CS/COE0447: Computer Organization and Assembly Language

Logic Design Introduction (Brief?)

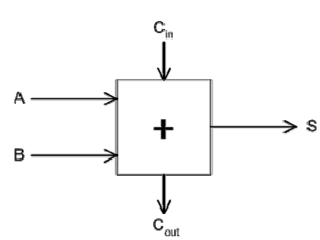
Appendix C: The Basics of Logic Design

modified by Bruce Childers
Sangyeun Cho

Dept. of Computer Science
University of Pittsburgh

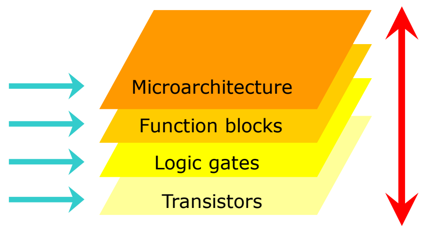
Logic design?

- Digital hardware is implemented by way of *logic design*
- Digital circuits process and produce two discrete values: 0 and 1
- Example: 1-bit full adder (FA)



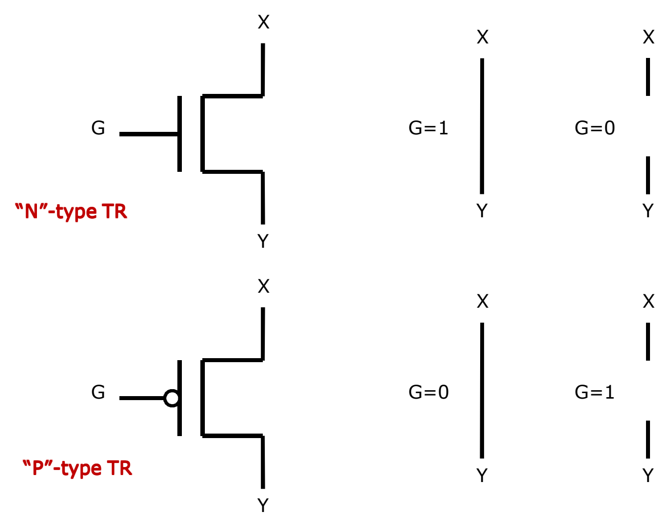
Layered design approach

- Logic design is done using **logic gates**
- Often we design a desired function using high-level languages and somewhat higher level than logic gates
- Two approaches in design
  - Top down
  - Bottom up

