AN ARRAY OF GATES THAT IS PROGRAMMABLE, WITH WIRING THAT IS ALSO PROGRAMMABLE
  - PROGRAMMABLE GATES ARE "LOOK-UP TABLES" (LUT), AND ARE THE LOGIC ELEMENTS OF THE FPGA
    - ONE CAN IMPLEMENT WHATEVER INTERCONNECTION OF GATES USING LUT
    - THINK OF A TABLE OF OUTPUTS, ASSOCIATED WITH ALL POSSIBLE INPUTS
      - YOU — THE DESIGNER — DETERMINES HOW THE OUTPUTS WILL BE.

EXAMPLE: SAY WE WANT TO IMPLEMENT AN ARBITRARY SYSTEM WITH 2 INPUTS USING A LOOK UP TABLE (LUT)

HOW MANY POSSIBLE OUTPUTS CAN WE HAVE? 4

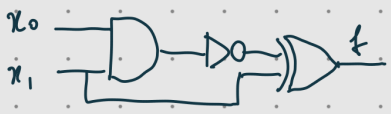
THE LUT WILL HOLD THE TRUTH TABLE OF OUTPUTS FOR THIS ARBITRARY CIRCUIT

x_1	x_0	f
0	0	f_0
0	1	f_1
1	0	f_2
1	1	f_3

THESE WILL BE MEMORY CELLS PROGRAMMED WITH THE VALUES OF f THAT WE WANT FOR OUR CIRCUIT

FOR 2 INPUTS, WE HAVE 4 POSSIBLE OUTPUTS

EX. WHAT WOULD BE THE LUT IMPLEMENTATION OF THE LOGIC CIRCUIT:



ANSWER:

x_1	x_0	f	
0	0	1	f_0
0	1	0	f_1
1	0	1	f_2
1	1	1	f_3

f_0 f_1 f_2 f_3

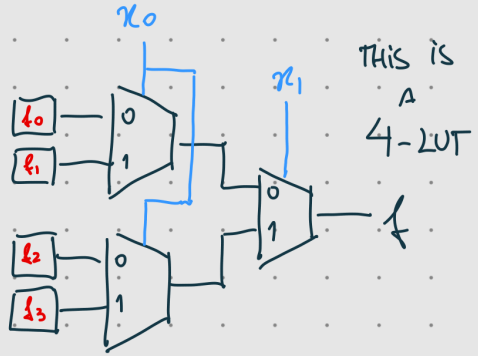
THE VALUES (1, 0, 1, 1) ARE STORED IN MEMORY CELLS AND ARE CHOSEN AS OUTPUT DEPENDING ON THE INPUTS.

⇒ INPUTS SELECT THE OUTPUTS FROM THE LUT

↳ MULTIPLEXERS ARE USED

LOOK UP TABLE (LUT)

x_1	x_0	f
0	0	f_0
0	1	f_1
1	0	f_2
1	1	f_3

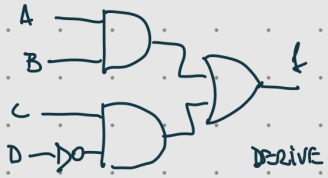


THIS IS A 4-LUT

THE FPGAS IN THE LAB BOARDS (DE1-SoC) HAVE ~85,000 4-LUT

NOT TESTABLE! SEARCH { EEVblog ON YOUTUBE — EPISODE # 496 (AND # 635!) ; NANDLAND ON YOUTUBE — FIRST EPISODE

EXERCISE: WHAT IS STORED IN THE LOOK UP TABLE THAT IMPLEMENTS THE CIRCUIT:



DERIVE TRUTH TABLE

NOTE: 4 INPUTS ⇒ 16 POSSIBLE OUTPUTS

A	B	C	D	f
0	0	0	0	f_0
0	0	0	1	f_1
⋮				
1	1	1	1	f_{15}

ANSWER

1111 | 0100 | 0100 | 0100

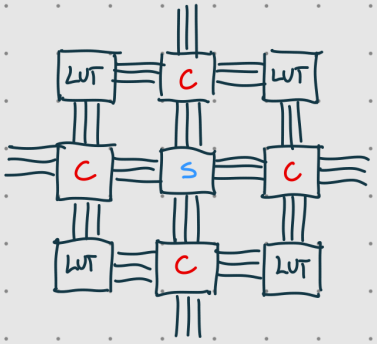
f_{15} f_{12} f_8 f_4 f_0

TO IMPLEMENT THIS LUT, WE NEED TO STORE 16 OUTPUTS (OR GROUPS OF 4)

ECE241 (3) 2023

LOOKING INSIDE AN FPGA

LOGIC ELEMENTS (LUT), CONNECTION BLOCKS (C), SWITCHING BLOCKS (S)



CONNECTION BLOCKS CONNECT THE LOGIC ELEMENTS (LUT) TO WIRES

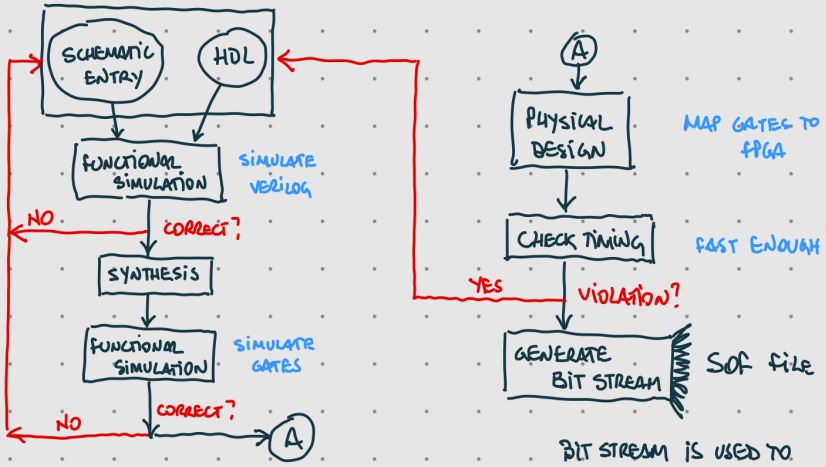
SWITCHING BLOCKS CONNECT WIRES TOGETHER

CONFIGURATION BITS WILL PROGRAM THE LOGIC ELEMENT FUNCTIONS, AND SET THE SWITCH CONNECTIONS (SOF FILE)

- SOF FILE DETERMINE HOW THE PINS WILL BE CONFIGURED

✳ STEPS OF THE DESIGN PROCESS OF AN FPGA-BASED SYSTEM

(AT EVERY STEP, CHECK: DOES IT WORK? IF NOT, GO BACK & REDESIGN)



BIT STREAM IS USED TO PROGRAM THE FPGA

ECE241 (4) 2023

- AT THIS POINT WE HAVE SEEN LOGIC GATES, TRUTH TABLES BOOLEAN ALGEBRA, MUX, LUTs, SOME VERILOG
- LET'S BUILD ON THE HALF ADDER, WE HAD BEFORE, LEADING TO A FULL ADDER (THIS WILL BE THE ARITHMETIC ADDITION OF TWO BITS)

✳ FULL ADDER

- THE IDEA: PER BIT, THE ADDITION MUST TAKE INTO ACCOUNT A "CARRY IN" BIT AND PRODUCE THE RESULT WITH A "CARRY OUT" BIT

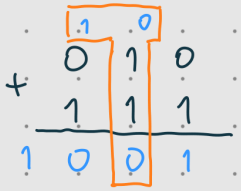
LET'S LOOK AT THE ARITHMETIC ADDITION OF TWO 3-BIT NUMBERS

$a_2\ a_1\ a_0$

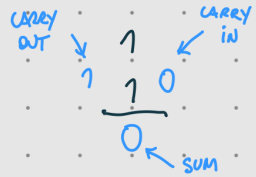
$A = 0\ 1\ 0$

$b_0\ b_1\ b_0$

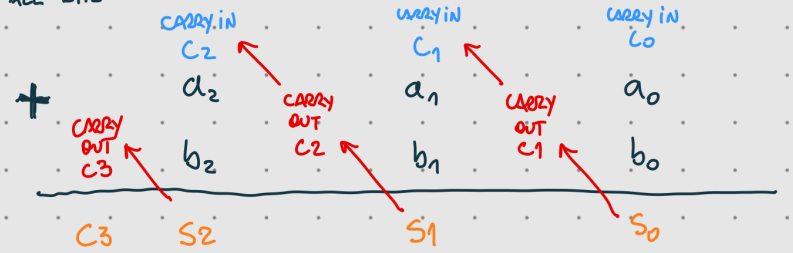
$B = 1\ 1\ 1$



LOOKING AT THE SECOND BIT, WE HAVE

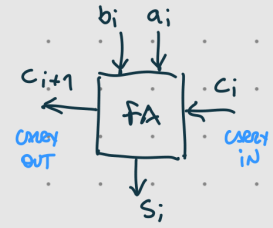


WE CAN SEE THAT THERE IS A "BASE" CONFIGURATION THAT REPEATS FOR ALL BITS



THIS "BASE" CONFIGURATION IS THE 1 BIT FULL ADDER (FA)

THIS IS PER BIT



a_i

c_{i+1} b_i c_i

S_i

LET'S BUILD A TRUTH TABLE FOR THE FULL ADDER

CARRY IN	OPERAND a	OPERAND b	CARRY OUT	SUM
c_i	a_i	b_i	c_{i+1}	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