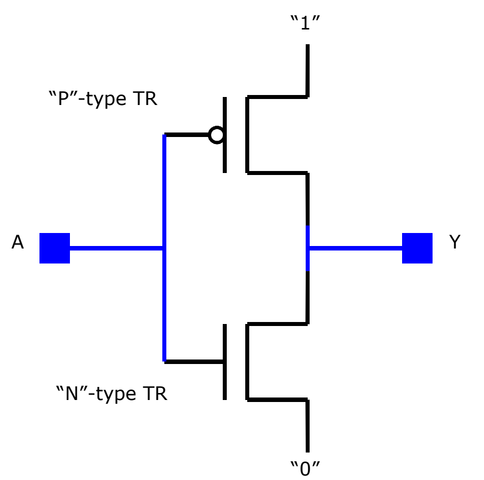


We'll do logic bottom up

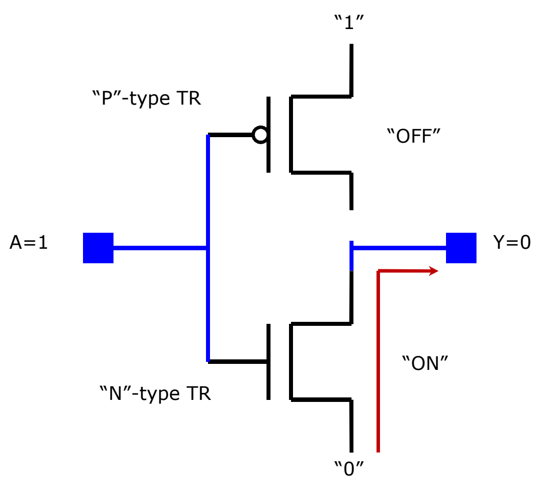
Transistor as a switch



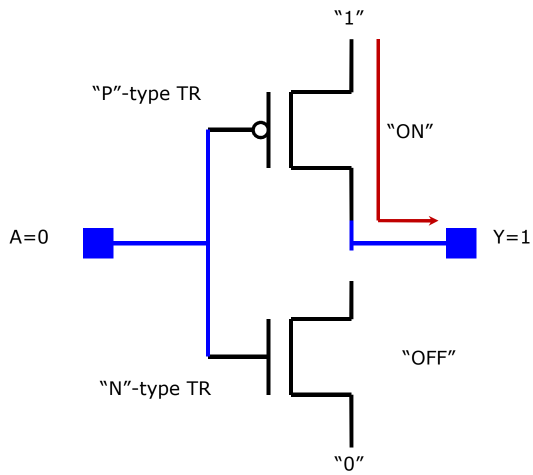
An inverter



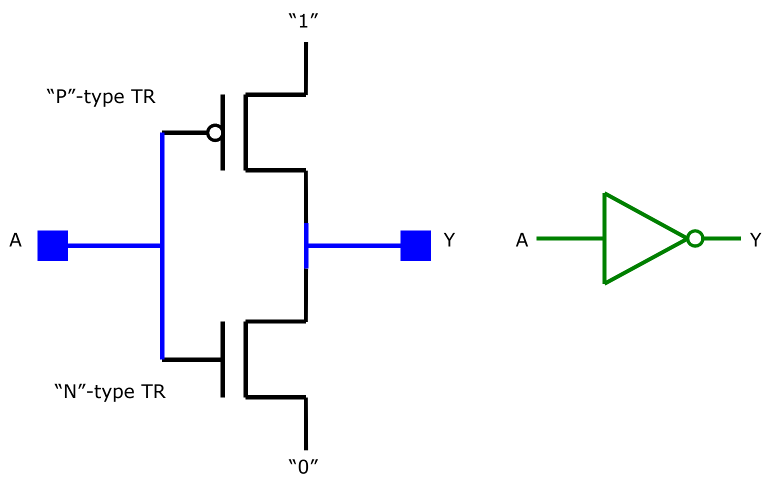
When A = 1



When A = 0

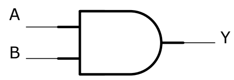


Abstraction



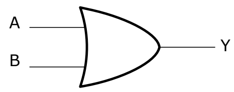
Logic gates

2-input AND



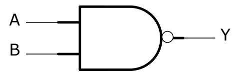
Y=A & B

2-input OR



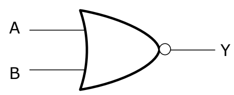
Y=A | B

2-input NAND



Y=~(A & B)

2-input NOR



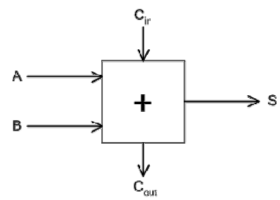
Y=~(A | B)

Describing a function

- $\text{Output}_A = F(\text{Input}_0, \text{Input}_1, \ldots, \text{Input}_{N-1})$
- $\text{Output}_B = F'(\text{Input}_0, \text{Input}_1, \ldots, \text{Input}_{N-1})$
- $\text{Output}_C = F''(\text{Input}_0, \text{Input}_1, \ldots, \text{Input}_{N-1})$
- …

- Methods
  - Truth table
  - Sum of products
  - Product of sums

Truth table



Input			Output	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	$\boxed{1}$	0
0	1	0	$\boxed{1}$	0
0	1	1	0	$\boxed{1}$
1	0	0	$\boxed{1}$	0
1	0	1	0	$\boxed{1}$
1	1	0	0	$\boxed{1}$
1	1	1	$\boxed{1}$	$\boxed{1}$

Sum of products

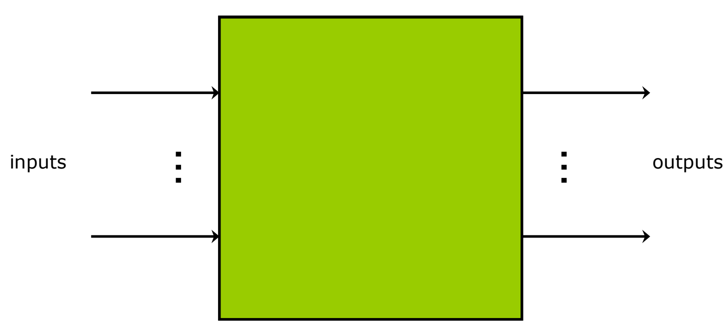
Input			Output	
A	B	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	$\boxed{1}$	0
0	1	0	$\boxed{1}$	0
0	1	1	0	$\boxed{1}$
1	0	0	$\boxed{1}$	0
1	0	1	0	$\boxed{1}$
1	1	0	0	$\boxed{1}$
1	1	1	$\boxed{1}$	$\boxed{1}$

- $S = A'B'C_{in} + A'BC_{in}' + AB'C_{in}' + ABC_{in}$
- $C_{out} = A'BC_{in} + AB'C_{in} + ABC_{in}' + ABC_{in}$

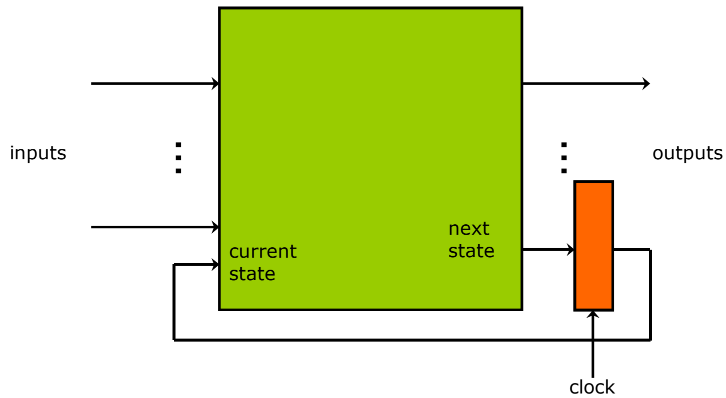
Combinational vs. sequential logic

- Combinational logic = **function**
  - A function whose outputs are dependent only on the current inputs
  - As soon as inputs are known, outputs can be determined
- Sequential logic = **combinational logic + memory**
  - Some memory elements (i.e., "state")
  - Outputs are dependent on the current state and the current inputs
  - Next state is dependent on the current state and the current inputs

Combinational logic



Sequential logic



Combinational logic

- Any combinational logic can be implemented using sum of products or product of sums
- Input-output relationship can be defined in the truth table format
- From the truth table, each output function can be derived
- Boolean expressions can be further manipulated (e.g., to reduce cost) using various Boolean algebraic rules

Boolean algebra

- Boole, George (1815~1864): mathematician and philosopher; inventor of Boolean Algebra, the basis of all computer arithmetic

- Binary values: {0,1}
- Two binary operations: AND (×/·), OR (+)
- One unary operation: NOT (~)

Boolean algebra

- Binary operations: AND (×/·), OR (+)
  - Idempotent
    - a·a = a+a = a
  - Commutative
    - a·b = b·a
    - a+b = b+a
  - Associative
    - a·(b·c) = (a·b)·c
    - a+(b+c) = (a+b)+c
  - Distributive
    - a·(b+c) = a·b + a·c
    - a+(b·c) = (a+b)·(a+c)

Boolean algebra

- De Morgan's laws
  - ~(a·b) = ~a + ~ b
  - ~(a+b) = ~a·~b
- More…
  - a+(a·b) = a
  - a·(a+b) = a
  - ~~a = a
  - a+~a = 1
  - a·(~a) = 0

It is not true I ate the sandwich and the soup.

same as:

I didn't eat the sandwich or I didn't eat the soup.

It is not true that I went to the store or the library.

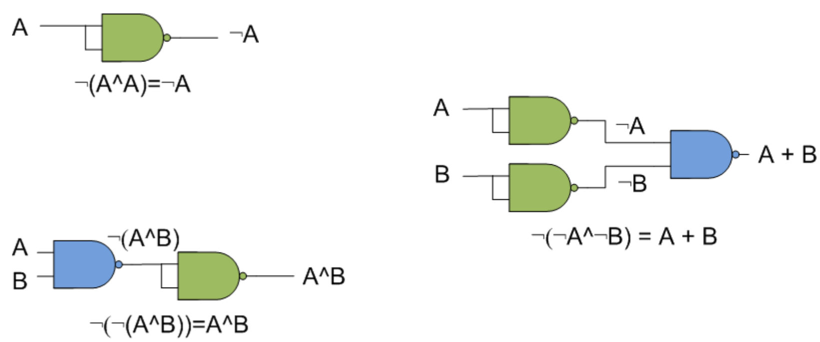
same as:

I didn't go to the store and I didn't go to the library.