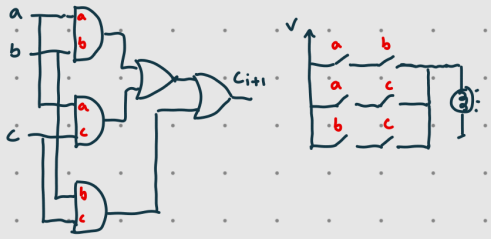
INPUTS | OUTPUTS

CARRY OUT — SOP

$c_{i+1} = a_i b_i + c_i b_i + c_i a_i$

THIS IS CALLED A MAJORITY FUNCTION

- WHEN THE MAJORITY OF THE INPUTS IS 1, c_{i+1} IS 1 (2 OUT OF 3 INPUTS)



As for OUTPUT s_i (using SOP) → $s_i = \bar{a}_i \bar{b}_i c_i + a_i \bar{b}_i \bar{c}_i + \bar{a}_i b_i \bar{c}_i + a_i b_i c_i$

$$\Rightarrow s_i = c_i(\underbrace{\bar{a}_i \bar{b}_i + a_i b_i}_{\overline{a_i \oplus b_i}}) + \bar{c}_i(\underbrace{\bar{a}_i b_i + a_i \bar{b}_i}_{a_i \oplus b_i})$$

RECALL XOR: $\bar{x}y + x\bar{y} = x \oplus y$

LET'S SEE $\overline{\bar{a}\bar{b} + ab} = \overline{\bar{a}\bar{b}} \cdot \overline{ab}$

DE MORGAN: $\overline{x+y} = \bar{x} \cdot \bar{y}$, $\overline{xy} = \bar{x} + \bar{y}$

$\overline{ab} = \bar{a} + \bar{b}$

$\overline{\bar{a} \cdot \bar{b}} = \bar{\bar{a}} + \bar{\bar{b}} = a + b$

THEN $\overline{\bar{a}\bar{b} + ab} = (\bar{a} + \bar{b})(a + b) = \cancel{\bar{a}a}^{0} + \bar{a}b + \bar{b}a + \cancel{\bar{b}b}^{0}$

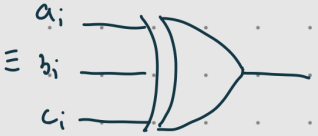
$\bar{a}b + a\bar{b} = a \oplus b$

THEREFORE $s_i = c_i(\overline{a_i \oplus b_i}) + \bar{c}_i(a \oplus b)$

$$s_i = c_i \oplus (a_i \oplus b_i)$$

WE IMPLEMENT s_i WITH XOR GATES

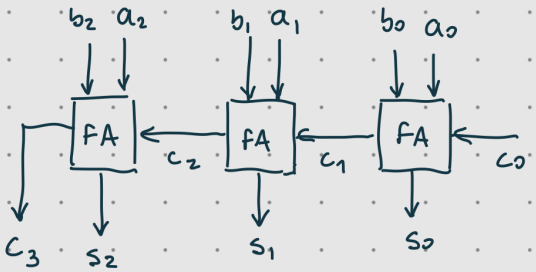
≡



THIS CIRCUIT OUTPUTS 1 WHEN THE INPUTS HAVE AN ODD NUMBER OF ONES (LOOK @ TRUTH TABLE)

- THIS IS KNOWN AS A PARITY (OR ODD) FUNCTION

NOW THAT WE HAVE THE FULL ADDER FOR ONE BIT, LET'S BUILD A THREE BIT ADDER



THIS IS KNOWN AS A "RIPPLE CARRY ADDER"

- NOTE THAT THE CARRY BIT "RIPPLES" THROUGH THE ADDER

IT IS IMPORTANT TO DEVELOP A DIAGRAM PRIOR TO WRITING VERILOG, BECAUSE THE CODE WILL DESCRIBE THE DIAGRAM

- LET US START BY DESIGNING A MODULE FOR THE FULL ADDER, THEN WE INSTANTIATE IT WHEN WE DESIGN THE ADDER

```
module fA (input a, b, cin, output s, cout);
    assign s = a ^ b ^ c;
    assign cout = (a & b) | (b & c) | (a & cin);
end module
```

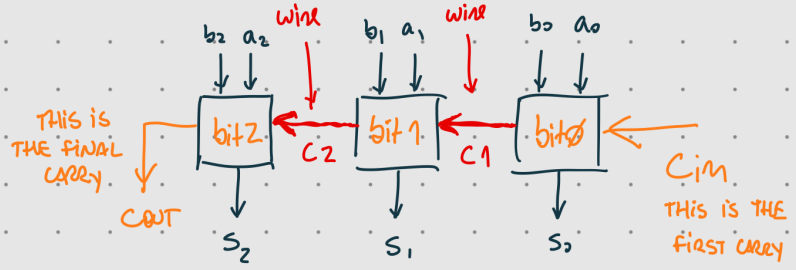
ECE496 (7) 2023

WE NOW CREATE A LARGER MODULE INSTANTIATING AND CONNECTING THE SMALLER MODULES (FA)

```
module adder (A, B, Cin, S, Cout);
    input [2:0] A, B;        WE HAVE 3 BITS NOW
    input Cin;
    output [2:0] S;
    output Cout;
    wire C1, C2;
    FA bit0 (A[0], B[0], Cin, S[0], C1);
    FA bit1 (A[1], B[1], C1, S[1], C2)
    FA bit2 (A[2], B[2], C2, S[2], Cout);
end module
```

THESE ARE WIRES

INPUT ⇓ OUTPUT ⇓



- NOTE THAT THE PARTS OPERATE IN PARALLEL, NOT SEQUENTIALLY AS IN A C PROGRAM
- ALSO, A WIRE PASSES A VALUE FROM BLOCK TO BLOCK, BUT IT DOES NOT STORE A VALUE.

ECE241 (8) 2023

(*) ALWAYS BLOCKS, MULTIPLEXERS REVISITED

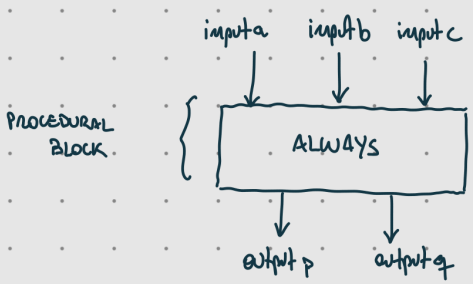
- THE ALWAYS BLOCK DEFINES BEHAVIOUR USING PROCEDURAL STATEMENTS
- IT IS A DESCRIPTION OF A "PATTERN" THAT EXECUTES IN ORDER, SEQUENTIAL

```
always @ (*)
    begin
        (PROCEDURAL STATEMENTS)
    end
```

"ALWAYS AT EVERY INPUT" (*)

PROCEED IN ORDER ↓

THIS CAN BE REPRESENTED VISUALLY AS



THE EXECUTION IS SENSITIVE TO ALL INPUTS (*)

OUTPUTS WILL BE REGISTERED (i.e. WILL NOT CHANGE) UNTIL THE ALWAYS BLOCK RUNS AGAIN VIA REG COMMAND

LET'S LOOK @ THE 2:1 MUX AGAIN, BUT NOW WE'LL USE THE ALWAYS BLOCK

```
reg m;
always @ (*)
    begin
        m = ~s & x | s & y;
    end
```

OR...

```
m = s ? y : x
```

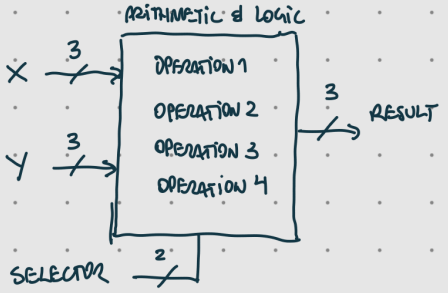
WE WROTE BEFORE...

```
module mux2to1 (x, y, s, m)
    input x, y, s;
    output m;
    assign m = (~s & x) | (s & y);
end module
```

⊛ DESIGN EXAMPLE : ALU

ECE241 ⑨ 2023

- LET US DESIGN AN ARITHMETIC & LOGIC UNIT
- IT WILL TAKE AS INPUTS:
  - TWO 3-BIT OPERANDS
  - ONE 2-BIT SELECTOR, WHICH ALLOWS FOR SELECTION BETWEEN 4 OPERATIONS
- FIRST CRUDE DIAGRAM



SAY WE WISH TO HAVE THE OPERATIONS

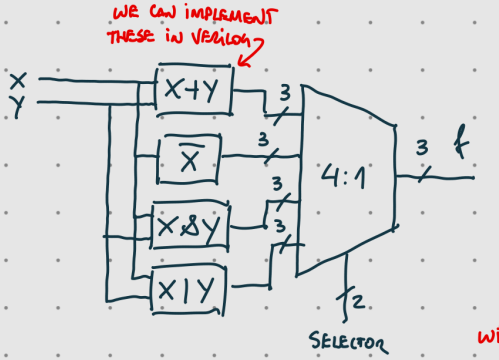
ADDITION	(ARITHMETIC)
NOT	(LOGIC)
AND	(LOGIC)
OR	(LOGIC)

SELECTOR INPUT	OUTPUT
0 0	X + Y
0 1	$\overline{X}$
1 0	X & Y
1 1	X | Y

input output

WE CAN SEE THAT THIS IS A 4:1 MUX

A MORE REFINED SCHEMATIC




```
reg f;
always @ (*)
    case (SELECTOR)
        2'b00 : f = X + Y;
        2'b01 : f = ~X;
        2'b10 : f = X & Y;
        2'b11 : f = X | Y;
        default : f = 3'b000;
    end case
end
```

EACH OPERAND IS 3 BITS WIDE

IMPLEMENT MUX WITH CASE STATEMENT

MUST HAVE DEFAULT CASE

3 bits output

KARNAUGH MAPS (K-MAPS)

- SO FAR, WE HAVE USED THE TRUTH TABLE ALONG WITH BOOLEAN ALGEBRA TO FIND THE MINIMIZED/SIMPLIFIED/SYNTHESIZED VERSION OF THE BOOLEAN EQUATION THAT FULLY DESCRIBES A DIGITAL SYSTEM.

- NOW, WE WILL USE A VISUAL TOOL TO EFFECT THIS SIMPLIFICATION

- SAY WE HAVE THE FOLLOWING TRUTH TABLE

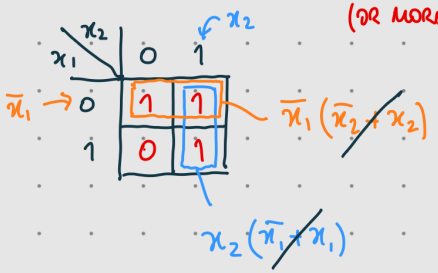
x_1	x_2	f
0	0	1
0	1	1
1	0	0
1	1	1

I CAN THEN WRITE (SOP)

$$f = \bar{x}_1\bar{x}_2 + \bar{x}_1 x_2 + x_1 x_2$$
$$= \bar{x}_1\bar{x}_2 + \bar{x}_1 x_2 (+\bar{x}_1 x_2) + x_1 x_2$$
$$= \bar{x}_1(\bar{x}_2 + x_2) + x_2(\bar{x}_1 + x_1)$$
$$= \bar{x}_1 + x_2 \leftarrow \text{SYNTHESIZED}$$

I CAN BUILD A MAP FROM THE TRUTH TABLE THAT IDENTIFIES THE VARIABLES TO BE ELIMINATED

- I WILL LOOK FOR PATTERNS OF ONES (SOP) THAT WILL ALLOW ME TO ELIMINATE ONE (OR MORE) OF THE VARIABLES

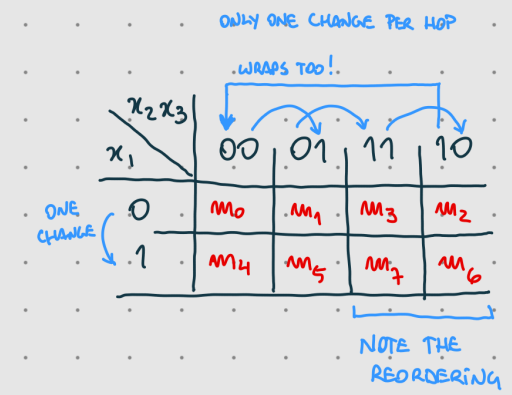


$$f = \bar{x}_1 + x_2$$

WHEN WE HAVE 3 VARIABLES, THAT IS $f(x_1, x_2, x_3)$ WE REORDER THE MAP SO THAT TWO ADJACENT ROWS OR COLUMNS WILL HAVE ONLY ONE OF THE VARIABLES CHANGING.

SOP

x_1	x_2	x_3	f
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7



LET'S WRITE THE FULL EQUATION FROM THE K-MAP USING THE MINTERMS WE'LL DO THAT FOR $x_1 = 0$ JUST TO ILLUSTRATE THE REASON FOR REORDERING

FOR THE LINE $x_1 = 0$ WE HAVE

$$\bar{x}_1\bar{x}_2\bar{x}_3 + \bar{x}_1\bar{x}_2 x_3 + \bar{x}_1 x_2\bar{x}_3 + \bar{x}_1 x_2 x_3$$
$$0\ 0\ 0 \quad 0\ 0\ 1 \quad 0\ 1\ 0 \quad 0\ 1\ 1$$
$$m_0 \quad m_1 \quad m_3 \quad m_2$$

ONLY ONE CHANGE
EASIER TO VISUALLY IDENTIFY ON K MAP
- CAN APPLY PROPERTY RIGHT AWAY

EXAMPLE

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

PROCESS:

a) CREATE K-MAP MAKING SURE ONLY ONE VARIABLE (INPUT) CHANGES VALUE PER COLUMN OR ROW

x_1 \ x_2x_3	00	01	11	10
0	0	0	1	1
1	1	0	0	1

ECE241 ③ 2023

b) IDENTIFY THE EVEN-NUMBERED (POWER OF TWO SIZED) GROUPS OF ONES (SOP) INCLUDING "WRAPPING" GROUPS

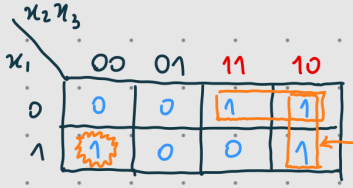
c) ELIMINATE FROM THE GROUPING THE VARIABLE THAT CHANGES

TOP LINE $x_1=0 \rightarrow \bar{x}_1 x_2 x_3 + \bar{x}_1 x_2 \bar{x}_3 = \bar{x}_1 x_2$

BOTTOM LINE $x_1=1 \rightarrow x_1 \bar{x}_2 \bar{x}_3 + x_1 x_2 \bar{x}_3 = x_1 \bar{x}_3$

THEREFORE, $\boxed{f = \bar{x}_1 x_2 + x_1 \bar{x}_3}$

QUESTION: WE SEE A GROUPING IN COLUMN 10. WHY DON'T WE USE THAT?

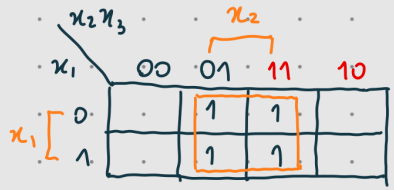


$f = \bar{x}_1 x_2 + x_2 \bar{x}_3 + x_1 \bar{x}_2 \bar{x}_3$

SELECTING THE GROUP ON COLUMN 10 WOULD ELIMINATE VARIABLE x_1 BUT WOULD LEAVE A 1 ON $x_1 \bar{x}_2 \bar{x}_3$ THEREFORE NOT ACHIEVING A MINIMIZED BOOLEAN EQUATION FOR THE SYSTEM

EXAMPLE

x_1	x_2	x_3	f
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	0
1	1	1	1



WE GROUP ALL FOUR 1 TO ELIMINATE TWO VARIABLES: x_1 & x_2

$\bar{x}_1 \bar{x}_2 x_3 + \bar{x}_1 x_2 x_3 + x_1 \bar{x}_2 x_3 + x_1 x_2 x_3$

$\boxed{f = x_3}$

ECE241 ④ 2023

EXAMPLE

x_1	x_2	x_3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

K-MAP

$\boxed{f = \bar{x}_2 \bar{x}_3 + x_1}$

THINGS TO REMEMBER:

(A) ORDER OF VARIABLES → TO EXPOSE WHICH INPUTS ARE CONSTANT & WHICH INPUTS CHANGE

(B) LOOK FOR EVEN-NUMBERED (POWERS OF TWO) GROUPS OF 1
→ TO ELIMINATE THE TERMS THAT CHANGE FROM THE PRODUCT TERM
→ WE THEN USE THE REMAINING TERMS TO FORM THE PRODUCT TERM (SOP)

LET'S LOOK @ SOME DEFINITIONS

⊖ LITERAL - IS A VARIABLE OR ITS COMPLEMENT. E.G. $x_1, \bar{x}_1, x_3, \bar{x}_4$ etc

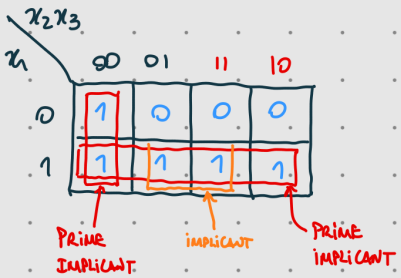
⊖ IMPLICANT - IS ANY PRODUCT TERM FOR WHICH THE FUNCTION IS ONE ($f=1$)

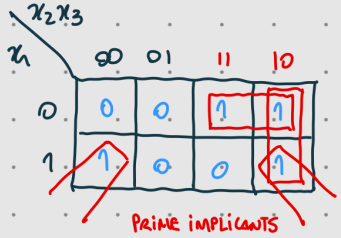
e.g.:

5 MINTERMS (ALL VARIABLES!)
+ 5 PAIRS OF ONES
+ 1 GROUP OF FOUR ONES
11 IMPLICANTS

THE 5 MINTERMS → $\bar{x}_1 \bar{x}_2 \bar{x}_3$, $x_1 \bar{x}_2 \bar{x}_3$, $x_1 \bar{x}_2 x_3$, $x_1 x_2 \bar{x}_3$, $x_1 x_2 x_3$

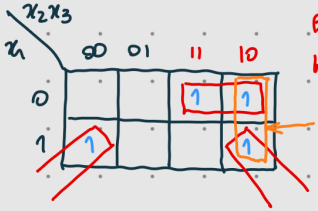
⊖ PRIME IMPLICANT - GROUPS OF 1 THAT CANNOT BE MADE LARGER





- COVER - ANY GROUP OF IMPLICANTS THAT COVER ALL ONES
  - A SET OF ALL MINTERMS FOR WHICH $f=1$ IS A COVER
  - A SET OF ALL PRIME IMPLICANTS IS A COVER (pg 88 book)

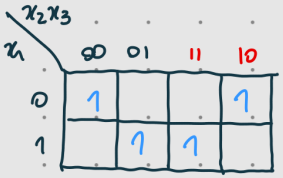
- ESSENTIAL PI - A PRIME IMPLICANT THAT MUST BE INCLUDED IN ANY COVER OF THE FUNCTION



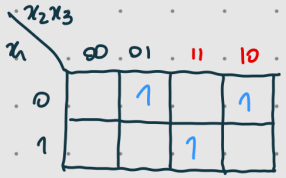
ESSENTIAL PI INCLUDES A 1 THAT HAS BEEN ONLY CIRCLED ONCE

TO SUMMARIZE: TRY TO GROUP IN THE SMALLEST NUMBER OF GROUPS THE LARGEST NUMBER OF ONES

EXAMPLE

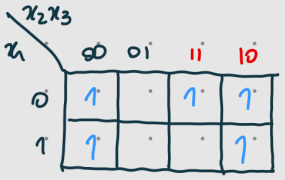


EXAMPLE



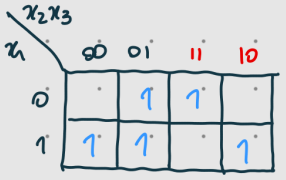
$\bar{x}_1\bar{x}_2x_3 + x_1x_2x_3 + \bar{x}_1x_2\bar{x}_3$

EXAMPLE

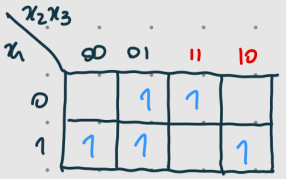


$\bar{x}_1x_2 + x_3$

EXAMPLE



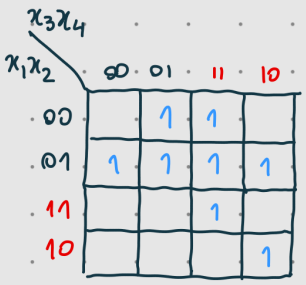
$\bar{x}_1x_3 + x_1\bar{x}_2 + x_1\bar{x}_3$

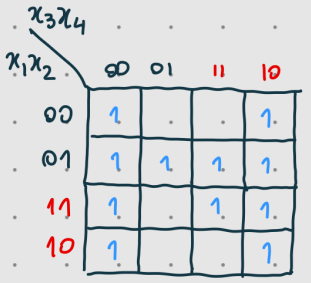


SAME!

$\bar{x}_1x_3 + x_1\bar{x}_3 + \bar{x}_2x_3$

EXAMPLE



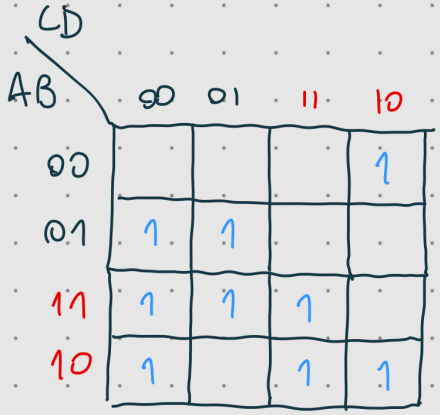


EXAMPLE: GIVEN $f(A,B,C,D)$, FIND PRIME IMPLICANTS & ESSENTIAL PI

$$f = \sum m(2,4,5,8,10,11,12,13,15)$$

TT

A	B	C	D	f
0	0	0	0	m_0
0	0	0	1	m_1
0	0	1	0	m_2
0	0	1	1	m_3
0	1	0	0	m_4
0	1	0	1	m_5
0	1	1	0	m_6
0	1	1	1	m_7
1	0	0	0	m_8
1	0	0	1	m_9
1	0	1	0	m_{10}
1	0	1	1	m_{11}
1	1	0	0	m_{12}
1	1	0	1	m_{13}
1	1	1	0	m_{14}
1	1	1	1	m_{15}



- SUMMARY
  - FIND ALL PRIME IMPLICANTS → LARGEST GROUPS OF 1
  - IDENTIFY ESSENTIAL PI
    - IF ALL 1s ARE COVERED - YOU'RE DONE
  - CHOOSE NON-ESSENTIAL PIs FOR A MINIMUM COST COVER

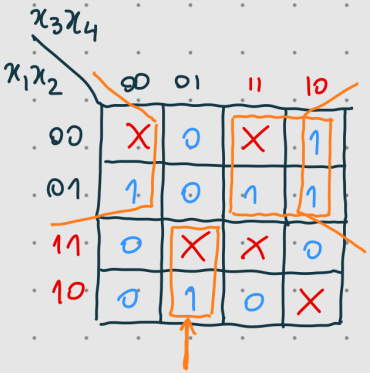
(*) THE USE OF DON'T CARE (IRRELEVANT CONDITION)

- THE "DON'T CARE" CONDITION SHOWS WHEN THE OUPUT CAN BE 0 OR 1 EITHER BECAUSE AN INPUT CONDITION WILL NEVER HAPPEN OR THE OUPUT FOR THE PARTICULAR CASE CAN EITHER BE 1 OR 0.

- IN THE PROCESS OF MINIMIZATION, GROUP THE "DON'T CARE" ALONG WITH 1 (SOP) OR 0 (POS).

DON'T CARES DO NOT HAVE TO BE USED, BUT THEY CAN BE OF GREAT ADVANTAGE WHEN USED

EXAMPLE



$$f = \overline{x}_1 x_3 + \overline{x}_1 x_4 + x_1 \overline{x}_3 x_4$$

(from IDDETA pg 140)

NOTE HERE: THE 1 MUST BE USED, THE X NOT NECESSARILY
- THE X COMES TO OUR ADVANTAGE

EXAMPLE LET'S USE PoS

$$f(x_1,x_2,x_3,x_4) = \prod M(2,3,4,5,6,8,9) + \sum d(10,11,12,13,14,15)$$

EXPRESSED AS $\prod$ MAXTERMS & $\sum$ DON'T CARES

PoS → LOOK @ ZEROS & GROUP THEM

POS FORM (ZEROS)

ECE241 (9) 2023

x_1x_2 \ x_3x_4	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	d	d	d	d
10	1	1	d	d

$f = (x_1 + x_2 + x_3)(\bar{x}_2 + \bar{x}_3 + \bar{x}_4)$

OR, 3 IN — OR, 3 IN, 3 NOT, 3 IN

2 INPUT AND

COST: 6 GATES, 11 INPUT = 17

SOP FORM (ONES)

x_1x_2 \ x_3x_4	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	d	d	d	d
10	1	1	d	d

$f = x_1 + \bar{x}_2 x_3 + x_2 \bar{x}_3 + x_3 \bar{x}_4$

AND, 2 IN, NOT, 1 IN — AND, 2 IN, NOT, 1 IN — AND, 2 IN, NOT, 1 IN

4 INPUT OR

COST: 7 GATES, 13 INPUT = 20

NOTE: WE DON'T NEED TO COUNT INVERTERS INTO THE COST; WE MAY ESTIMATE WITHOUT INVERTERS

WITHOUT COUNTING INVERTERS, WE HAVE:

POS COST: 3 GATES + 8 INPUTS = 11 ← LESS "REAL ESTATE"

SOP COST: 4 GATES + 10 INPUTS = 14

ECE241, Supplement, Oct. 2019
ECE, University of Toronto

5 Variable Karnaugh Map

Bruno Korst - bkf@ece.utoronto.ca

Abstract
This supplement illustrates the configuration, interpretation and use of a 5 variable Karnaugh Map (K-Map).

The 4 Variable K-Map

Below is the general representation of a 4 Variable K-Map. The output of a system represented through this K-Map is a function of 4 input variables: A, B, C and D. A Truth Table for the system must be used to populate this K-Map with 0s and 1s. This K-Map format presented on Figure 1 will be the basis for the 5 variable K-Map.

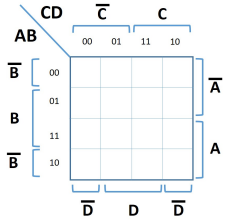


Figure 1. The 4 Variable K-Map

The 5 Variable K-Map

If the system to be mapped has 5 input variables, it follows that one can represent the 4 variable K-Map twice. For the case below, one 4 variable K-Map is representing the outputs for B, C, D and E when variable A is 0 (A-bar), and the other K-Map is representing the cases when the A is 1 (B, C, D and E are clearly also represented there).

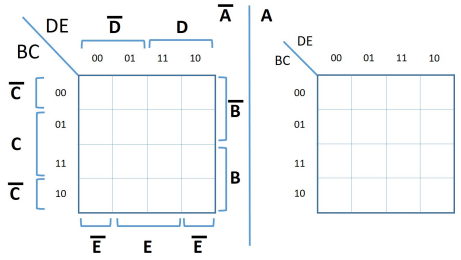


Figure 2. On the left is the K-Map setup for outputs related to $\overline{A}$; on the right, to A

Finding Each Variable

Below it is shown how to identify the regions pertaining to some of the variables within the 5 variable K-Map. Finding all regions is left as an exercise.

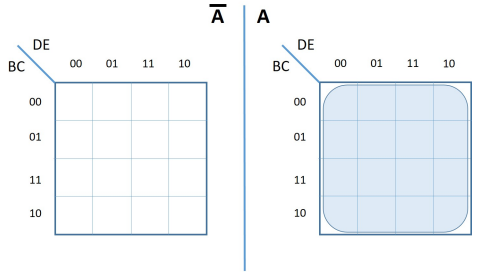


Figure 3. Region where A = 1

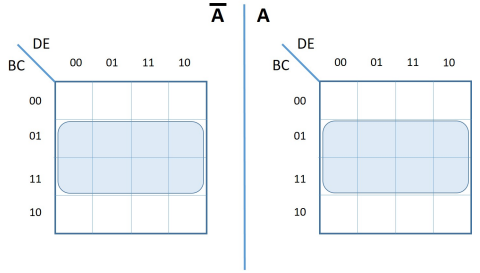


Figure 4. Regions where C = 1

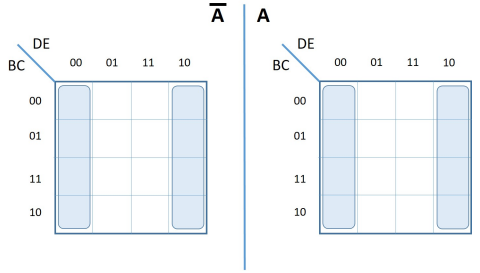


Figure 5. Regions where E = 0

Placing an Output on the 5 Variable K-Map

Knowing where all the regions are within the 5 variable K-Map, one can fill out the map based on a Truth Table (TT) with 32 possible outputs. Below, only one output is represented on the map. Assuming that f = 1 when A=0, B=1, C=1, D=0 and E=1, we place the output on the map as shown.

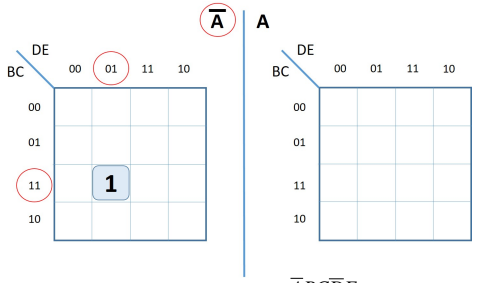


Figure 6. Output is 1 for $\overline{A}BC\overline{D}E$

Minimization Using the 5 Variable K-Map

Knowing all this, it is possible to visualize how to group the implicants on the 5 variable K-Map. One can imagine the 4 variable K-Map for $\overline{A}$ *hovering above*, or *overlapping with* the 4 variable K-Map for A, and then group the implicants appropriately. From this point on, the example follows an SOP implementation. Note that the designer should base a decision on minimizing *cost*, so the choice of a POS versus an SOP implementation is up to the designer.

Consider the TT below. The exercise is to find a minimum cost implementation using a 5 Variable K-Map. The same methodology to group implicants used for 4, 3 and 2 variable K-Maps applies here. With 5 variables, however, one has the chance to identify groups of 8 implicants as well (as opposed to a maximum of 4), as it will be shown. Should the Truth Table present "don't cares", it is up to the designer to use them as appropriate, in similar fashion to how they are used in for 4, 3 and 2 variable K-Maps.

A	B	C	D	E	f
0	0	0	0	0	1
0	0	0	0	1	1
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	1
0	0	1	0	1	1
0	0	1	1	0	0
0	0	1	1	1	0
0	1	0	0	0	1
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	0	1	0
0	1	1	1	0	0
0	1	1	1	1	0

A	B	C	D	E	f
1	0	0	0	0	1
1	0	0	0	1	1
1	0	0	1	0	0
1	0	0	1	1	0
1	0	1	0	0	1
1	0	1	0	1	1
1	0	1	1	0	1
1	0	1	1	1	0
1	1	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	0
1	1	1	0	1	0
1	1	1	1	0	1
1	1	1	1	1	1

The Truth Table presented above results in the following 5 variable K-Map. The groups of implicants are shown, highlighted in different colours. There are: one group of 8 implicants ($\overline{BD}$), two groups of 4 implicants ($\overline{CDE}$ and ABD) and one group of two implicants ($ACD\overline{E}$)

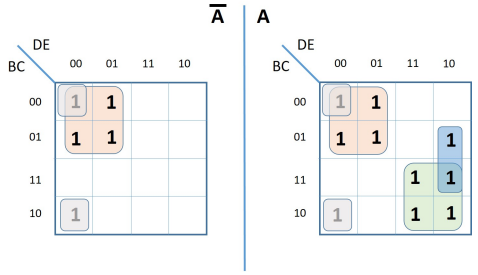


Figure 7. $f = \overline{BD} + \overline{CDE} + ABD + ACD\overline{E}$

Another way to visualize the interaction between the two halves of the 5 variable K-Map is to attempt to overlap them, as suggested earlier. This is shown in the picture below. One can see how to form the one group of 8 implicants and one of the groups of 4 implicants that span both halves "vertically", with the map for $\overline{A}$ overlapping the map of A.

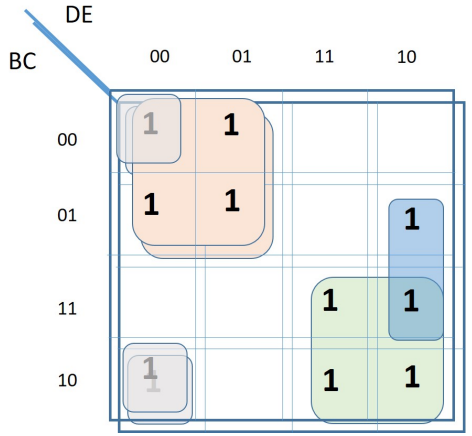


Figure 8. $f = \overline{BD} + \overline{CDE} + ABD + ACD\overline{E}$

This document was prepared based on I.V. Idoeta & F.G. Capuano, "Elementos de Eletrônica Digital", 40th Ed., Editora Érica, 2008, pp.130-140

SEQUENTIAL CIRCUITS

ECE241 (1) 2023

- SO FAR, WE HAVE LOOKED AT COMBINATIONAL CIRCUITS.
- UP TO NOW, THE OUTPUT IS ONLY A FUNCTION OF THE CURRENT INPUT
  - FOR EACH SET OF INPUTS, THERE IS ONE OUTPUT VALUE (TT)
- NOW WE WILL LOOK AT CIRCUITS IN WHICH THE OUTPUT DEPENDS ON THE CURRENT INPUT AND PAST VALUES OR HISTORY OF INPUTS.
  - ⇒ SUCH CIRCUITS HAVE "MEMORY" CALLED STATE
- THE OUTPUT IS A FUNCTION OF THE SEQUENCE WITH WHICH THE INPUT VALUES WERE APPLIED TO THE CIRCUIT
  - DIFFERENT SEQUENCES LEAD TO DIFFERENT RESULTS

OUTPUT = PRESENT INPUT + STATE

(STATE: CHARACTERIZED BY BINARY STATE VARIABLES)

- SINCE WE'RE TALKING ABOUT "REMEMBERING" PAST INPUT VALUES, WE ARE REALLY TALKING ABOUT MEMORY

(WE'LL SEE THAT MEMORY HERE IS RELATED TO FEEDBACK)

LET'S SEE AN EXAMPLE IN WHICH WE WISH TO DESIGN AN ALARM SYSTEM FOR THE BEDROOM DOOR THAT

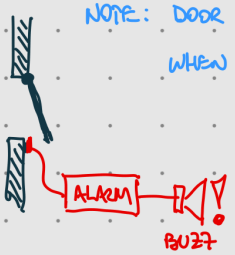
- ONCE ACTIVATED, DOES NOT TURN OFF WHEN THE DOOR IS CLOSED AGAIN.
  - i.e. IT "REMEMBERS" OR "STORES" THAT THE DOOR HAS BEEN OPEN BEFORE

ECE241 (2) 2023

NOTE: DOOR OPEN → BUZZ

WHEN THE DOOR IS CLOSED AGAIN → ALARM STILL ON

THAT IS: IT REMEMBERS THAT THE DOOR WAS ONCE OPEN



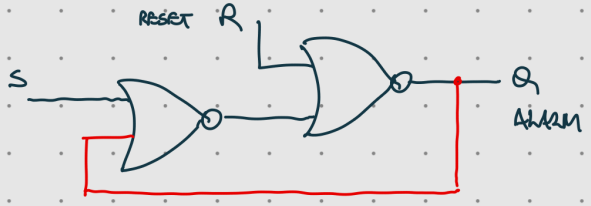
I NEED A SYSTEM IN WHICH THE OUTPUT OF "MEMORY" IS

- $ON/\overline{OFF} = 1$ — ALARM HIGH : SENSOR HIGH ⇒ SET
- $ON/\overline{OFF} = 0$ — ALARM OFF : WHEN THE ALARM IS HIGH AND RESET ARRIVES

NOTATION: ON → "ACTIVE ON HIGH"; $\overline{OFF}$ → "ACTIVE ON LOW"

OPERATION

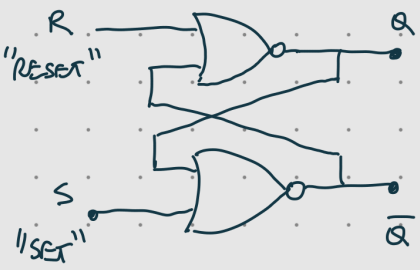
0. ASSUME THAT DOOR IS CLOSED
1. RESET ALARM TO A KNOWN STATE OF ALARM OFF
2. R=0 (FORCE R=0)
3. OPEN THE DOOR (SENSOR GOES HIGH, OUTPUT GOES HIGH)
   ⇒ S → 1, SO Q → 1 ⇒ ALARM SOUNDS!
4. CLOSE THE DOOR (SENSOR GOES LOW, OUTPUT STILL HIGH)
   LOOK @ CIRCUIT → DUE TO FEEDBACK
   ⇒ S → 0 BUT Q STAYS AT 1
5. Q STAYS AT 1 UNTIL RESET IS BROUGHT HIGH



THIS IS CALLED AN RS LATCH (OR SR - SET/RESET)

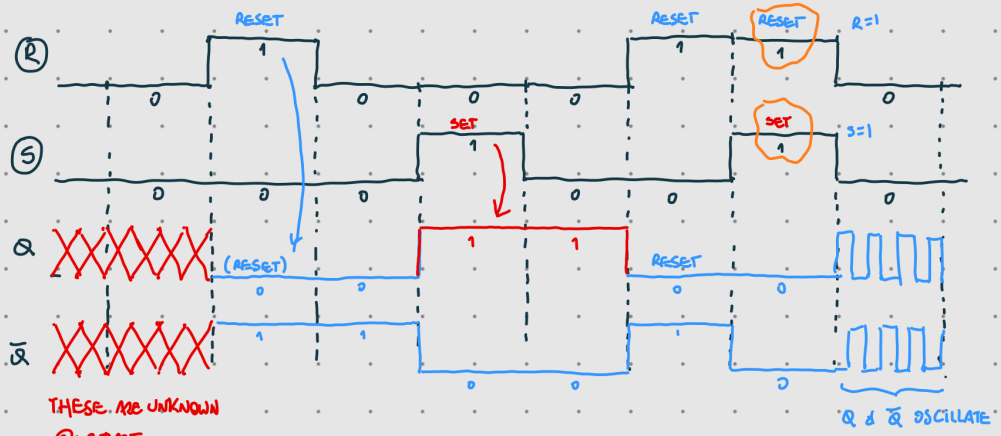
NOTE: FEEDBACK KEY IN MAINTAINING Q HIGH PRIOR TO RESET

⊕ RS LATCH - TWO CROSS COUPLED NOR GATES



"SET" Q TO 1
"RESET" Q TO 0

TIMING DIAGRAM



THESE ARE UNKNOWN @ START
IN REALITY → "SOME" VALUE
IN SIMULATION → UNKNOWN

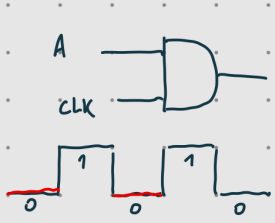
THE RS LATCH HAS THE FOLLOWING CHARACTERISTIC TABLE

S	R	Q(t+1)	$\overline{Q(t+1)}$	
0	0	Q(t)	$\overline{Q(t)}$	WHATEVER PREVIOUS VALUE
0	1	0	1	
0	0	0	1	HOLD
1	0	1	0	
0	0	1	0	HOLD
0	1	0	1	
1	1	0	0	DO NOT USE

⊛ GATED RS LATCH

- WE ADD ANOTHER INPUT TO THE RS LATCH SO THAT WE CAN USE A CLOCK TO "ENABLE" AND "DISABLE" THE LATCH
- THIS IS DONE WITH AN AND GATE

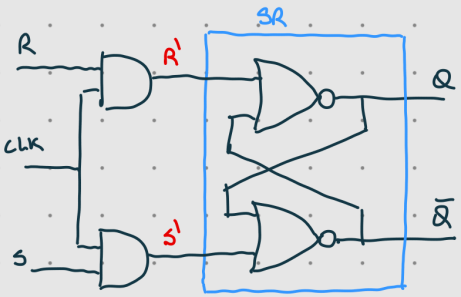
LET'S CONSIDER THE AND GATE BY ITSELF WITH ONE OF THE INPUTS ALTERNATING PERIODICALLY BETWEEN 0 & 1 (A CLOCK)



CLK	A	f
0	0	0
0	1	0
1	0	0
1	1	1

WHEN CLOCK IS ZERO, OUTPUT IS ZERO. WHEN CLOCK IS ONE INPUT VALUE IS NOT AFFECTED

OUTPUT FOLLOWS A



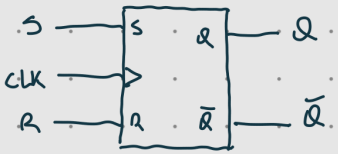
IF CLOCK IS ZERO, R' & S' ARE ZERO
IF R' & S' IS ZERO, Q IS NOT AFFECTED

S	R	Q	$\overline{Q}$
0	0	Q(t)	$\overline{Q(t)}$
0	1	0	1
1	0	1	0
1	1	DO NOT USE	

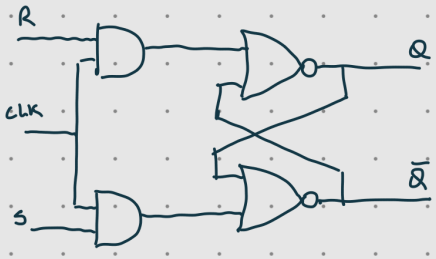
WE CAN DERIVE A CHARACTERISTIC TABLE FOR THIS CIRCUIT:

CLK	S	R	Q(t+1)
0	X	X	Q(t)
1	0	0	Q(t)
1	0	1	0
1	1	0	1
1	1	1	DON'T USE

SYMBOL

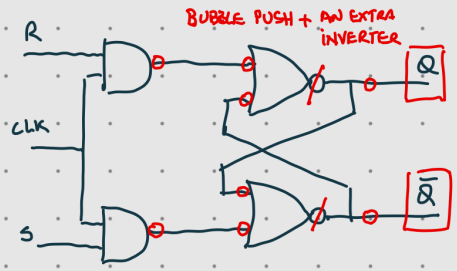


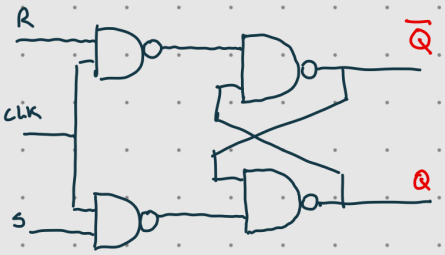
THE GATED SR LATCH CAN BE IMPLEMENTED WITH NAND GATES ONLY



RECALL

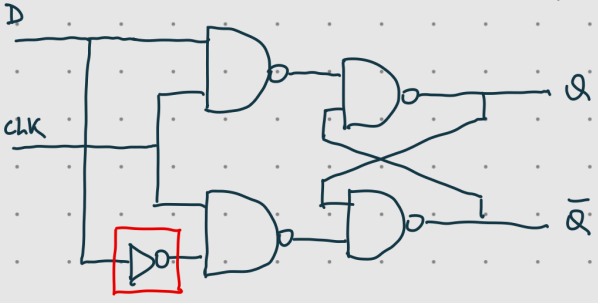
FOR THE INVERTER





⊗ GATED-D LATCH

- EASIER TO STORE A VALUE IF WE ONLY HAVE ONE INPUT
- WE KNOW BOTH INPUTS TO THE LATCH BEING 1 IS PROBLEMATIC
  - WE CAN AVOID THIS PROBLEM WITH A NOT GATE



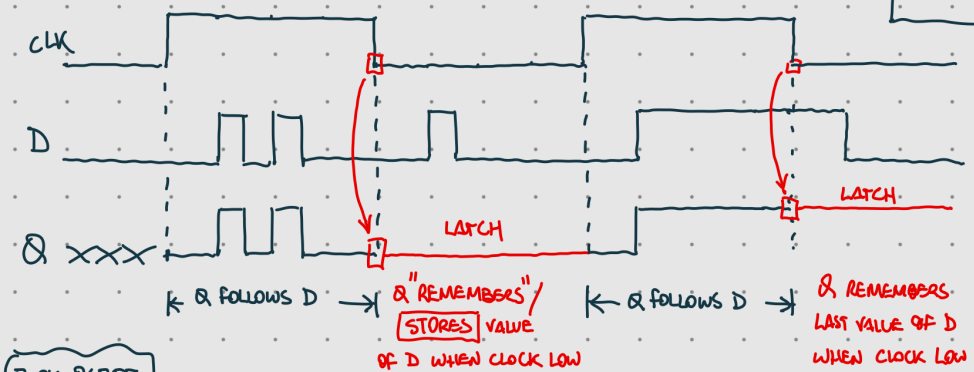
THIS PREVENTS S & R BEING 1 AT THE SAME TIME

CHARACTERISTIC TABLE

CLK	D	Q(t)	
0	X	Q(t)	STORED
1	0	0	"LOAD" VALUE AT D
1	1	1	

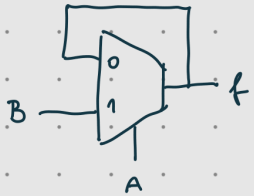
SYMBOL

TIMING DIAGRAM FOR GATED D LATCH { CLOCK HIGH, Q FOLLOWS D; CLOCK LOW, Q REMEMBERS LAST D



BOOK PG255

CONSIDER ALSO THE FOLLOWING MUX-BASED CIRCUIT



DON'T CARE!

A	B	f(t+1)	
0	0	f(t)	LATCHES TO LAST VALUE
0	1	f(t)	
1	0	0	FOLLOWS INPUT
1	1	1	

THIS ALSO BEHAVES AS A D LATCH

MIDTERM UP TO HERE

⊙ FLIP-FLOPS

ECE241 ① 2023

BOOK pg 258

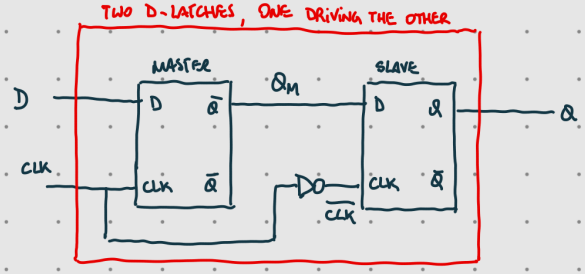
A ff IS A STORAGE ELEMENT THAT CHANGES ITS OUTPUT STATE AT THE EDGE OF A CONTROLLING (CLOCK) SIGNAL

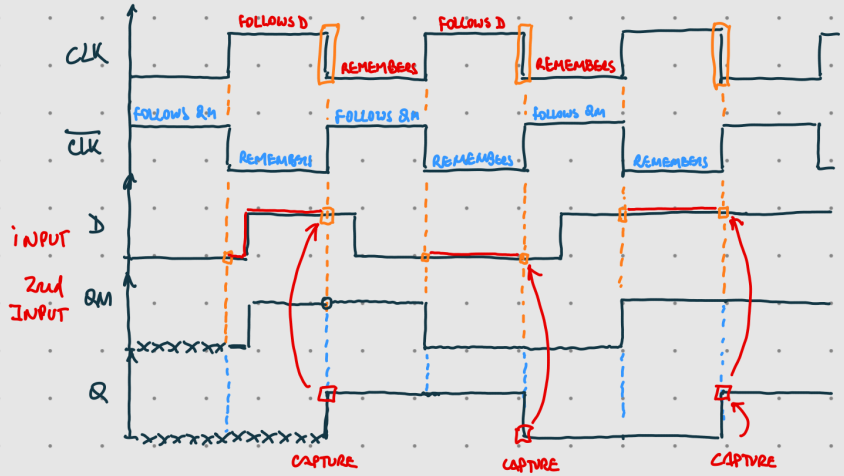
EDGE SENSITIVE

REMEMBER: GATED LATCHES ARE LEVEL-SENSITIVE { "FOLLOWS D WHEN CLOCK HIGH" / "LATCHES WHEN CLOCK LOW" }

⊙ PRIMARY-SECONDARY D FLIP-FLOP ("MASTER-SLAVE")

pg 257 Book



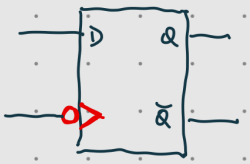


NOTE: THE CIRCUIT CHANGES STATE AT THE NEGATIVE EDGE (↓) OF THE CLOCK

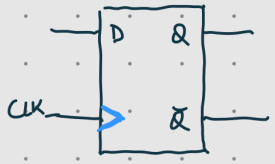
IT IS SAID: "IT CAPTURES D" @ NEGATIVE EDGE OF CLOCK

ECE241 ② 2023

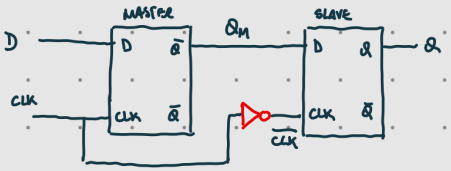
WE CAN HAVE D ff SENSITIVE TO POSITIVE OR NEGATIVE EDGE OF CLOCK

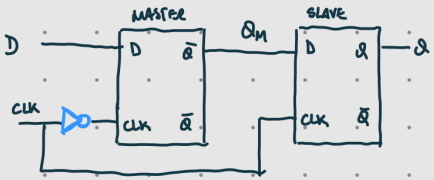


NEGATIVE EDGE SENSITIVE



POSITIVE EDGE SENSITIVE

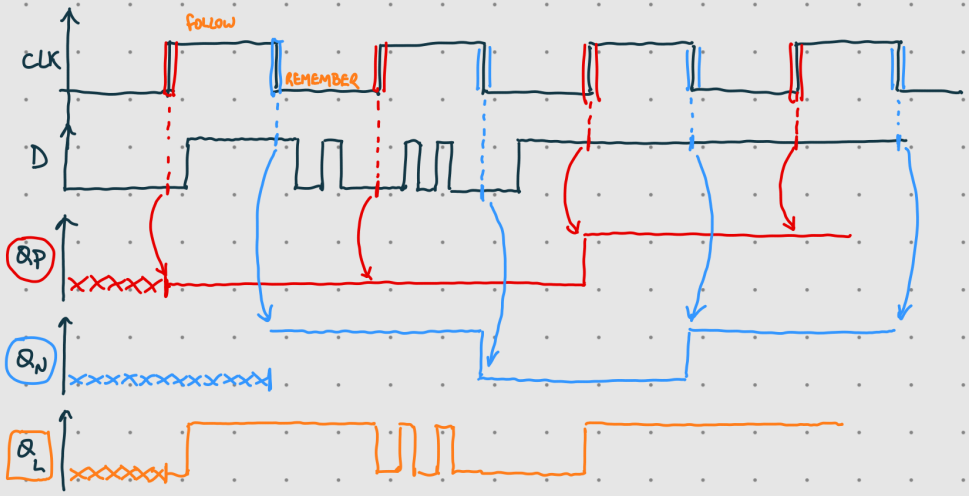




D ff "CAPTURES" D ON EDGE OF CLOCK, WHEREAS

D LATCH FOLLOWS D @ CLOCK = 1 & "REMEMBERS" @ CLOCK = 0

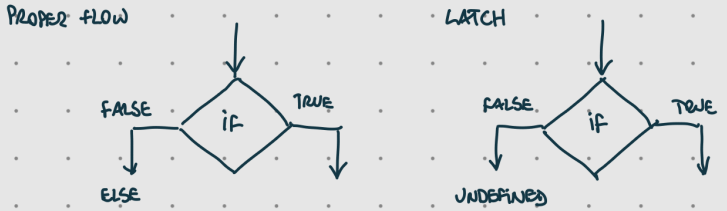
⊙ LET'S COMPARE DFF POS EDGE (Q_P), DFF NEG EDGE (Q_N), D LATCH (Q_L)



⊛ VERILOG for D LATCH — WHAT NOT TO DO

```
module Dlatch (input D, clk, output Q);
    reg Q;
    always @ (*)
        if (clk == 1)
            Q = D;
endmodule
```

NOTE: ONE WOULD THINK THAT SINCE WHEN CLK == 0 WE KEEP THE OLD VALUE (OR "REMEMBER", OR "LATCH"), THEN WE WOULD NOT NEED AN ELSE. THIS IS A PROBLEM.



WE WILL AVOID THE USE OF LATCHES, AS WE SHOULD NOT "INFER" AN ELSE

⊛ VERILOG for FLIP-FLOPS (EDGE-TRIGGERED)

```
module DFF (input D, clock, output Q);
    reg Q;
    always @ (posedge clock)
        Q <= D;
endmodule
```

THAT IS: CAPTURE D (input) @ THE POSITIVE EDGE OF THE CLOCK

THIS IS AN ASSIGNMENT USED FOR EDGED-TRIGGERED MEMORY (CALLED "NON-BLOCKING")

VERILOG: BLOCKING vs NON-BLOCKING ASSIGNMENTS

= ⇐

⊙ BLOCKING

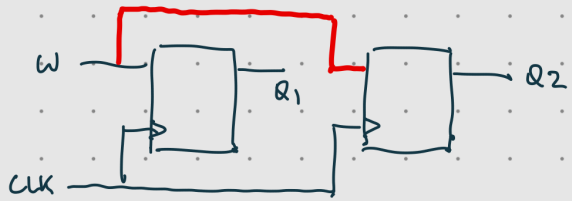
- COMPLETES & UPDATES LHS BEFORE NEXT ASSIGNMENT
- STATEMENTS EVALUATED IN THE ORDER THAT THEY ARE WRITTEN
- USED IN COMBINATIONAL CIRCUITS (NO CLOCK)

EXAMPLE: IF WE DO

```
reg Q1, Q2
always @ (posedge clk)
begin
    Q1 = w;
    Q2 = Q1;
end
```

FIRST Q1 IS ASSIGNED THE VALUE OF w

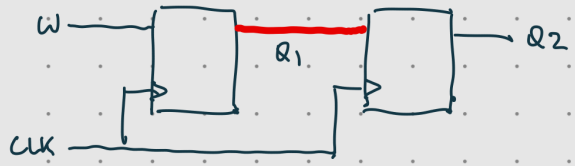
WHAT WE ARE ACTUALLY DOING IS



⊙ NON BLOCKING (<=)

- VALUES ARE UPDATED AT THE END OF THE ALWAYS BLOCK
- STATEMENTS ARE EVALUATED CONCURRENTLY
- IT IS USED IN SEQUENTIAL CIRCUITS (WHERE WE HAVE A CLOCK)

```
reg Q1, Q2;
always @ (posedge clock)
begin
    Q1 <= w;
    Q2 <= Q1;
end
```



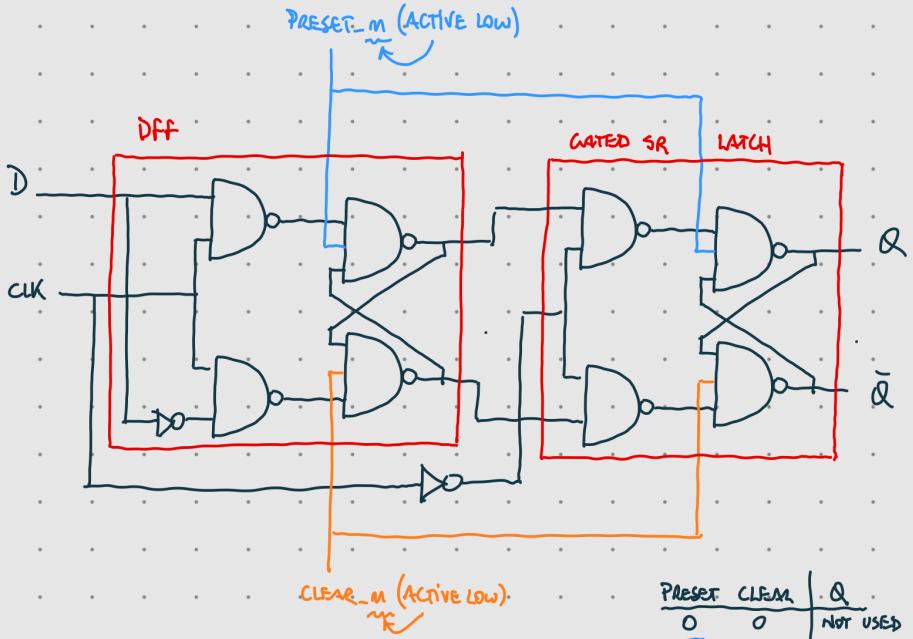
Q2 IS ASSIGNED THE VALUE OF Q1 @ THE START OF THE ALWAYS BLOCK

THEREFORE, BEFORE POSEDGE HAPPENS (BEFORE THE CLOCK EDGE)

(reset)

⊙ D ff WITH CLEAR & PRESET

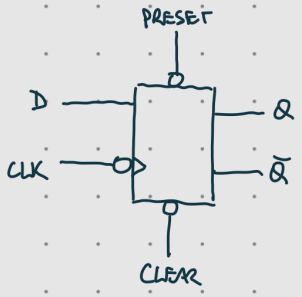
- USED WHEN THERE IS A NEED TO RETURN TO OR GO TO A KNOWN STATE
- FOR INSTANCE, WHEN WE COUNT, WE WANT TO START AT A KNOWN POINT
- IT IS MADE OF ONE DFF (PRIMARY/MASTER) AND AN SR LATCH (SECONDARY/SLAVE)



PRESET	CLEAR	Q
0	0	NOT USED
0	1	1
1	0	0
1	1	NO CHANGE

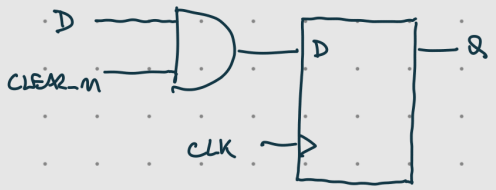
IF PRESET = 0, Q WILL ALWAYS BE 1

IF CLEAR = 0, Q̄ WILL ALWAYS BE 1 ⇒ Q=0



- BOTH ARE ACTIVE LOW (INDICATED BY BUBBLE & SUBSCRIPT m)
- NOTE THAT THIS IS ASYNCHRONOUS (GREEK FOR "NOT WITH TIME")
- NOTE THAT BOTH CLEAR & PRESET ARE OVERRIDING INPUTS
  THAT IS, THEY WILL CHANGE THE OUTPUT NO MATTER WHAT THE OTHER INPUTS ARE

IF WE DESIRE TO MAKE CLEAR OR RESET SYNCHRONOUS (CLOCK DEPENDENT) WE USE AN AND GATE WITH THE INPUT



WHEN CLEAR IS ZERO, IT FORCES A ZERO @ INPUT OF ff

⇒ Q=0 @ EDGE OF CLOCK

⊙ VERILOG

- SYNCHRONOUS RESET (ON CLOCK EDGE)
  ↑ "CLEAR"

```
module DFFR (input D, reset_m, clock, output Q);
    reg Q;
    always @ (posedge clock)
        if (!reset_m)
            Q <= 0;
        else
            Q <= D;
endmodule
```

SENSITIVE TO POSEDGE OF CLOCK

p.690 LOGICAL NOT (~ is BITWISE)

- NOTE THAT RESET IS ON NEGATIVE (0), SO "NOT RESET_m" MEANS WHEN RESET_m is ONE, "NOT RESET_m" RESULTS IN ZERO
- Q CAN ONLY CHANGE @ THE EDGE OF THE CLOCK, SO RESET_m IS ONLY EFFECTIVE "ON CLOCK" (SYNCHRONOUS)

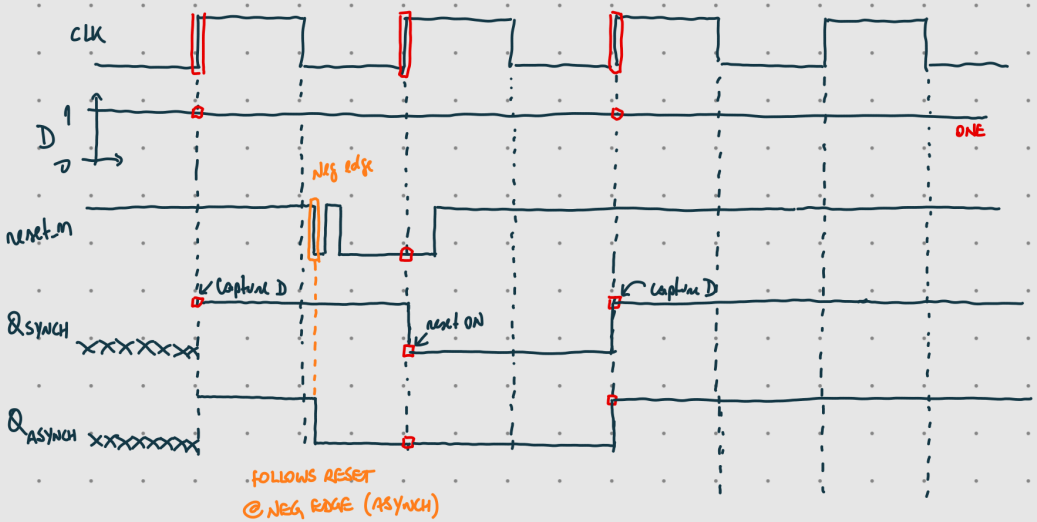
- ASYNCHRONOUS RESET – SENSITIVITY LIST INCLUDES RESET, IN ADDITION TO POSEDGE

```
module DFFR_asynch (input D, reset_n, clock, output Q);
    reg Q;
    always @ (posedge clock, negedge reset_n)
        if (!reset_n)
            Q <= 0;
        else
            Q <= D;
endmodule
```

SENSITIVE TO POSEDGE OF CLOCK AS WELL AS NEGEDGE OF RESET

– NOTE THAT Q WILL CHANGE WHEN RESET_n CHANGES (AT ITS NEGATIVE EDGE)

TIMING DIAGRAM



SYNCH – CLOCK POSEDGE HAPPENS – IS RESET "NOT TRUE"? IF YES – Q LOW TIL NEXT CLOCK POSEDGE

ASYNCH – CLOCK POSEDGE OR RESET_n NEGEDGE HAPPEN – IS RESET_n "NOT TRUE"? YES – Q LOW

LOOK @ 4.11.1 – BROWN/VRANESIĆ – VERILOG

– IF YOU DON'T SIMULATE YOUR DESIGN, IT WILL NOT WORK

IF YOU DO SIMULATE IT, IT MIGHT WORK

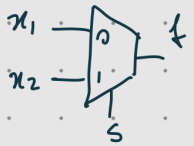
– READ THE WARNINGS

– CAREFUL WITH "INITIAL" STATEMENT – IT IS FOR SIMULATION ONLY AND IT IS NOT SYNTHESIZABLE

- USE ALWAYS BLOCK

– ALWAYS DRAW THE CIRCUIT, THEN WRITE/DESCRIBE IT IN CODE

- MUX USING PROCEDURAL IF (MUST BE IN THE ALWAYS BLOCK)




```
module MUX_b (input x1, x2, s, output f);
    reg f;
    always @ (x1, x2, s)
        begin
            if (!s)
                f = x1;
            else
                f = x2;
        end
endmodule
```

before

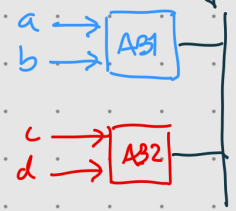
```
always@(*)
  begin
    m = s&x2 | ~s&x1;
  end
```

- PROBLEM WITH MULTIPLE ALWAYS BLOCKS

```
module M (...);
    wire a, b, c, d;
    reg f;

    always @ (*)
        f = a & b;
    always @ (*)
        f = c | d;
endmodule
```

AB1, AB2



THIS WILL CAUSE AN ERROR

– MODELSIM WILL RUN,

– QUARTUS WILL PRODUCE AN ERROR

ECE241
(3)
2023

- MULTIPLE ASSIGNMENTS IN ALWAYS BLOCK

```
module N (...);
    wire a, b, c, d;
    reg f;
    always @ (*)
        f = a & b;
        f = c | d;
endmodule
```

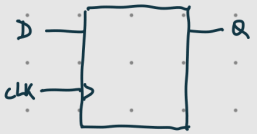
procedure

MODELSIM WILL WORK BUT THE OUTPUT IS NOT WHAT YOU WANT

f TAKES VALUE OF LAST ASSIGNMENT

- SOME DETAILS ON TIMING

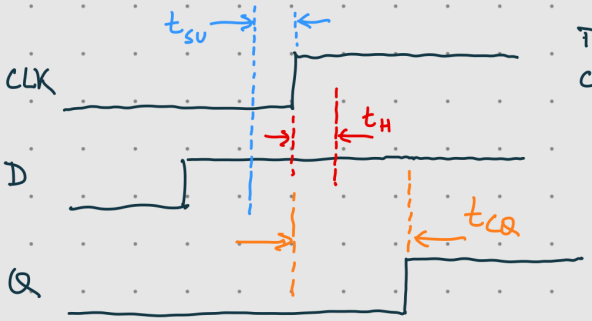
SAY WE HAVE



POS EDGE D ff

⇒ @ posedge, D is captured

IN REALITY
- THE LOGIC VALUE OF D MUST BE TABLE DURING SOME TIME PRIOR TO CLOCK EDGE, CALLED SET UP TIME (t_{su})
- THE LOGIC VALUE OF D MUST REMAIN STABLE DURING SOME TIME BEYOND THE EDGE OF THE CLOCK, CALLED HOLD TIME (t_H)



TIME WINDOW BETWEEN CLOCK EDGE & OUTPUT CALLED CLOCK-TO-Q t_{CQ}

MANUFACTURERS PROVIDE BENCHMARK VALUES