Expressive power

- With AND/OR/NOT, we can express any function in Boolean algebra
  - Sum (+) of products (·)
- What if we have NAND/NOR/NOT?
- What if we have NAND only?
- What if we have NOR only?

Using NAND only



Using NOR only (your turn)

- Can you do it?
- NOR is ¬(A + B)

<u>NOT</u>

= ¬(A + A)
= ¬A ^ ¬A
= ¬A

<u>AND</u>

= ¬(¬(A + A) + ¬(B + B))
= ¬ (¬A ^ ¬A + ¬B ^ ¬B)
= ¬(¬A + ¬B)
= ¬(¬A) ^ ¬(¬B)
= A ^ B

<u>OR</u>

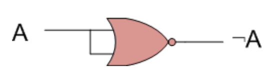
= ¬(¬(A + B) + ¬(A + B))
= (A + B) ^ (A + B)
= A + B

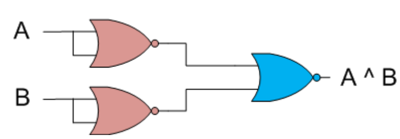
Using NOR only (your turn)

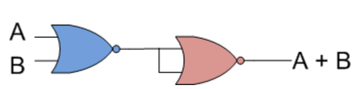
- Can you do it?
- NOR is ¬(A + B)
  - I.e., We need to write NOT, AND, and OR in terms of NOR

<u>NOT</u>	<u>AND</u>	<u>OR</u>
= ¬A	= A ^ B	= A + B
= ¬A ^ ¬A	= ¬(¬A) ^ ¬(¬B)	= (A + B) ^ (A + B)
= ¬(A + A)	= ¬(¬A + ¬B)	= ¬(¬(A + B) + ¬(A + B))
	= ¬ (¬A ^ ¬A + ¬B ^ ¬B)	
	= ¬(¬(A + A) + ¬(B + B))	

Using NOR only (your turn)





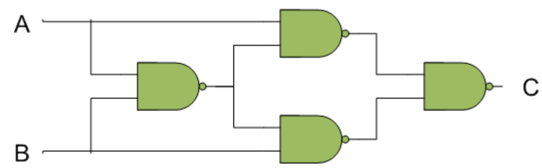


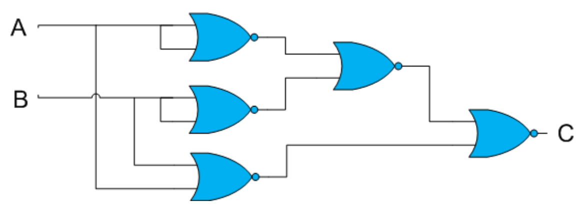
Now, it's really your turn….

- How about XOR?

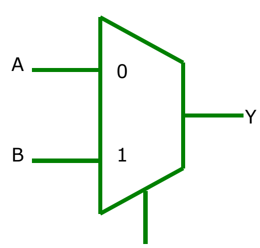
A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

C = A'B + AB'





Multiplexor (aka MUX)

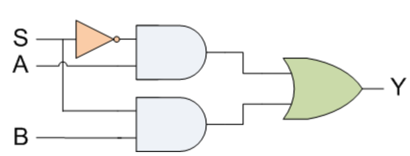


S	A	B	Y
0	0	x	0
0	1	x	1
1	x	0	0
1	x	1	1

Y=S'A+SB

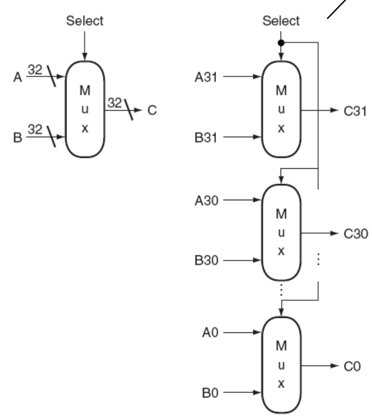
Y = (S) ? B:A;

when S =
0: output A
1: output B



A 32-bit MUX

Use 32 1-bit muxes
Each mux selects 1 bit
S is connected to each mux



a. A 32-bit wide 2-to-1 multiplexor

b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors