# CS/COE0447: Computer Organization and Assembly Language

## Logic Design Introduction (Brief?)

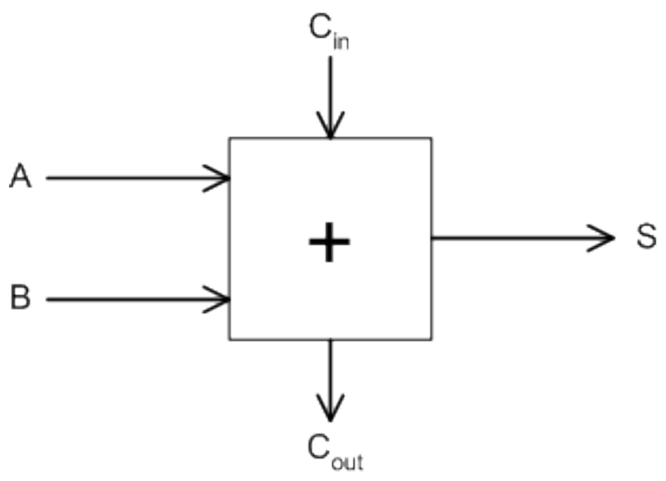
## Appendix B: The Basics of Logic Design

**Dept. of Computer Science**
**University of Pittsburgh**

## Logic design?

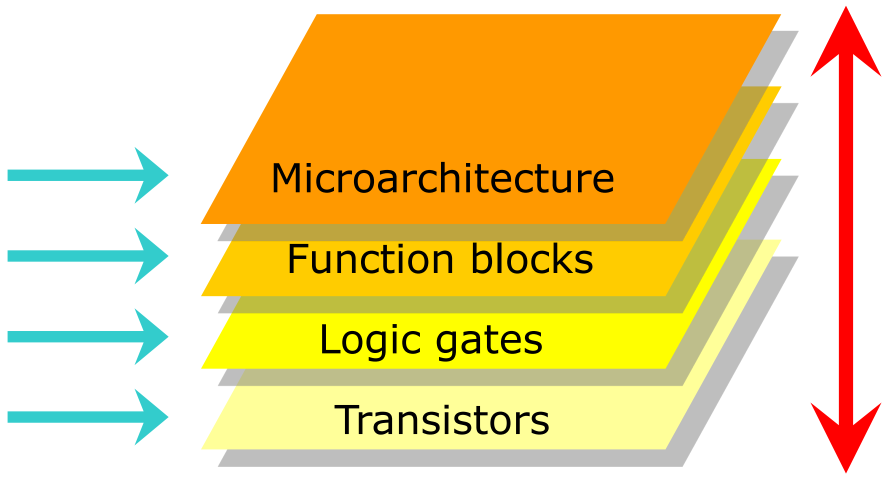
- Digital hardware is implemented by way of *logic design*
- Digital circuits process and produce two discrete values: 0 and 1

- Example: 1-bit full adder (FA)



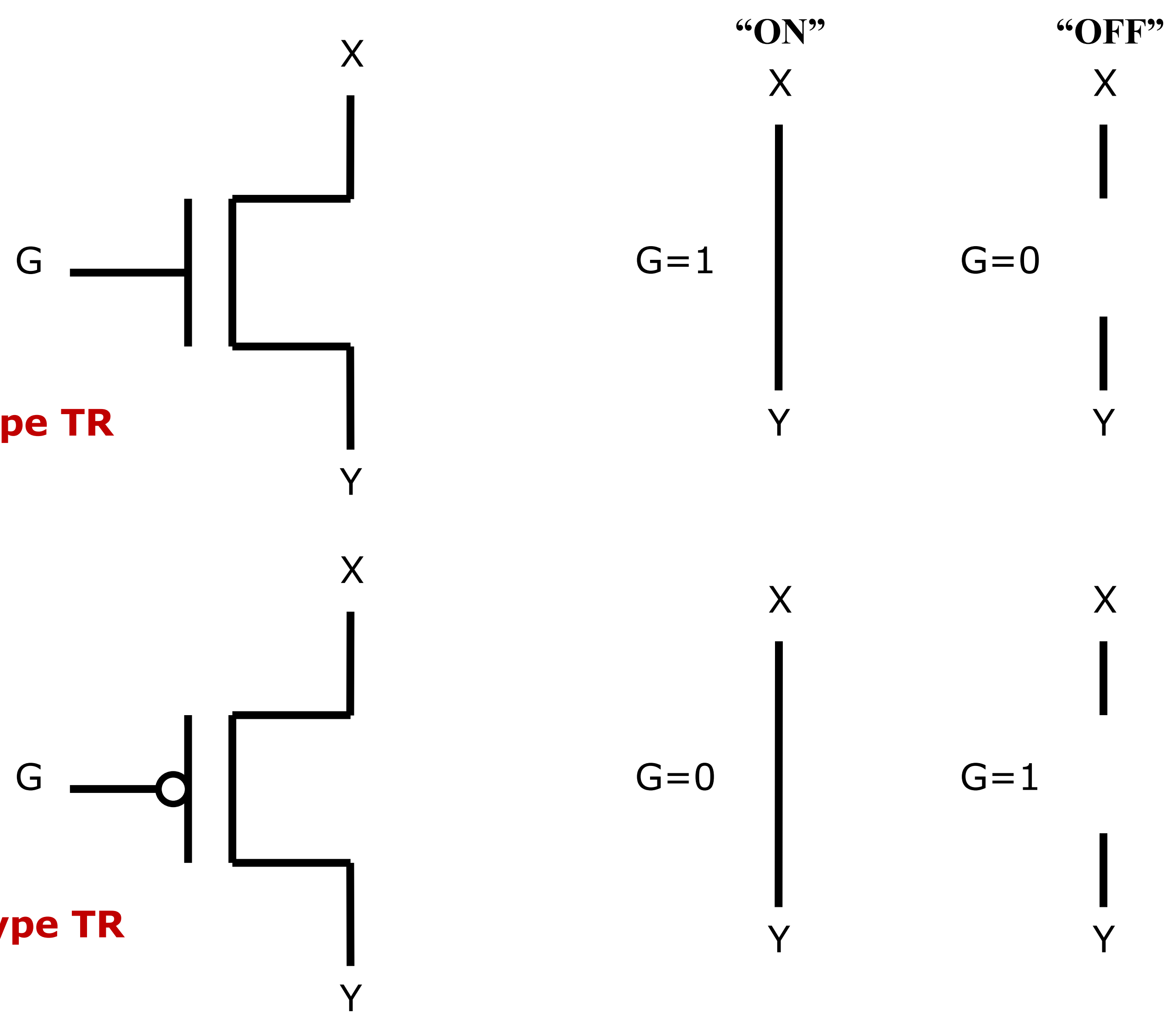
# Layered design approach

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

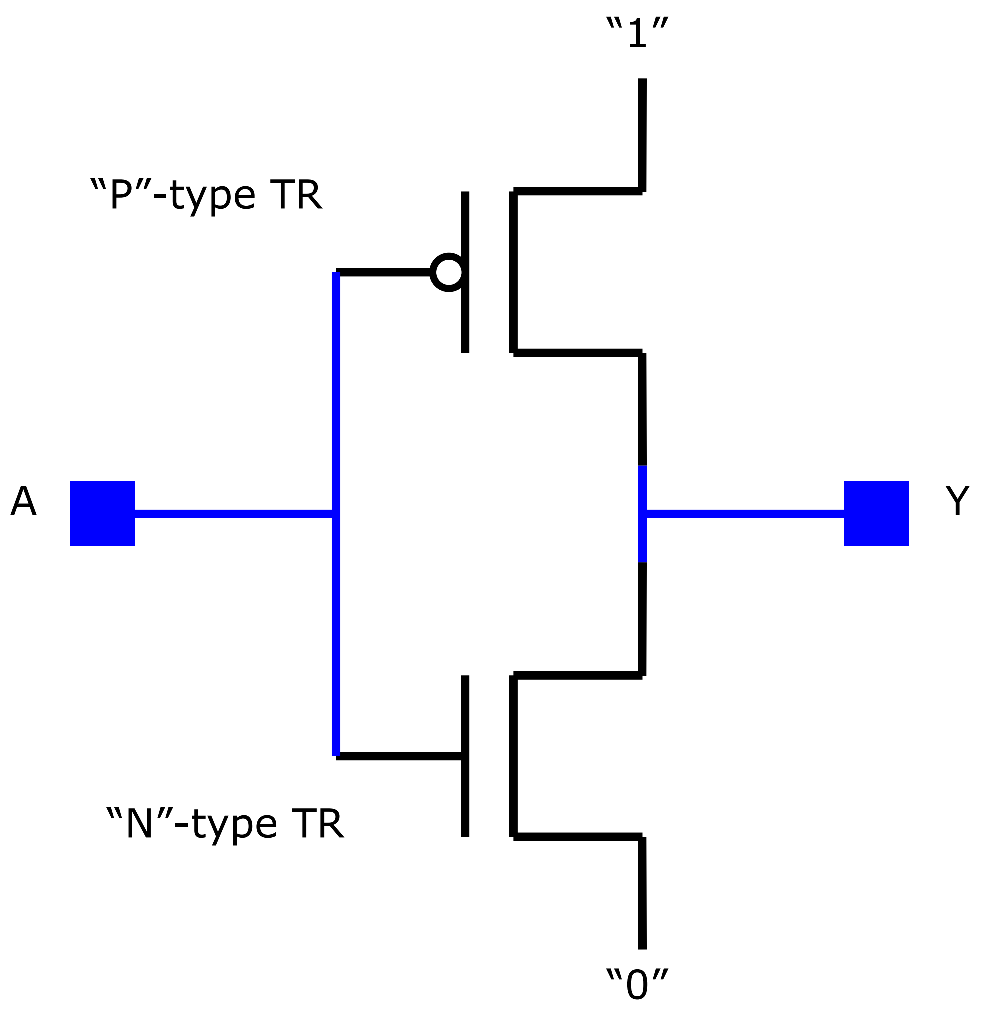


***We'll do logic bottom up***

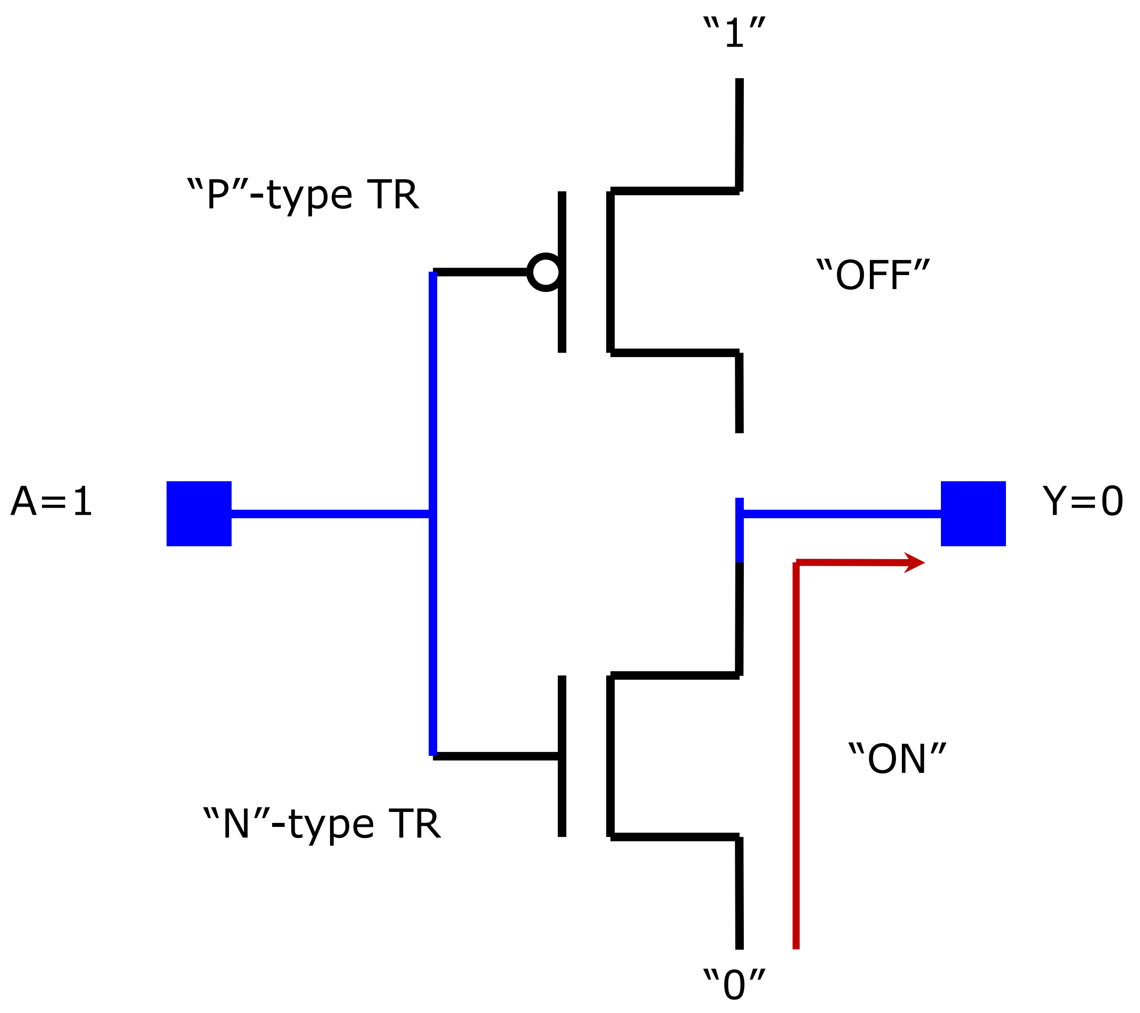
# Transistor as a switch



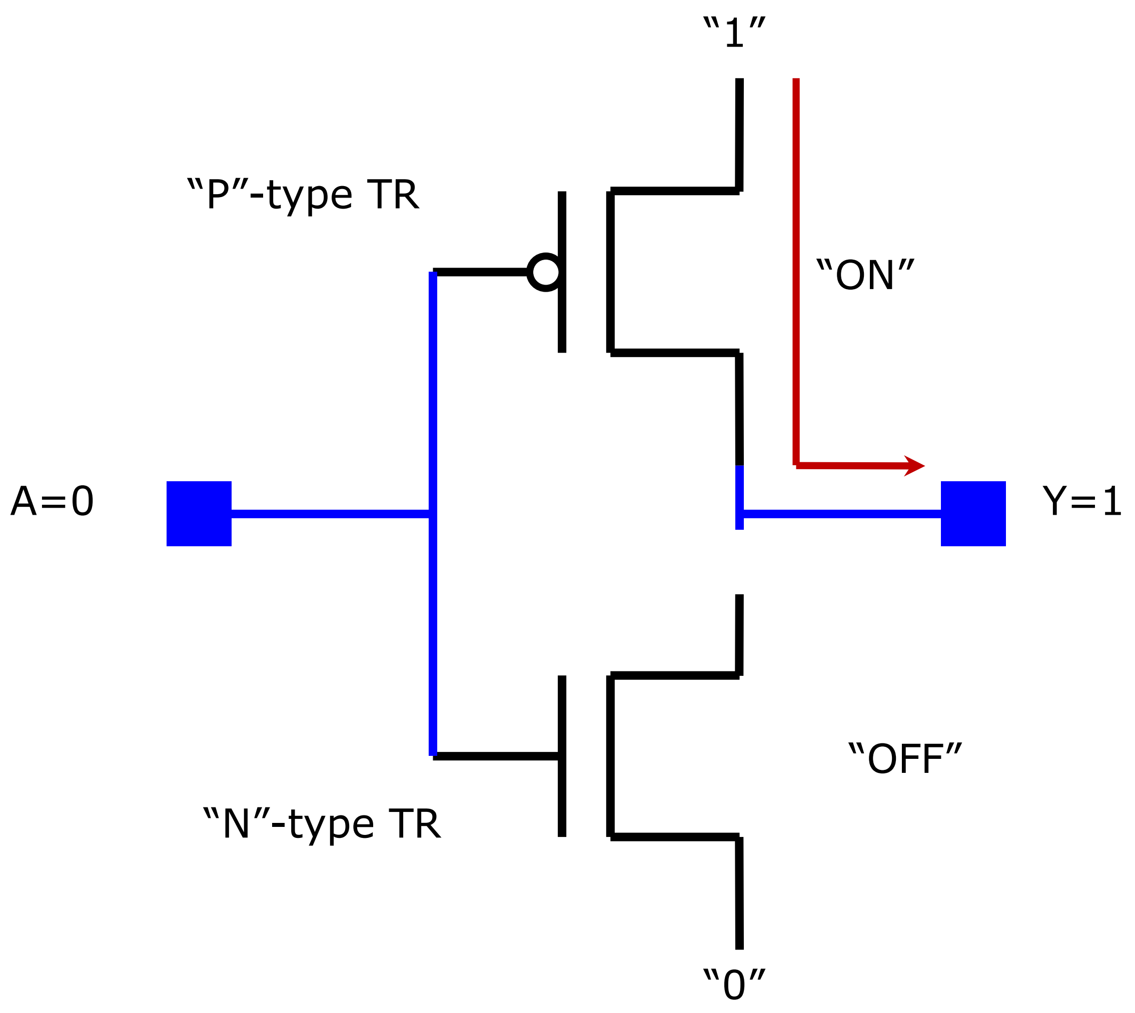
# An inverter



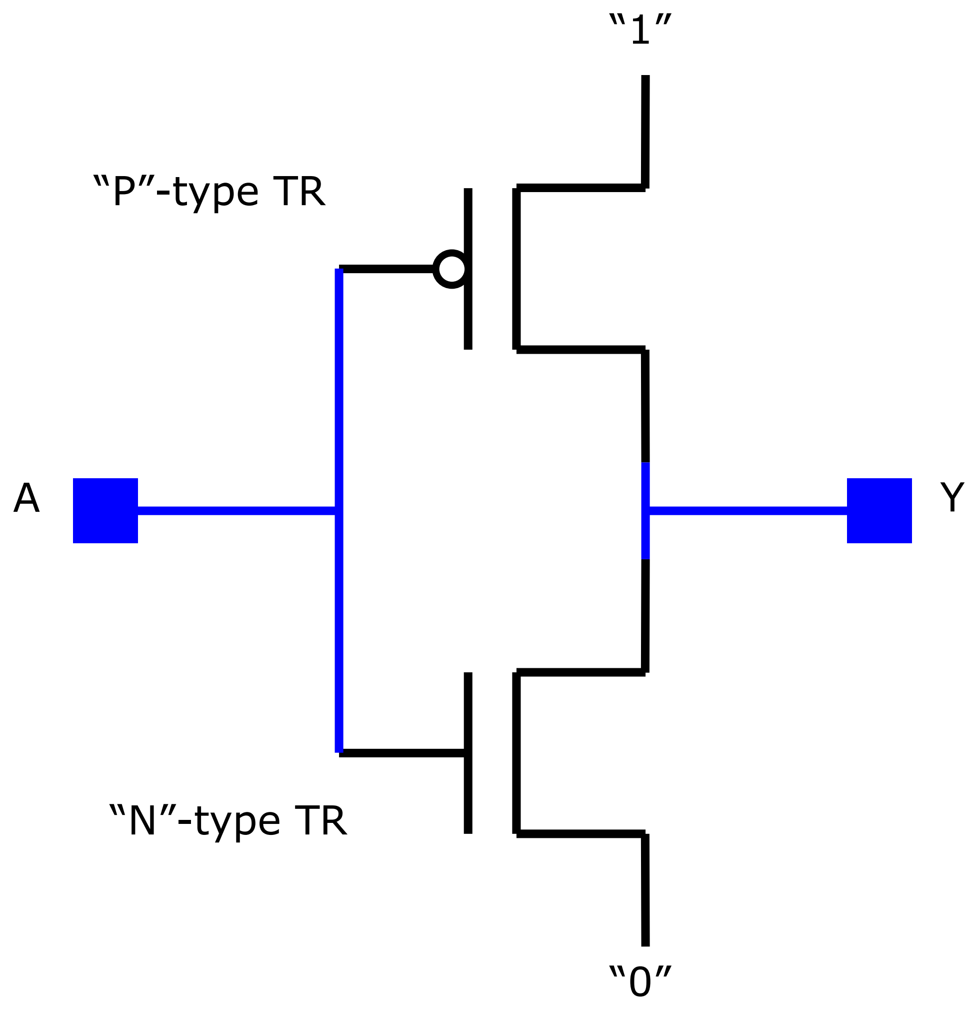
## When A = 1

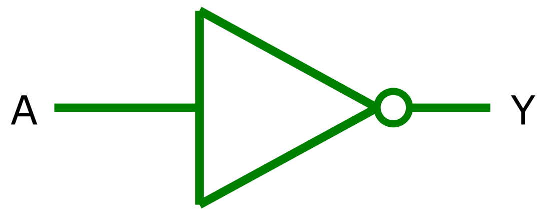


## When A = 0



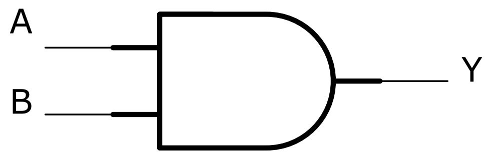
## Abstraction





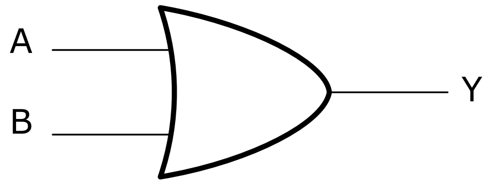
# Logic gates

2-input AND



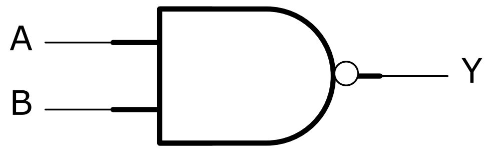
Y=A & B

2-input OR



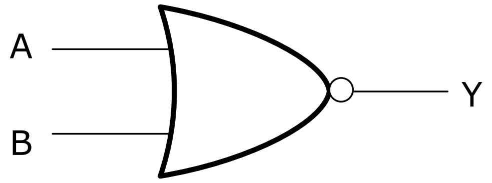
Y=A | B

2-input NAND



Y=~(A & B)

2-input NOR



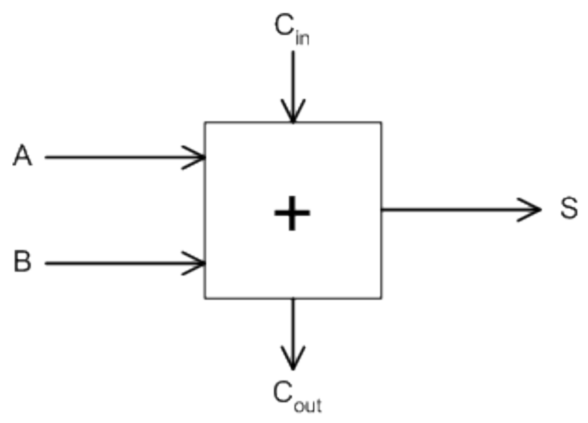
Y=~(A | B)

## 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 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 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 | $\boxed{1}$ | $\boxed{1}$ |

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

OR two minterms

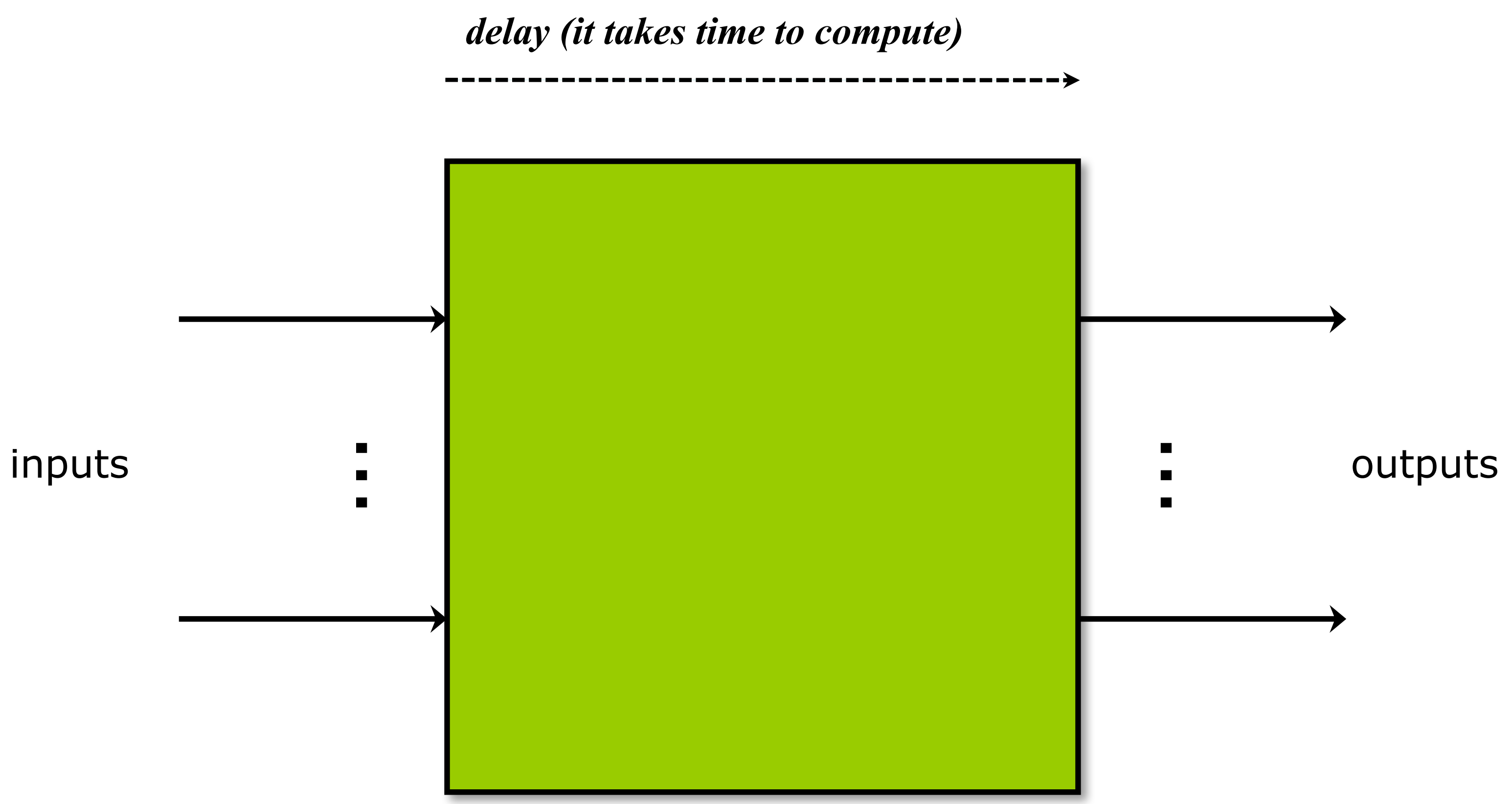
"Minterm"
NOT(A) AND NOT(B) AND Cin

## 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

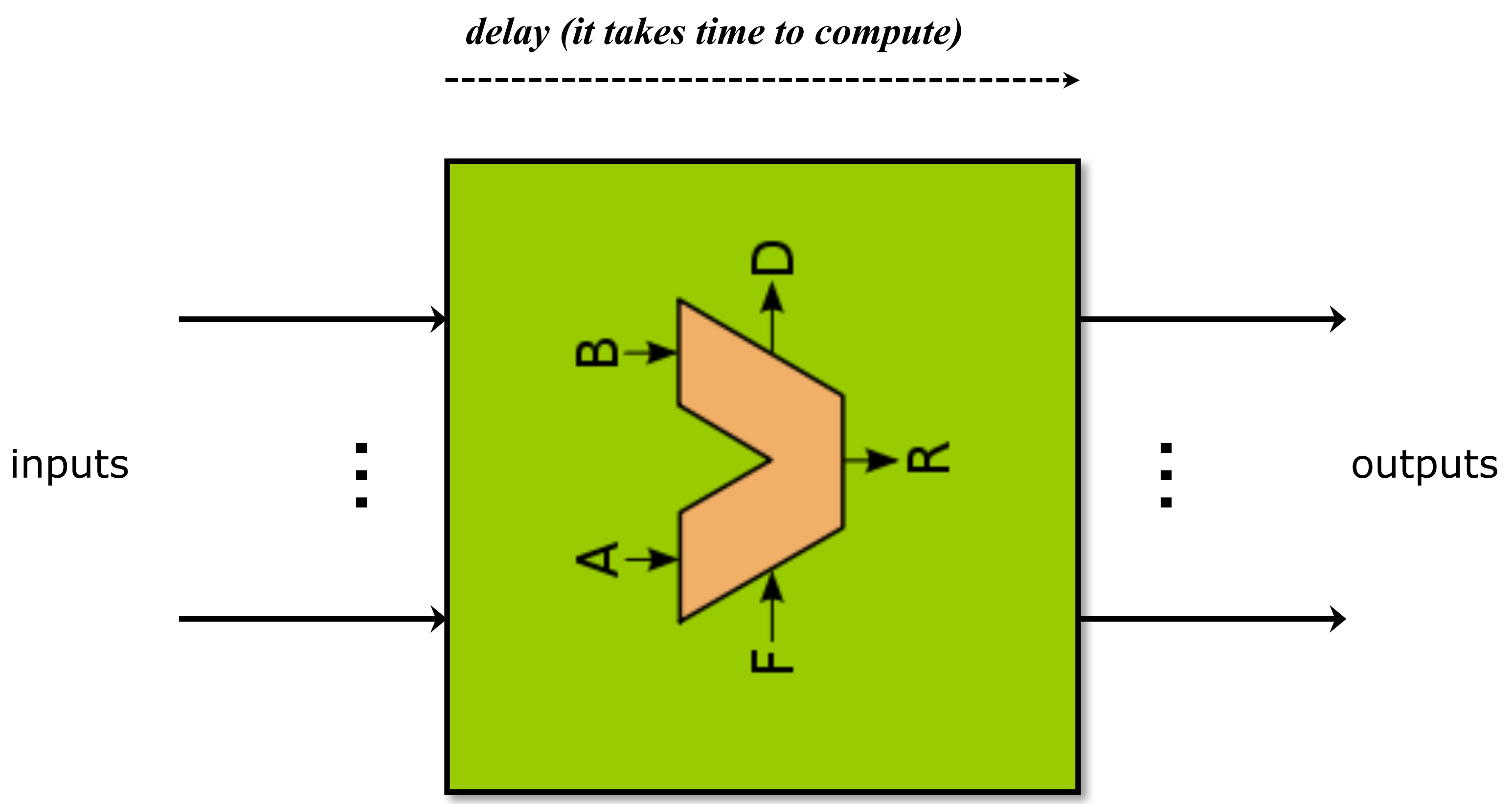
- Sequential logic = **combinational logic + memory**
  - Some memory elements (i.e., “state”)
  - Outputs are dependent on the current state and the current inputs
  - Next state is dependent on the current state and the current inputs

# Combinational logic



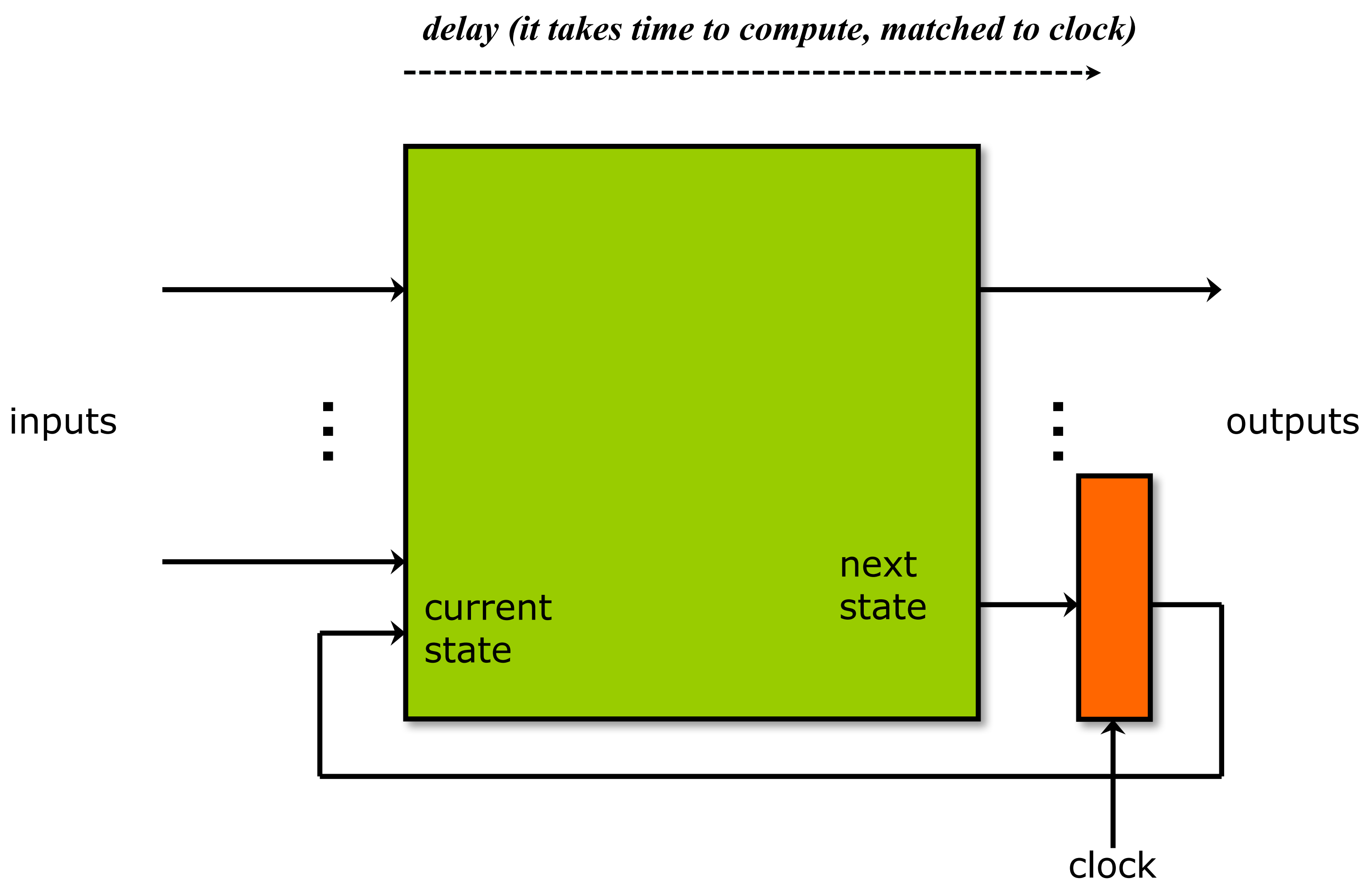
**Outputs are uniquely determined by the inputs at any moment**

## Combinational logic



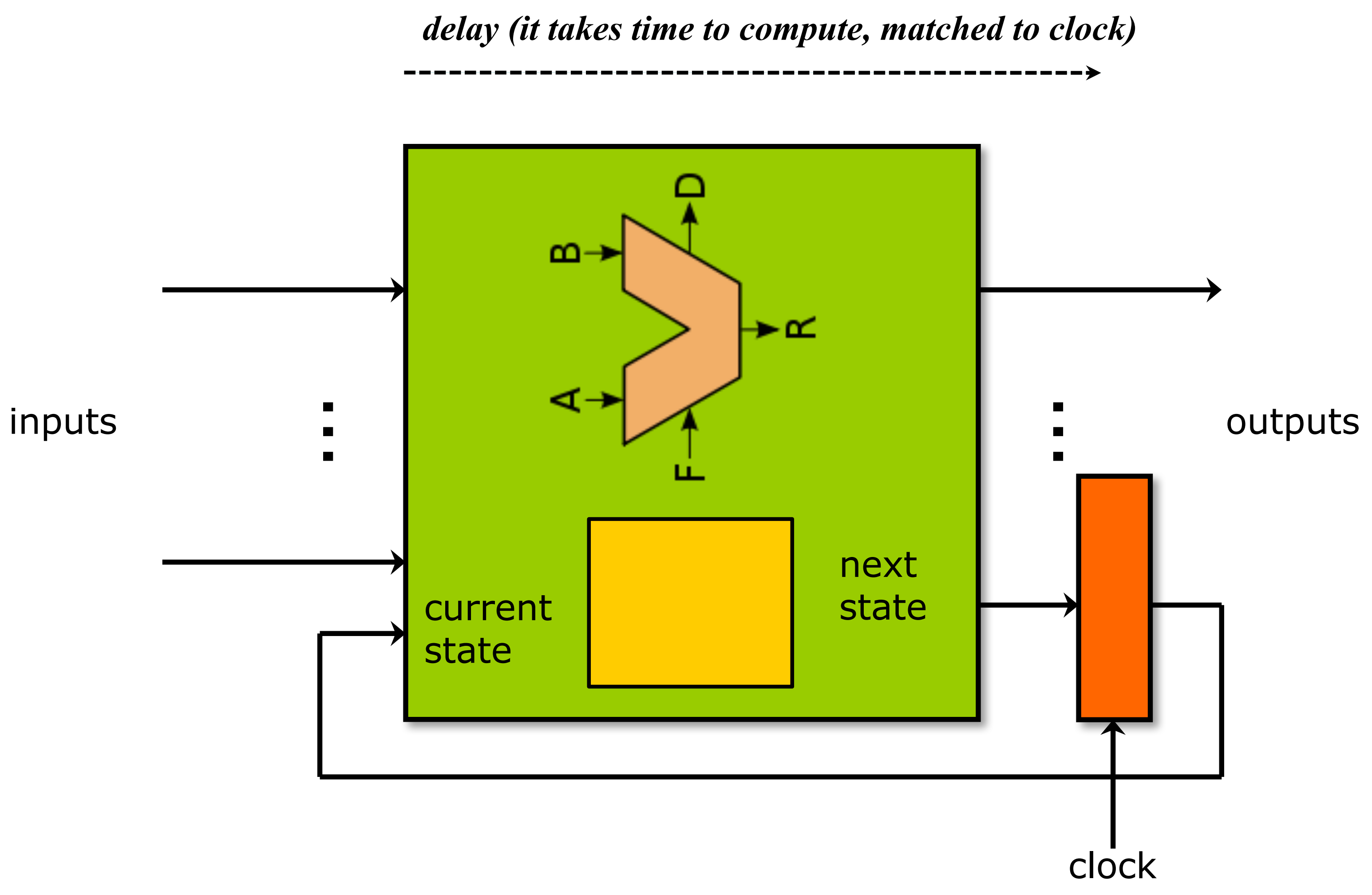
**Outputs are uniquely determined by the inputs at any moment**

## Sequential logic



**Outputs are determined by current & past inputs (past is "state")**

## Sequential logic

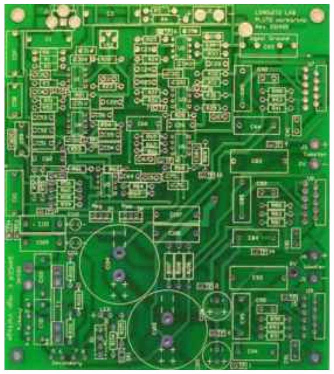


**Outputs are determined by current & past inputs (past is "state")**

## Combinational logic

- Any combinational logic can be implemented using sum of products (OR-AND) or product of sums (AND-OR)

- Input-output relationship can be defined in a ***truth table*** format
- From truth table, derive each ***output function***
- And then we can derive a circuit!! Let’s try it!
  - Example: Write circuit for an 1-bit ADDER



- 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
  - $\sim(a \cdot b) = \sim a + \sim b$
  - $\sim(a+b) = \sim a \cdot \sim b$

- More…
  - $a+(a \cdot b) = a$
  - $a \cdot (a+b) = a$
  - $\sim\sim a = a$
  - $a + \sim a = 1$
  - $a \cdot (\sim a) = 0$

**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:**

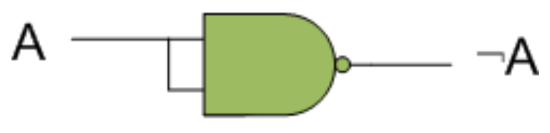
**I didn't go to the store and I didn't go to the library.**

## Expressive power

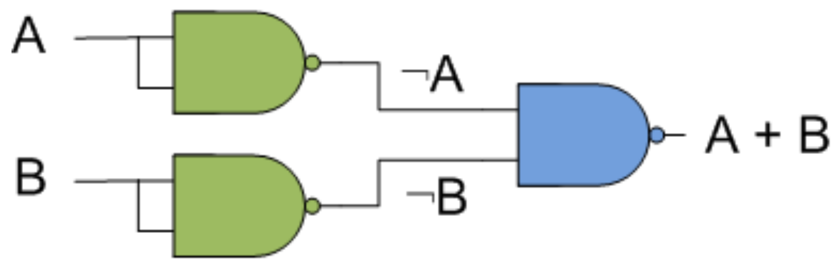
- With AND/OR/NOT, we can express any function in Boolean algebra
  - Sum (+) of products (·)

- What if we have NAND/NOR/NOT?
- What if we have NAND only?
- What if we have NOR only?

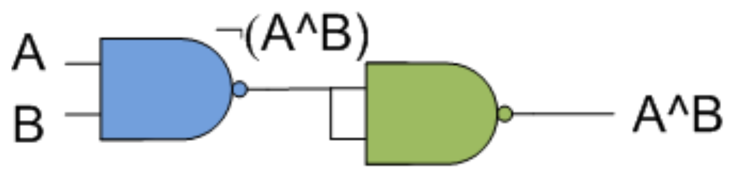
# Using NAND only



¬(A^A)=¬A



¬(¬A^¬B) = A + B



¬(¬(A^B))=A^B

# Using NOR only (your turn)

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

<u>NOT</u>

$= \neg(A + A)$

$= \neg A \wedge \neg A$

$= \neg A$

<u>AND</u>

$= \neg(\neg(A + A) + \neg(B + B))$

$= \neg\ (\neg A \wedge \neg A + \neg B \wedge \neg B)$

$= \neg(\neg A + \neg B)$

$= \neg(\neg A) \wedge \neg(\neg B)$

$= A \wedge B$

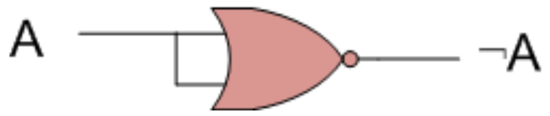
<u>OR</u>

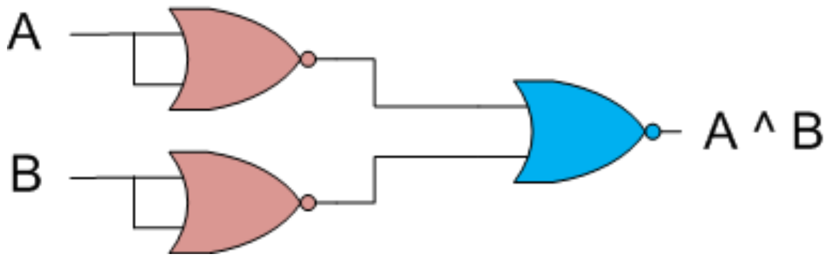
$= \neg(\neg(A + B) + \neg(A + B))$

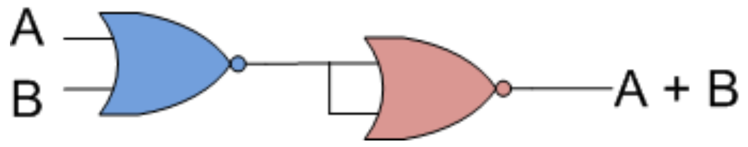
$= (A + B) \wedge (A + B)$

$= A + B$

## Using NOR only (your turn)

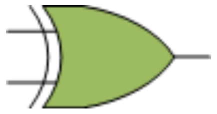






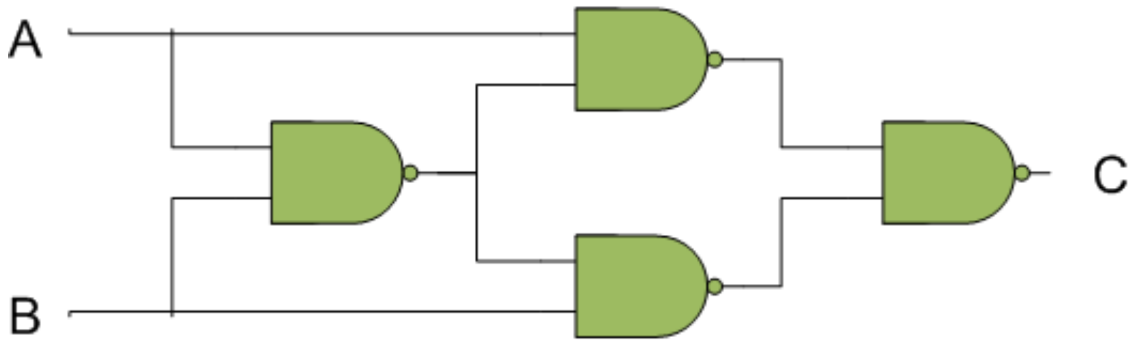
# Now, it's really your turn….

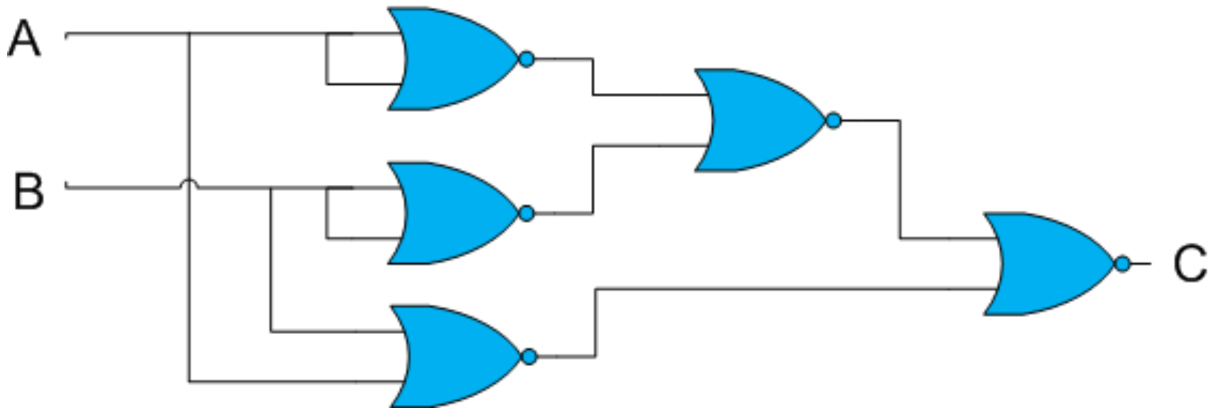
- How about XOR?



| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**C = A'B + AB'**



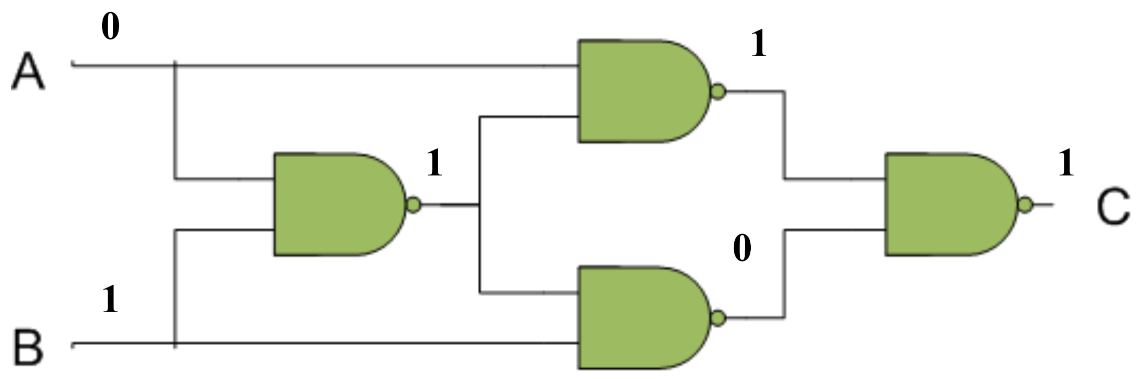


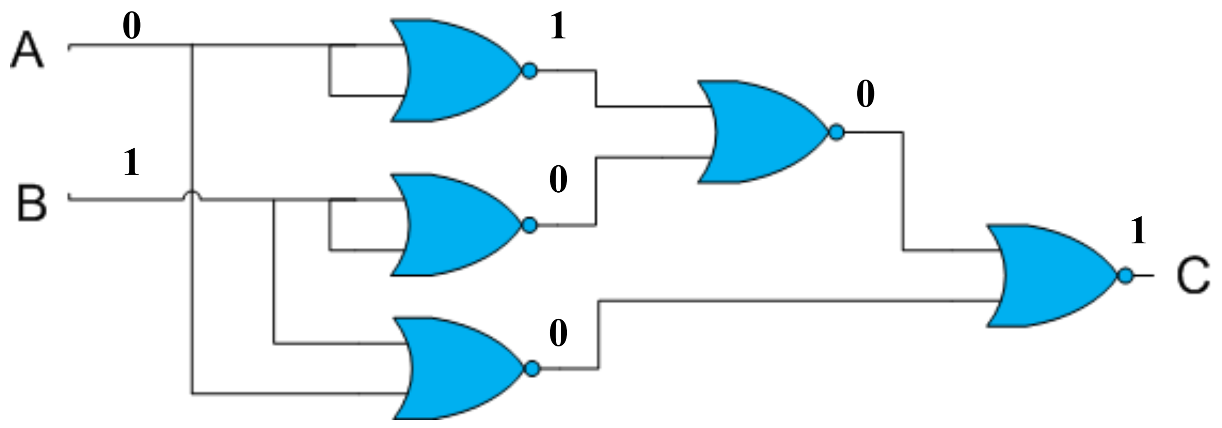
## Now, it's really your turn….

- How about XOR?

| A | B | C |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

**C = A'B + AB'**





## Simplifying expressions

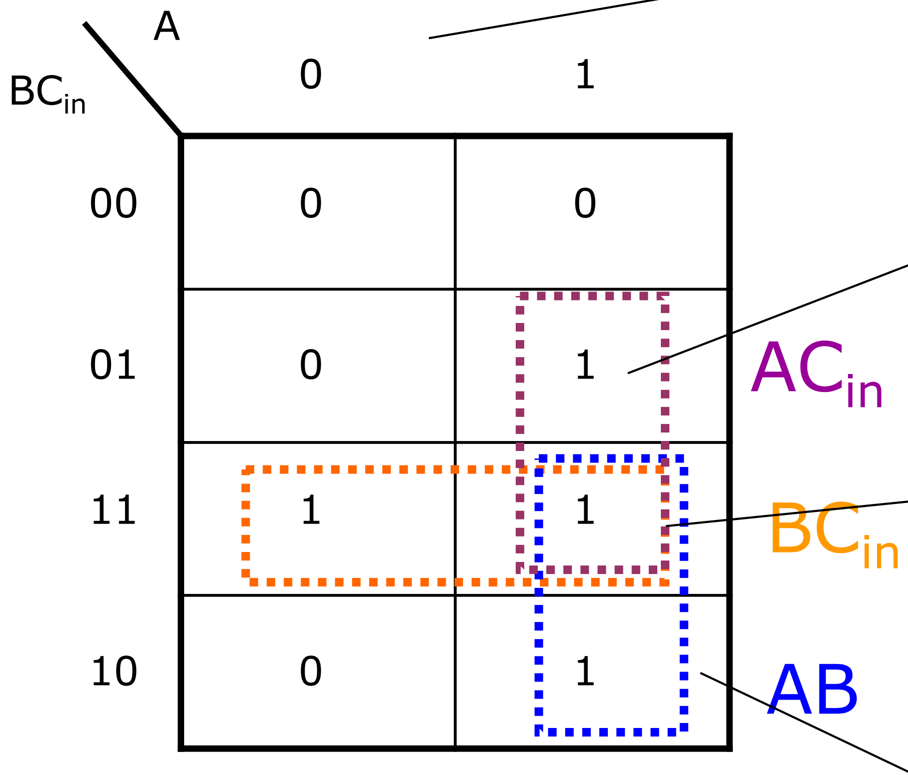
| Input | | | Output | |
|---|---|---|---|---|
| A | B | $C_{in}$ | S | $C_{out}$ |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | $\boxed{1}$ |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | $\boxed{1}$ |
| 1 | 1 | 0 | 0 | $\boxed{1}$ |
| 1 | 1 | 1 | 1 | $\boxed{1}$ |

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

- Simplification reduces complexity: faster, smaller circuit!

# Karnaugh map

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



| $BC_{in}$ \ A | 0 | 1 |
|---|---|---|
| 00 | 0 | 0 |
| 01 | 0 | 1 |
| 11 | 1 | 1 |
| 10 | 0 | 1 |

$C_{out} = BC_{in} + AB + AC_{in}$

**A "tool" to help simplify boolean expressions**
**Like a "slide rule": Useful but limited**

**A truth table listing "minterms"**
**Minterms written in Gray code order**
**One var value changes betw. col/row**

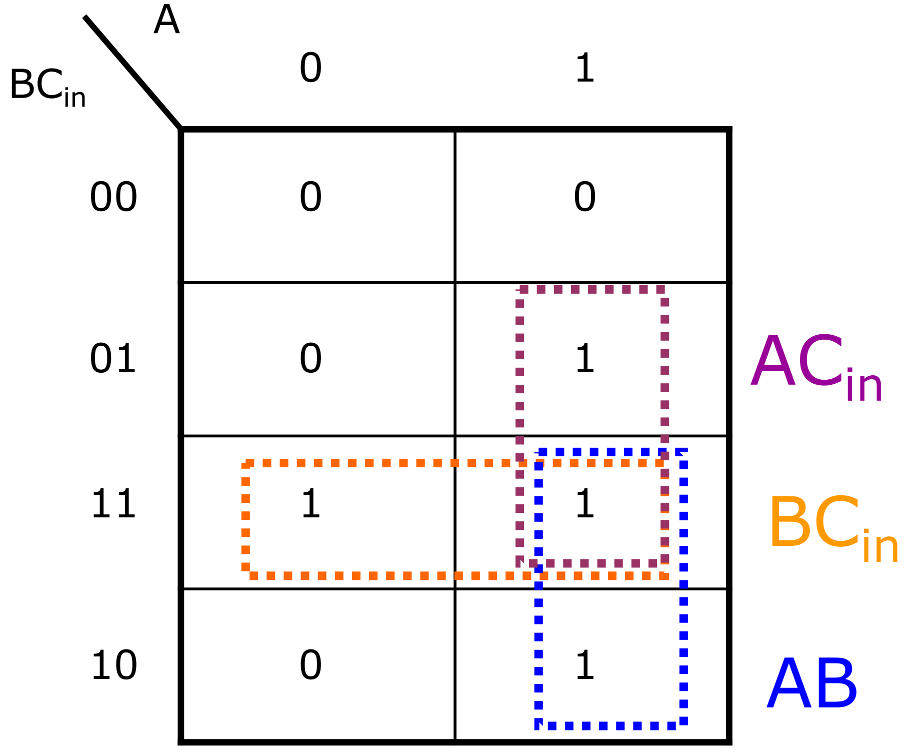
**Build from the initial boolean expr.**
**Put a "1" where a minterm is true**

**E.g.., $AB'C_{in}$ has a 1**

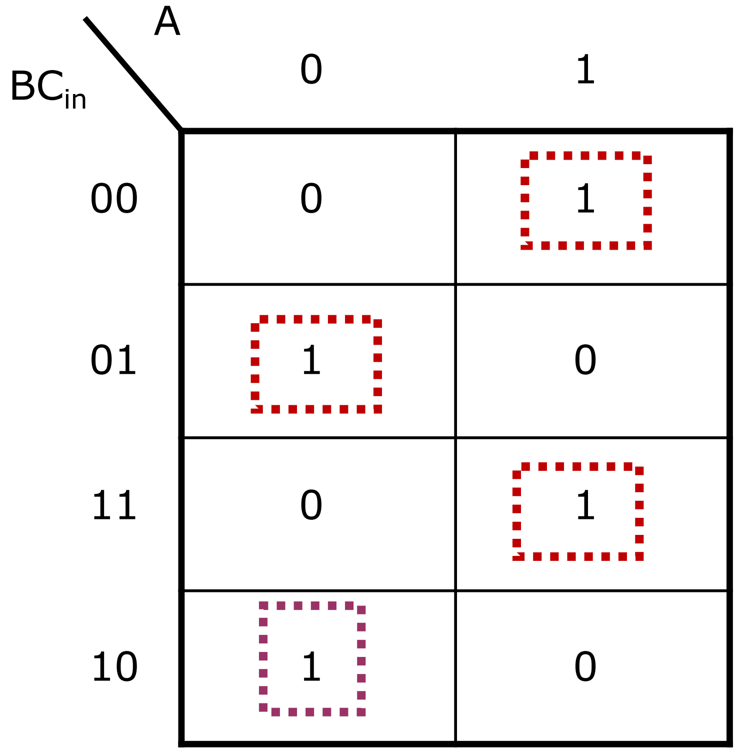
**Now, to simplify:**
**Look for adjacent max rectangular groups with power of 2 elements.**
**In such a group, some var is {0,1}**
**Eliminate that variable**

**Here's another one!**
**Groups can be vertical too.**
**They can even "wrap around"**
**They can also overlap**
**Diagonals aren't allowed**

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


AC_in
BC_in
AB

| $BC_{in}$ \ A | 0 | 1 |
|---|---|---|
| 00 | 0 | 0 |
| 01 | 0 | 1 |
| 11 | 1 | 1 |
| 10 | 0 | 1 |



| $BC_{in}$ \ A | 0 | 1 |
|---|---|---|
| 00 | 0 | 1 |
| 01 | 1 | 0 |
| 11 | 0 | 1 |
| 10 | 1 | 0 |

$C_{out} = BC_{in} + AB + AC_{in}$

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

# Four (or more?) Variables

CD

AB

| | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 0 | 0 | 0 | 0 |
| 11 | 0 | 1 | 1 | 0 |
| 10 | 0 | 1 | 1 | 0 |

**Can you minimize this one?**

**In AB: B is both {0,1}**
**In CD: C is both {0,1}**

**Eliminate B, C**
**Thus, we have just AD**

CD

AB

| | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 0 | 0 | 0 | 0 |
| 01 | 1 | 1 | 1 | 1 |
| 11 | 1 | 1 | 1 | 1 |
| 10 | 0 | 0 | 0 | 0 |

**Can you minimize this one?**

**C,D both have {0,1}**
**A has {0,1}**

**Eliminate A,C,D**
**Thus, we have just B**

# Four (or more?) Variables

CD

AB

| | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | 1 | 0 | 0 | 1 |
| 01 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 |
| 10 | 1 | 0 | 0 | 1 |

**Can you minimize this one?**

**Combine on top row**
**Combine on bottom row**

**A’B’D’**
**AB’D’**

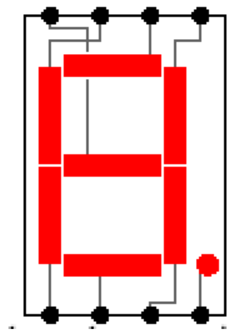
**These terms can now combine**
**Thus, we have B’D’**

**Karnaugh Maps (K-Maps) are a simple calculation tool.**

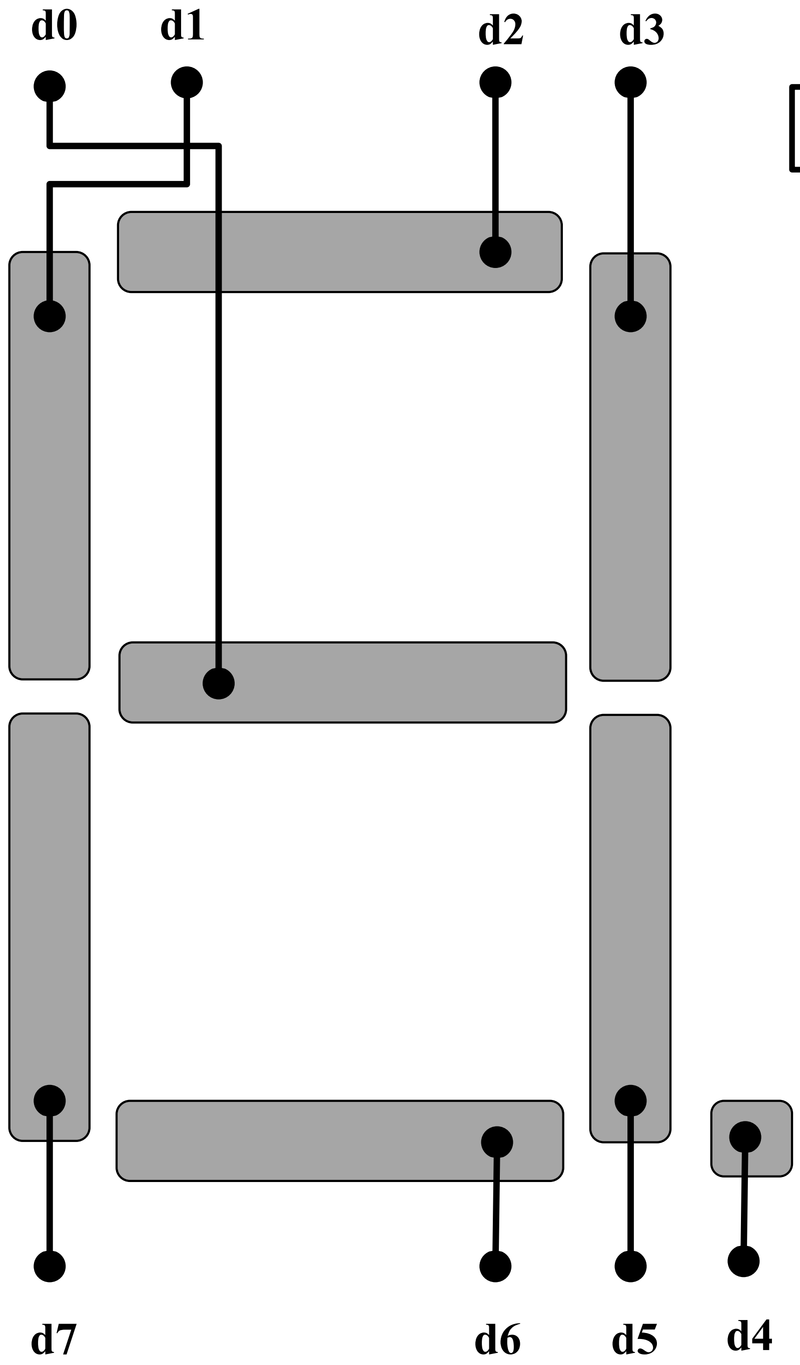
**In practice, sophisticated logic synthesis algorithms/tools are used.**

## In-class Example

- A device called a "7 segment LED digit"
- There are 8 LEDs – one for seven "segments" of a numeral and 1 for a decimal point



- **Problem**
  - Given a 3-bit number, draw the corresponding numeral
  - E.g., 000 is the numeral 0, 001 is numeral 1 and so forth
- **Solution**
  - Create a Boolean function for each segment. Ignore the decimal point.
  - Boolean function over three inputs for the 3-bit number.
- **Let's try it!!**



**Segments numbered d0 to d7**

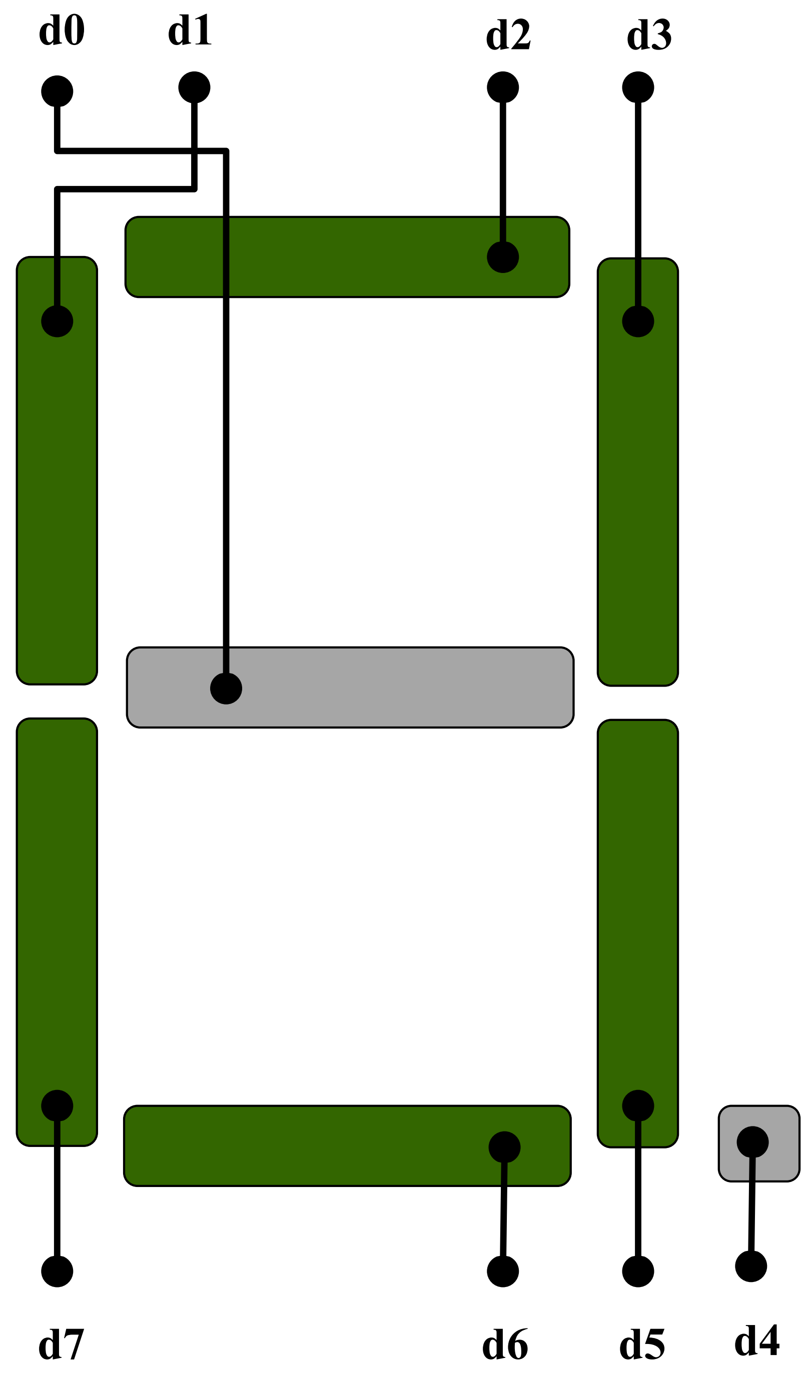
**<u>Hex Digit LED</u>**

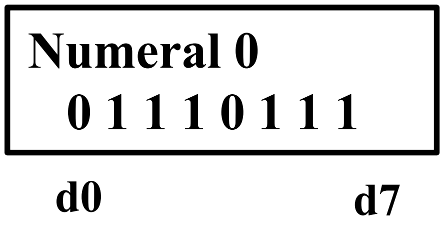
**7 segments, 1 decimal point**

**Turn each segment on/off**

**State: 0=OFF, 1=ON**

**“Draw” numbers 0 to 9**





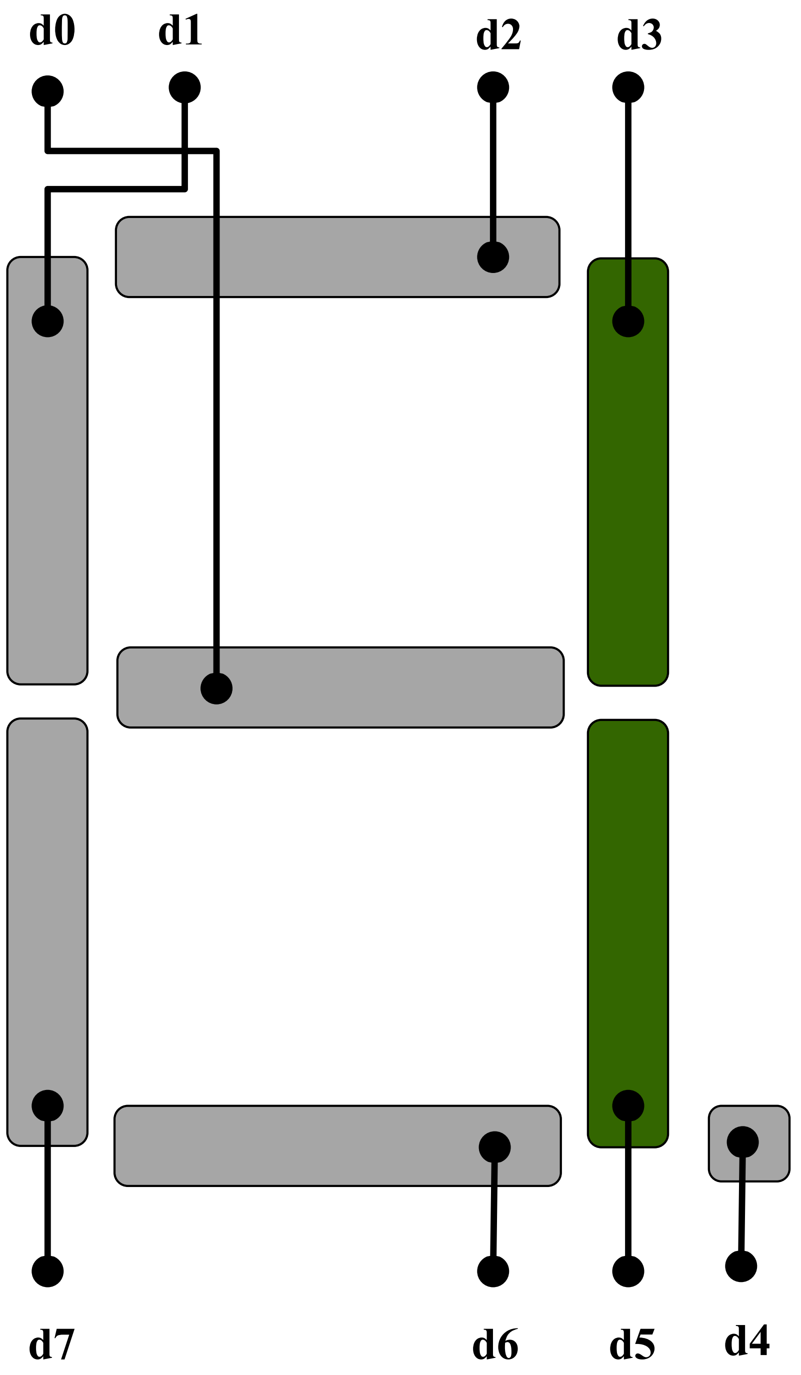
**<u>Hex Digit LED</u>**

**7 segments, 1 decimal point**

**Turn each segment on/off**

**State: 0=OFF, 1=ON**

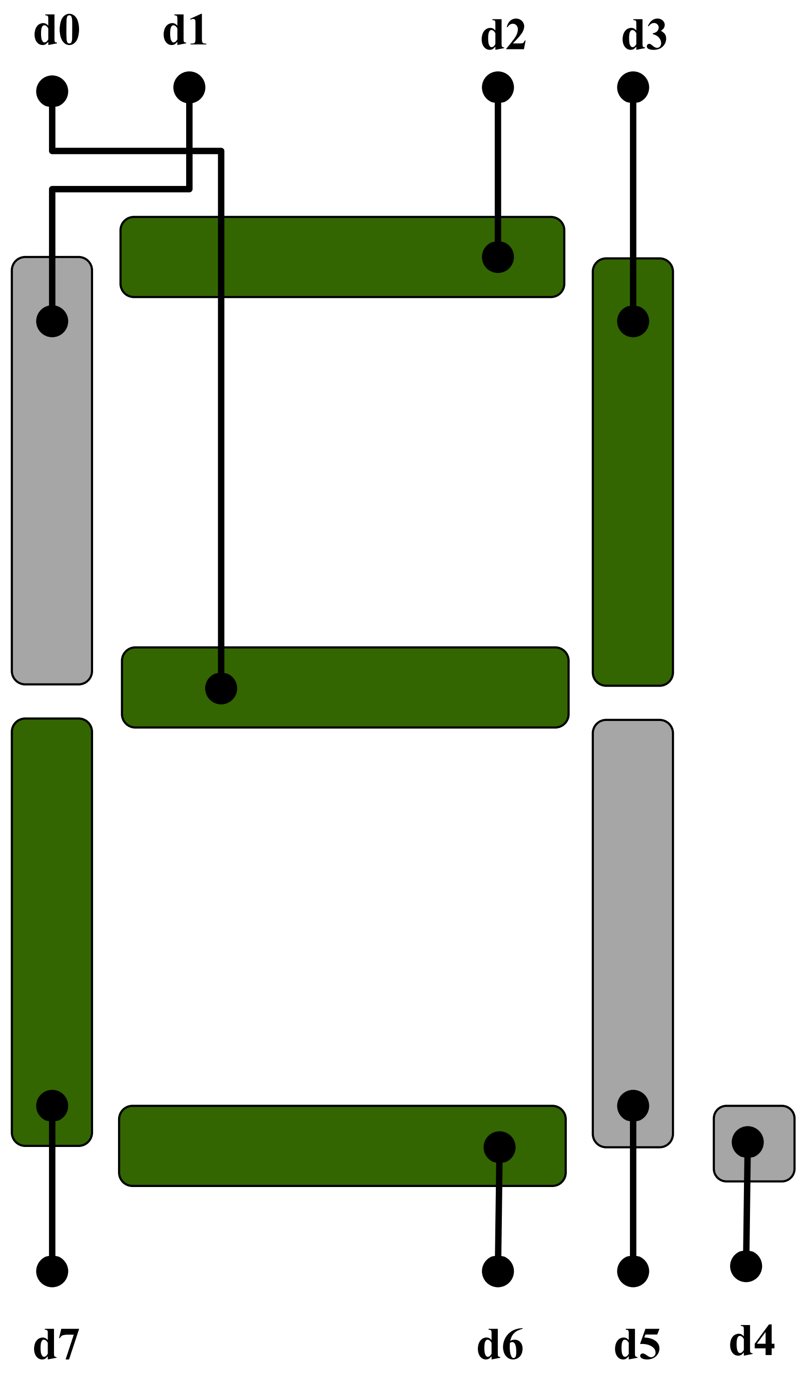
**“Draw” numbers 0 to 9**



Numeral 1
0 0 0 1 0 1 0 0

d0 d7

**Hex Digit LED**
**7 segments, 1 decimal point**
**Turn each segment on/off**
**State: 0=OFF, 1=ON**
**“Draw” numbers 0 to 9**



Numeral 2
1 0 1 1 0 0 1 1
d0 d7

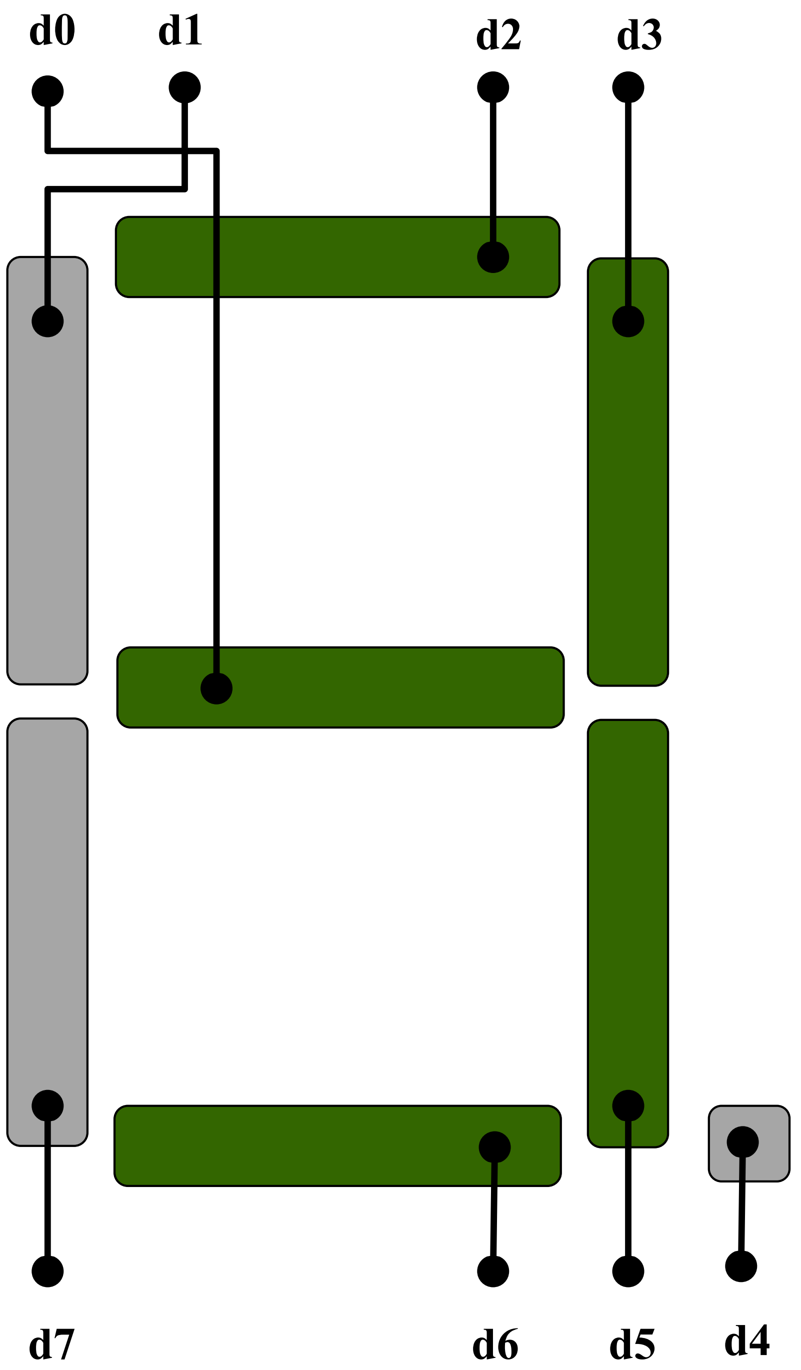
**Hex Digit LED**

7 segments, 1 decimal point

Turn each segment on/off

State: 0=OFF, 1=ON

“Draw” numbers 0 to 9



Numeral 3
1 0 1 1 0 1 1 0

d0 d7

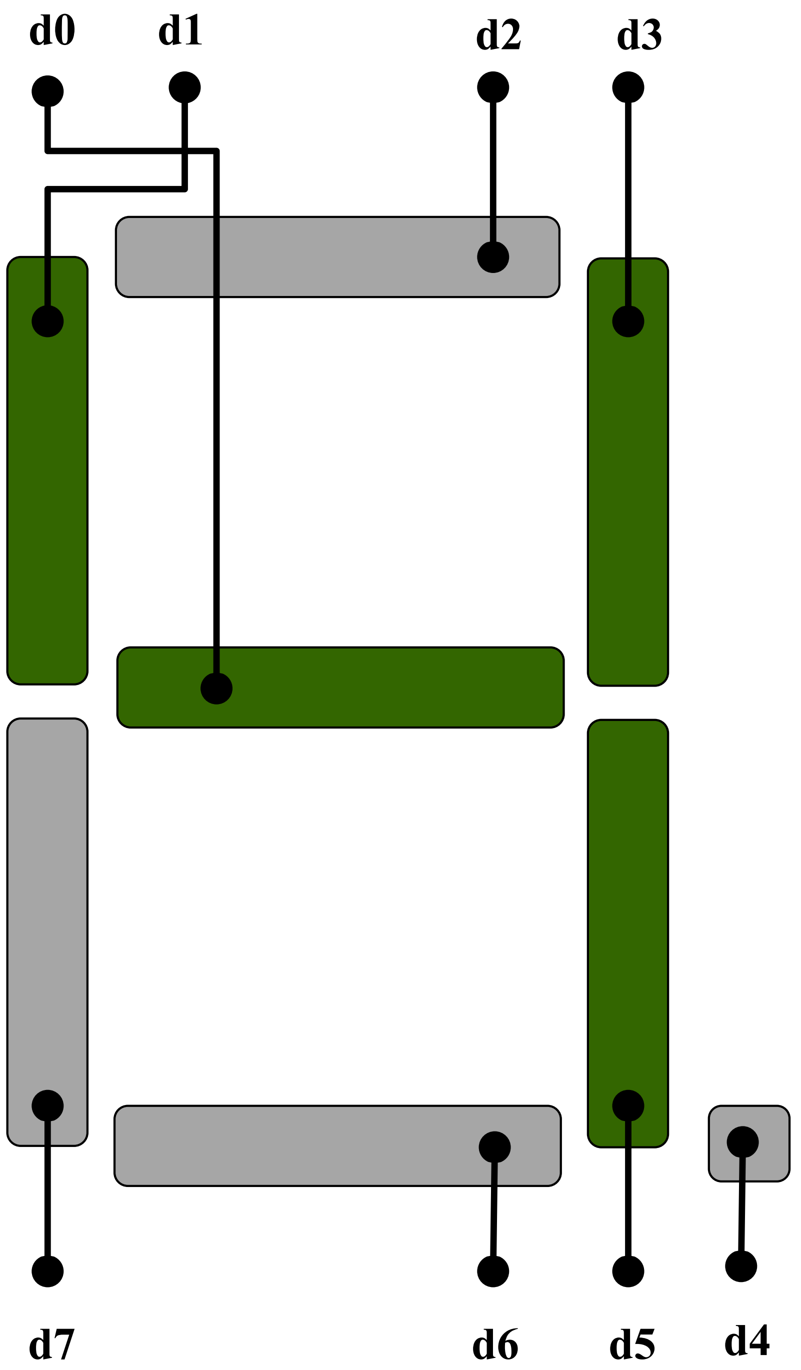
## Hex Digit LED

7 segments, 1 decimal point

Turn each segment on/off

State: 0=OFF, 1=ON

"Draw" numbers 0 to 9



Numeral 4
1 1 0 1 0 1 0 0
d0 d7

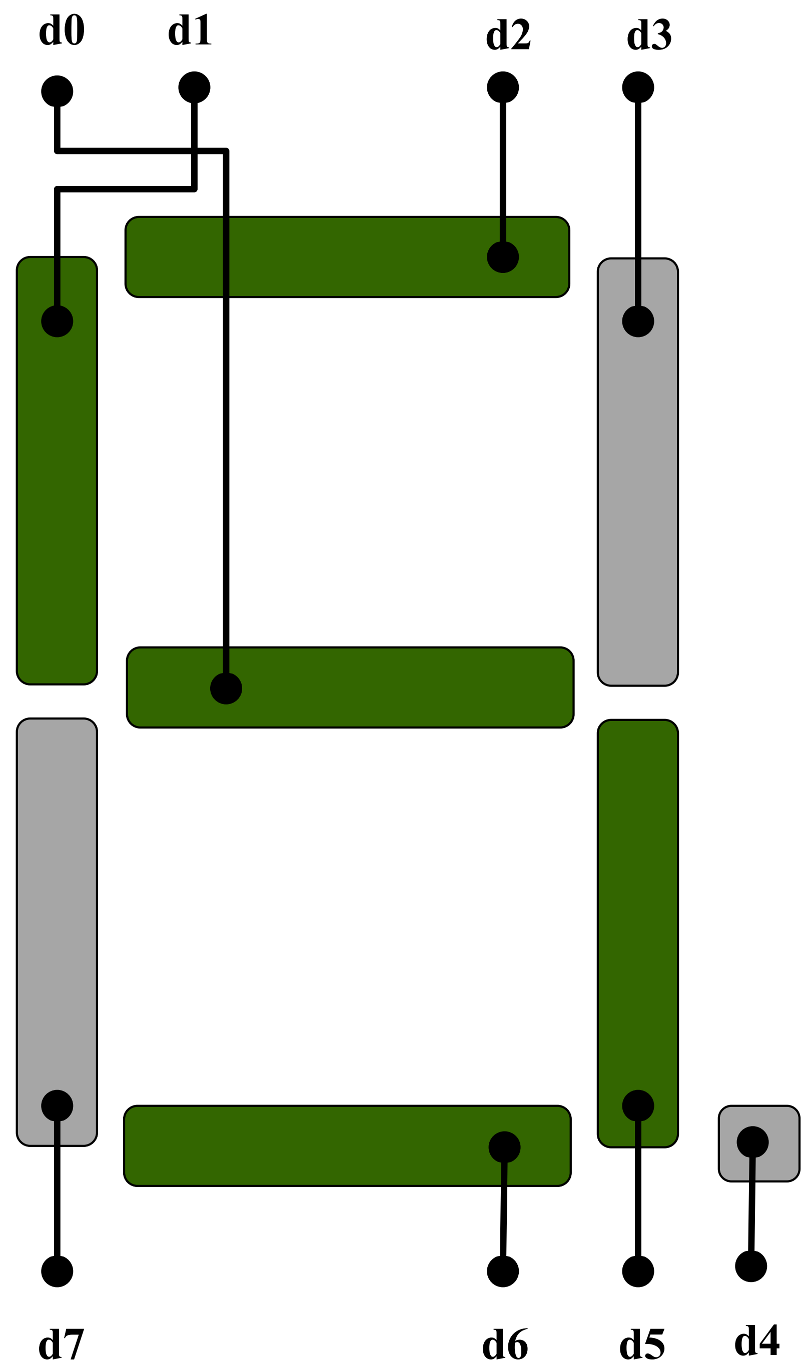
**Hex Digit LED**

**7 segments, 1 decimal point**

**Turn each segment on/off**

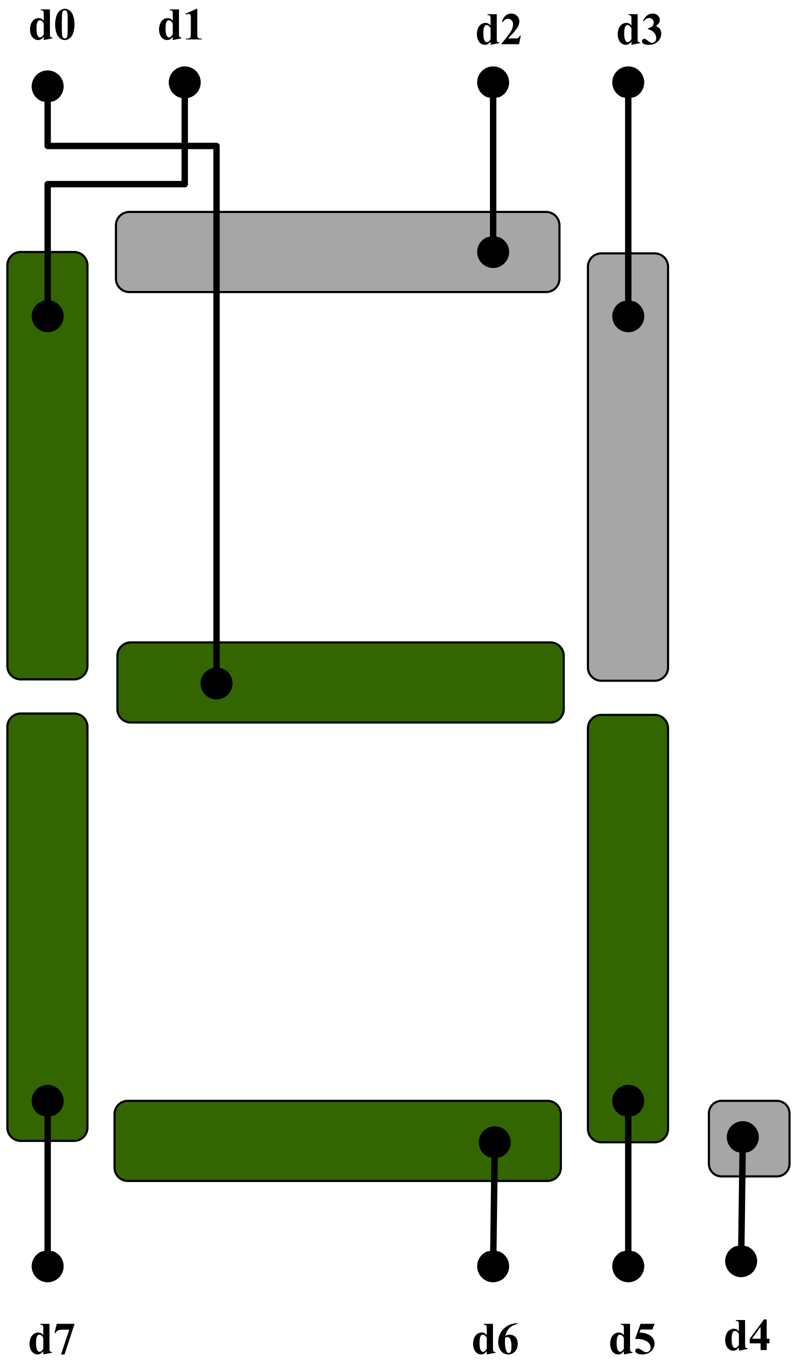
**State: 0=OFF, 1=ON**

**"Draw" numbers 0 to 9**



Numeral 5
1 1 1 0 0 1 1 0
d0 d7

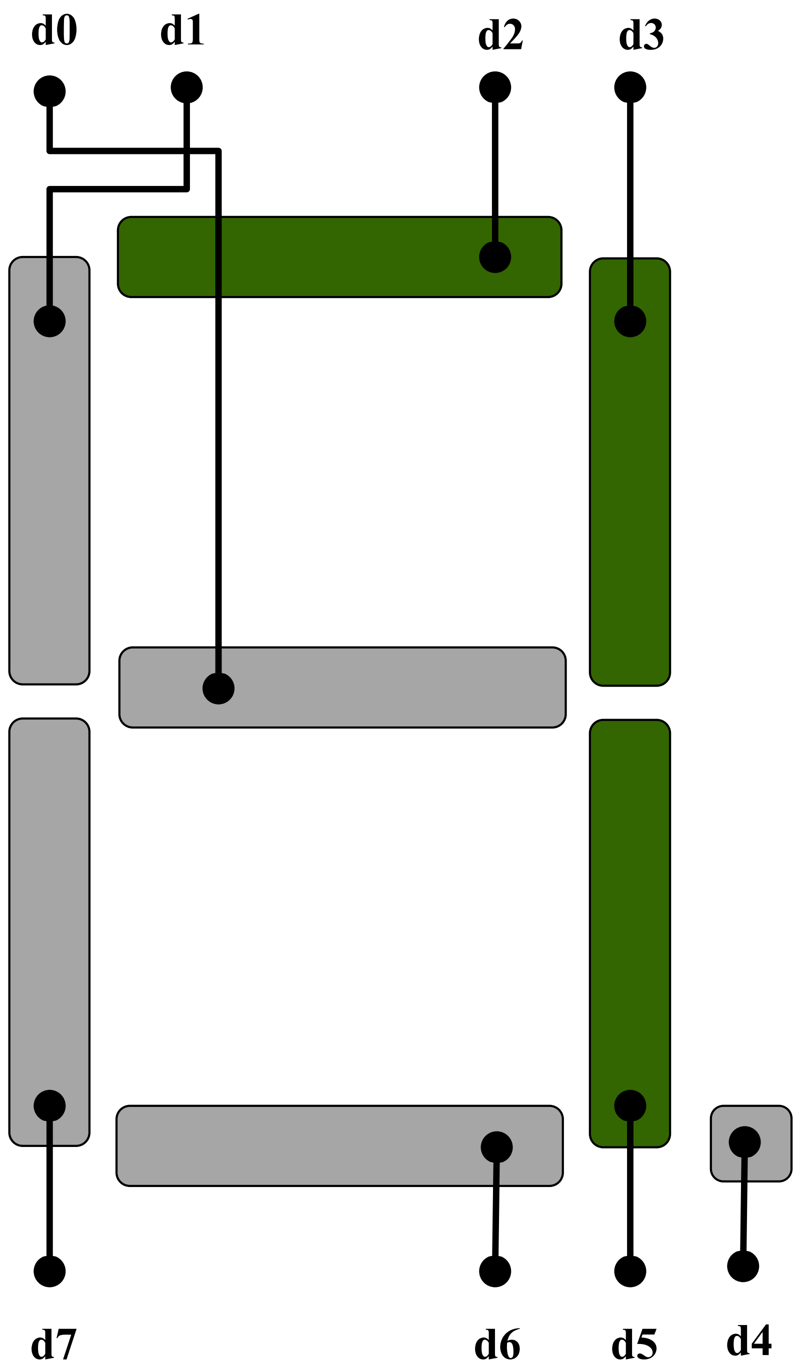
**Hex Digit LED**
7 segments, 1 decimal point
Turn each segment on/off
State: 0=OFF, 1=ON
"Draw" numbers 0 to 9



Numeral 6
1 1 0 0 0 1 1 1
d0 d7

## Hex Digit LED

7 segments, 1 decimal point
Turn each segment on/off
State: 0=OFF, 1=ON
"Draw" numbers 0 to 9



Numeral 7
0 0 1 1 0 1 0 0
d0 d7

**Hex Digit LED**

7 segments, 1 decimal point

Turn each segment on/off

State: 0=OFF, 1=ON

"Draw" numbers 0 to 9

## In-class Example

- Create a truth table
- Inputs are numbered i0 to i2 (3 bits)
- Outputs are numbered d0 to d7, corresponding to segments
- “Draw” the numerals by setting d0 to d7 to 1s or 0s.

| **inputs** | | | **outputs** | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **i2** | **i1** | **i0** | **d0** | **d1** | **d2** | **d3** | **d4** | **d5** | **d6** | **d7** |
| 0 | 0 | 0 | | | | | | | | |
| 0 | 0 | 1 | | | | | | | | |
| 0 | 1 | 0 | | | | | | | | |
| 0 | 1 | 1 | | | | | | | | |
| 1 | 0 | 0 | | | | | | | | |
| 1 | 0 | 1 | | | | | | | | |
| 1 | 1 | 0 | | | | | | | | |
| 1 | 1 | 1 | | | | | | | | |

**Input: 3-bit number**

**Outputs: Segments for the LED hex digit**

| inputs | | | outputs | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **i2** | **i1** | **i0** | **d0** | **d1** | **d2** | **d3** | **d4** | **d5** | **d6** | **d7** |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | | | | | | | | |
| 1 | 0 | 0 | | | | | | | | |
| 1 | 0 | 1 | | | | | | | | |
| 1 | 1 | 0 | | | | | | | | |
| 1 | 1 | 1 | | | | | | | | |

**Fill in the truth table for each numeral**
**Numerals 0 to 2 are shown.**
***Can you complete 3 to 7?***

inputs outputs

| i2 | i1 | i0 | d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

**Completed truth table**
**Now, write down the *minimal* (simplified) Boolean functions**
**Use a K-map to minimize each one!**

inputs outputs

| i2 | i1 | i0 | d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

**Completed truth table**
**Now, write down the *minimal* (simplified) Boolean functions**
**Use a K-map to minimize each one!**

**i1, i0**

**i2**

| | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | | | | |
| 1 | | | | |

**Use a K-map for each output function – d0 to d7**

**Let’s start with d0**
**We’ll only do a few – d0, d3 and d5**

***Can you do the rest on your own???***

## Function d0

| i2 \ i1, i0 | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 |

| i2 | i1 | i0 | d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

## Function d0

**i1, i0**

| **i2** | | **00** | **01** | **11** | **10** |
|---|---|---|---|---|---|
| | **0** | 0 | 0 | 1 | 1 |
| | **1** | 1 | 1 | 0 | 1 |

| **i2** | **i1** | **i0** | **d0** | **d1** | **d2** | **d3** | **d4** | **d5** | **d6** | **d7** |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

**3 terms**
**i2’i1**
**i2i1’**
**I2i0’**

$$d0=\overline{i2}i1 + i2\overline{i1} + i2\overline{i0}$$

## Function d3

| i2 \ i1, i0 | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 |

| i2 | i1 | i0 | d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

## Function d3

i1, i0

|  |  | 00 | 01 | 11 | 10 |
|---|---|---|---|---|---|
| i2 | 0 | 1 | 1 | 1 | 1 |
|  | 1 | 1 | 0 | 1 | 0 |

| i2 | i1 | i0 | d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

$$d3=\overline{i2} + \overline{i1}\,\overline{i0} + i1\,i0$$

## Function d5

| | | i1, i0 | | | |
|---|---|---|---|---|---|
| | | 00 | 01 | 11 | 10 |
| i2 | 0 | 1 | 1 | 1 | 0 |
| | 1 | 1 | 1 | 1 | 1 |

| i2 | i1 | i0 | d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

## Function d5

**i1, i0**

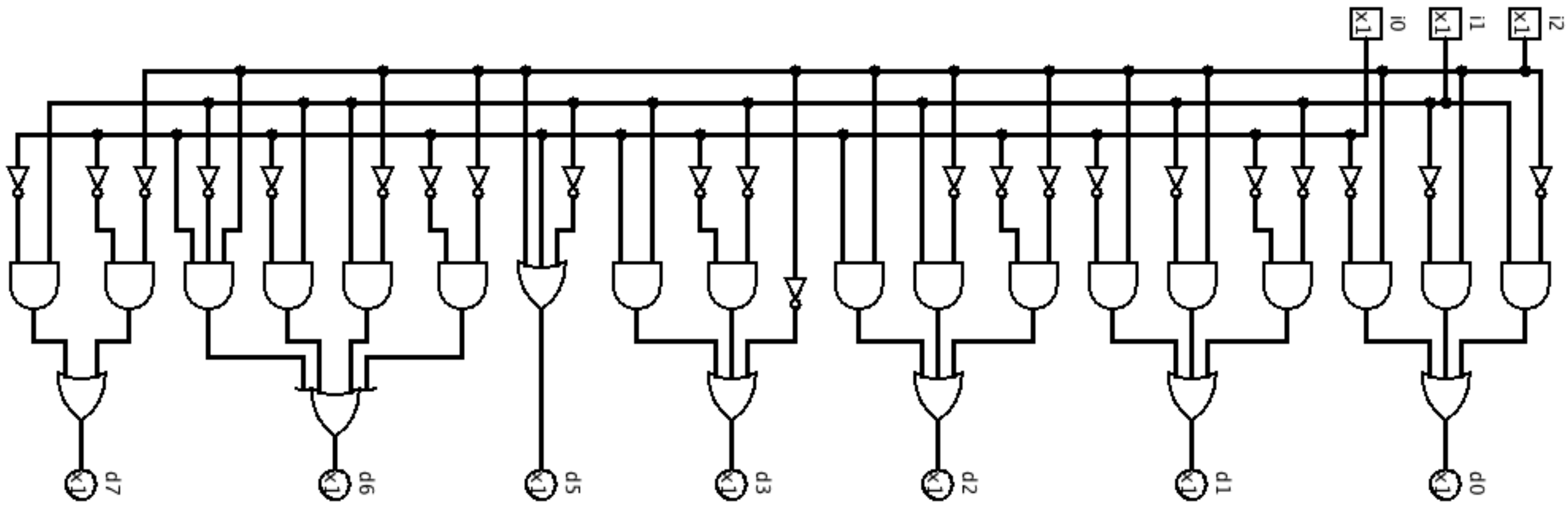
| i2 | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

| i2 | i1 | i0 | d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |

**$d5 = \overline{i1} + i0 + i2$**

## Completed Circuit with all functions d0 to d7

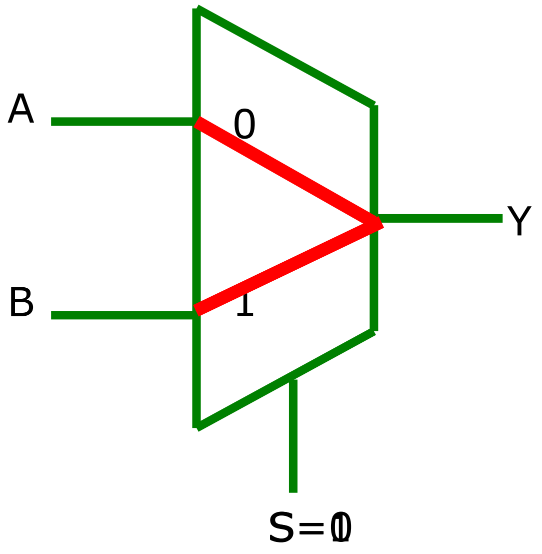
Inputs



Outputs to the LED hex digit

See example: LEDhexdigit.circ

# Multiplexor (aka MUX)
# An example, yet VERY useful circuit!

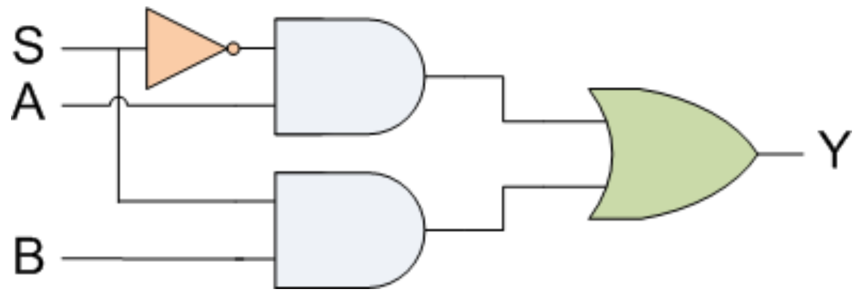


Y = (S) ? B:A;

| S | A | B | Y |
|---|---|---|---|
| 0 | 0 | x | 0 |
| 0 | 1 | x | 1 |
| 1 | x | 0 | 0 |
| 1 | x | 1 | 1 |

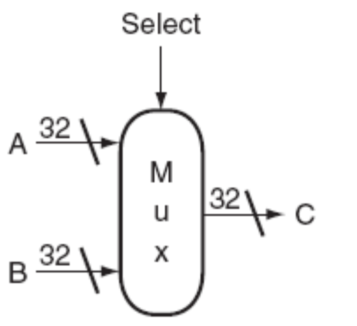
Y=S'A+SB

**when S =**
**0: output A**
**1: output B**

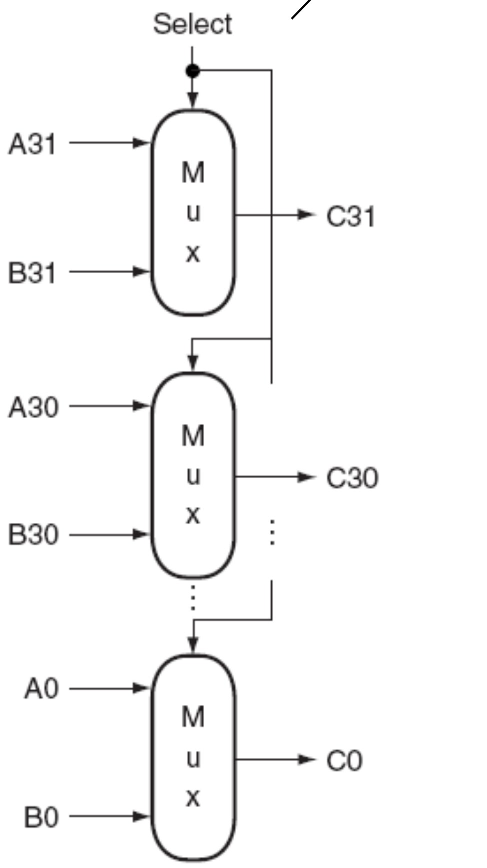


# A 32-bit MUX

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



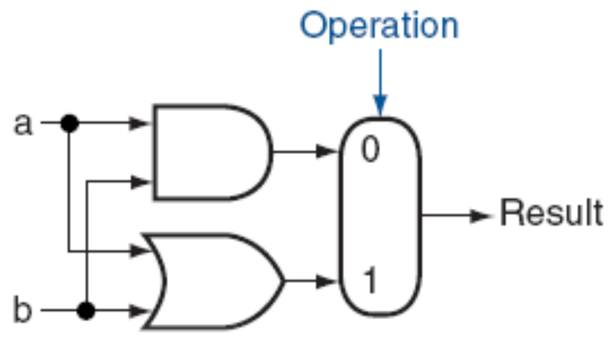
a. A 32-bit wide 2-to-1 multiplexor

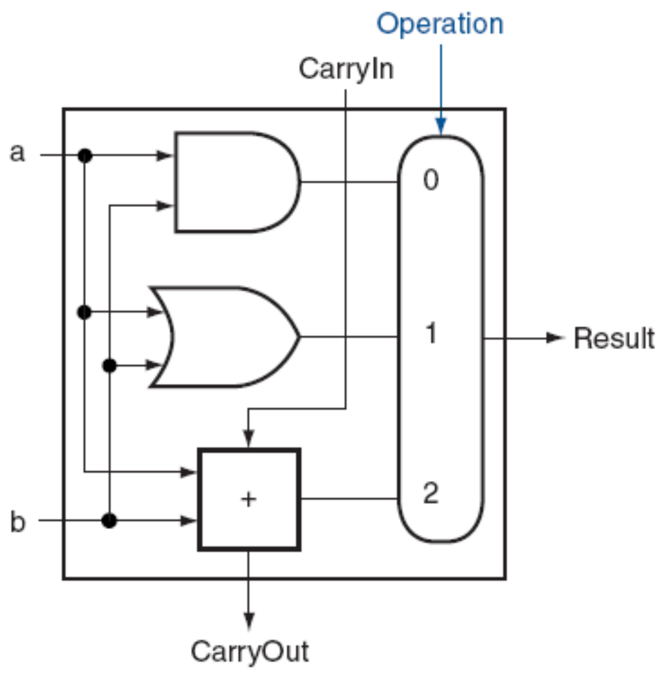


b. The 32-bit wide multiplexor is actually an array of 32 1-bit multiplexors

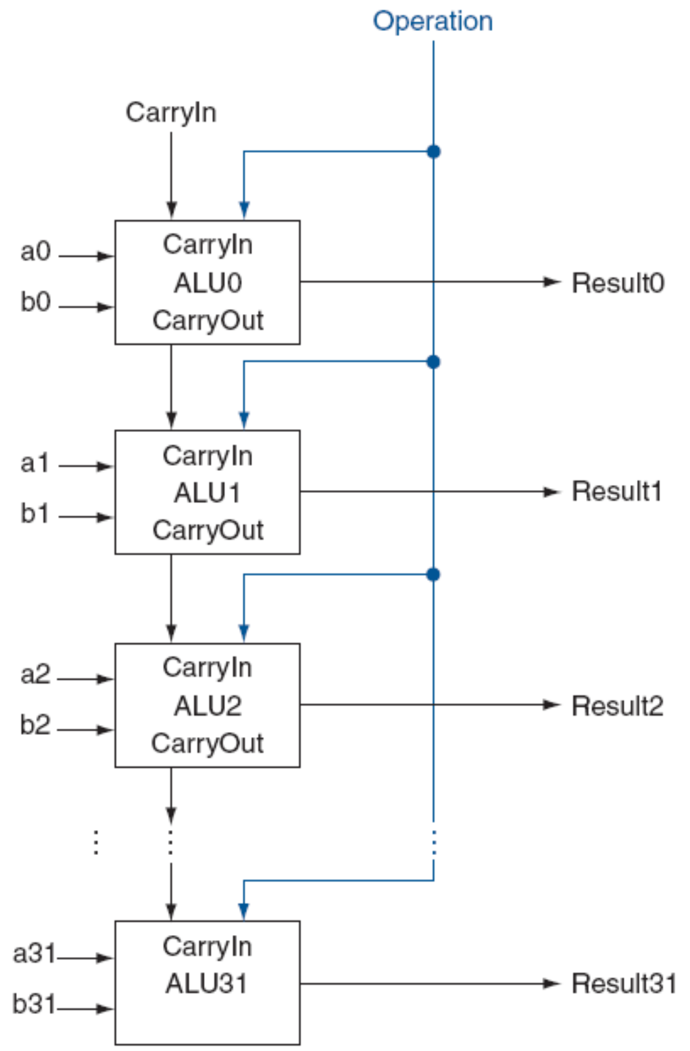
## Building a 1-bit ALU

- ALU = arithmetic logic unit = arithmetic unit + logic unit



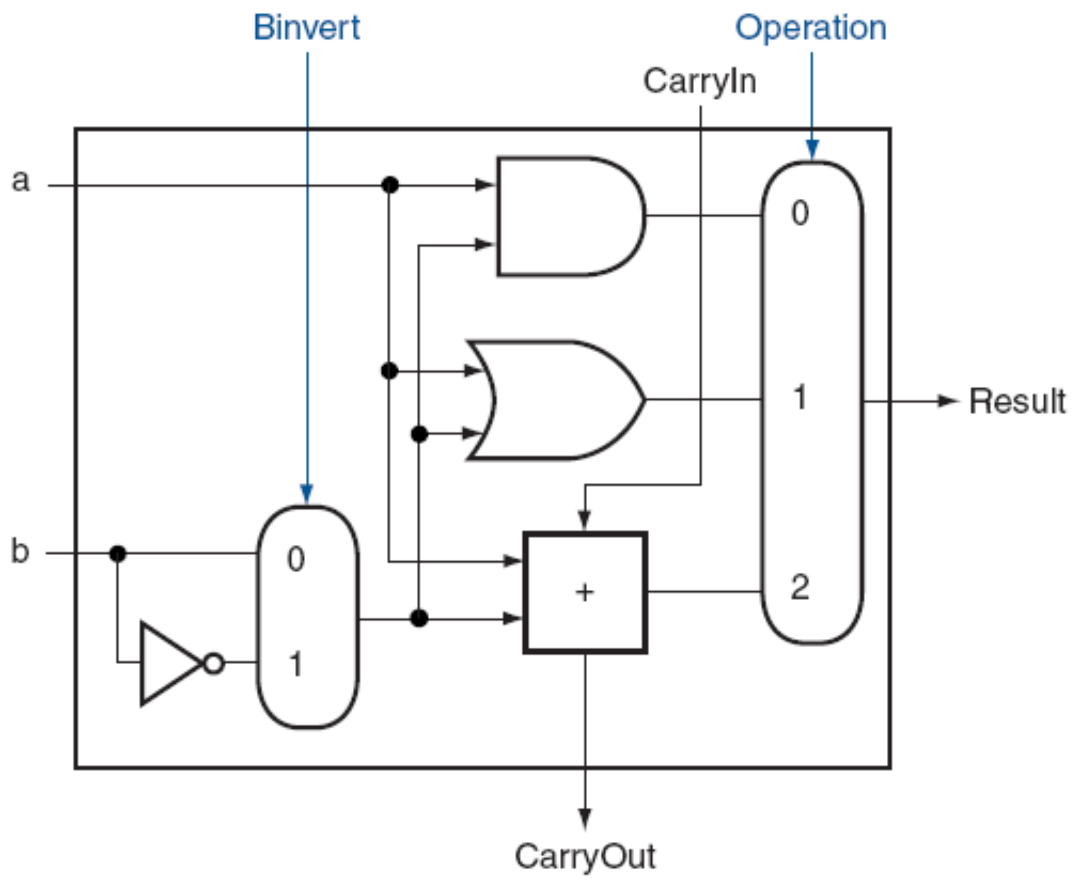


## Building a 32-bit ALU



# Implementing “sub”

**Binvert=1**
**CarryIn=1 for 1st 1-bit ALU**
**Operation=2**

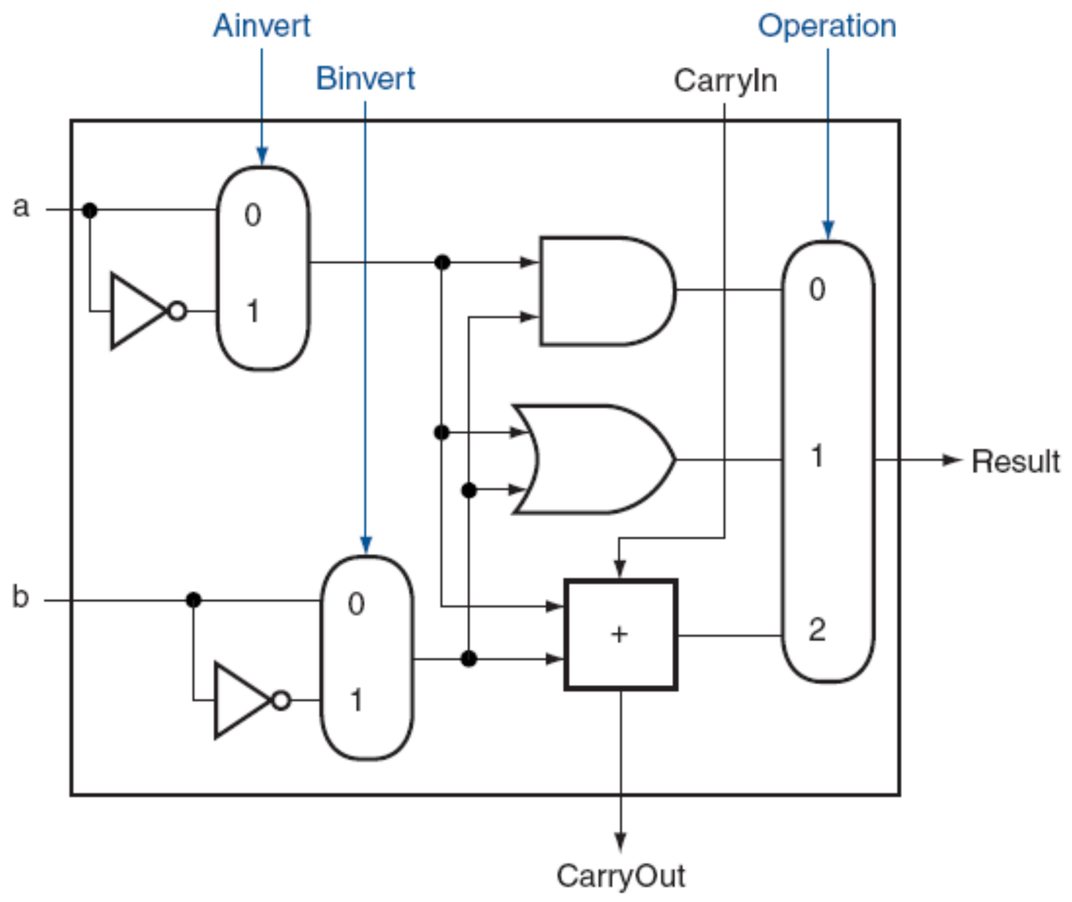


# Implementing NAND and NOR

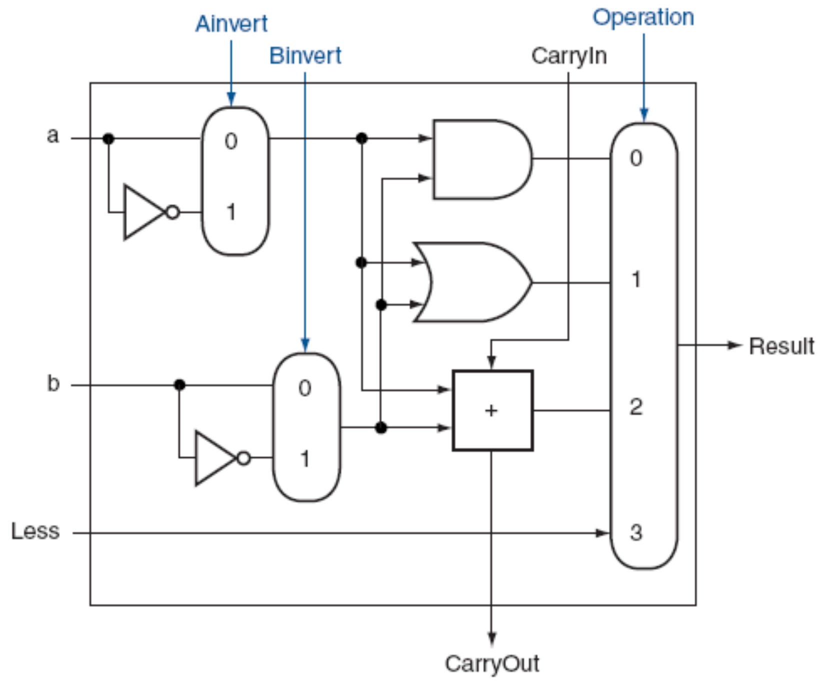
**NOR:**
**NOT (A OR B)**
**by DeMorgan's Law:**
**(NOT A) AND (NOT B)**

**Thus,**
**Operation=0,**
**Ainvert=1,**
**Binvert=1**

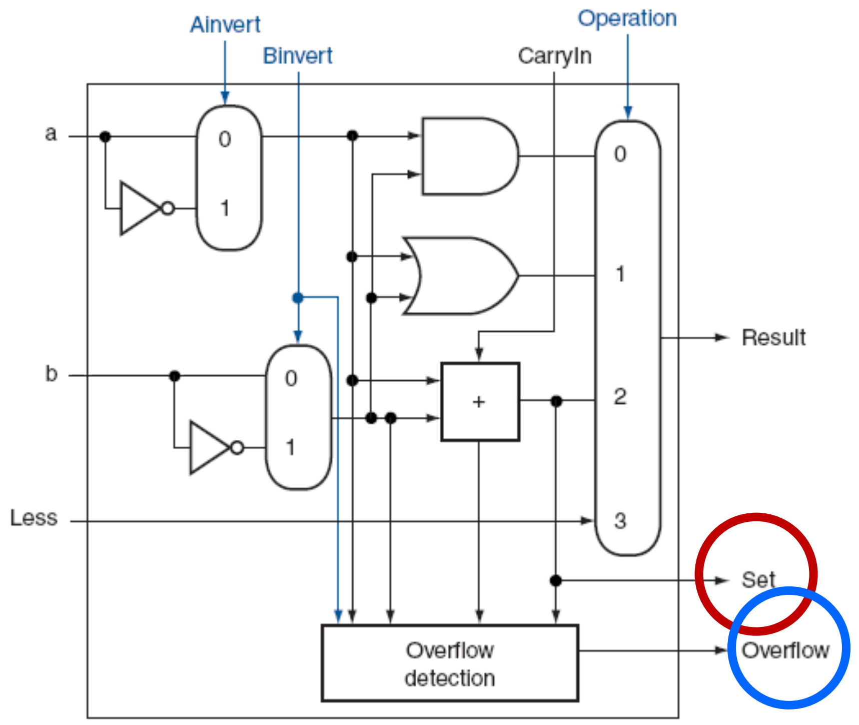
**And, NAND???**



# Implementing SLT (set-less-than)



**1-bit ALU for bits 0~30**



**1-bit ALU for bit 31**

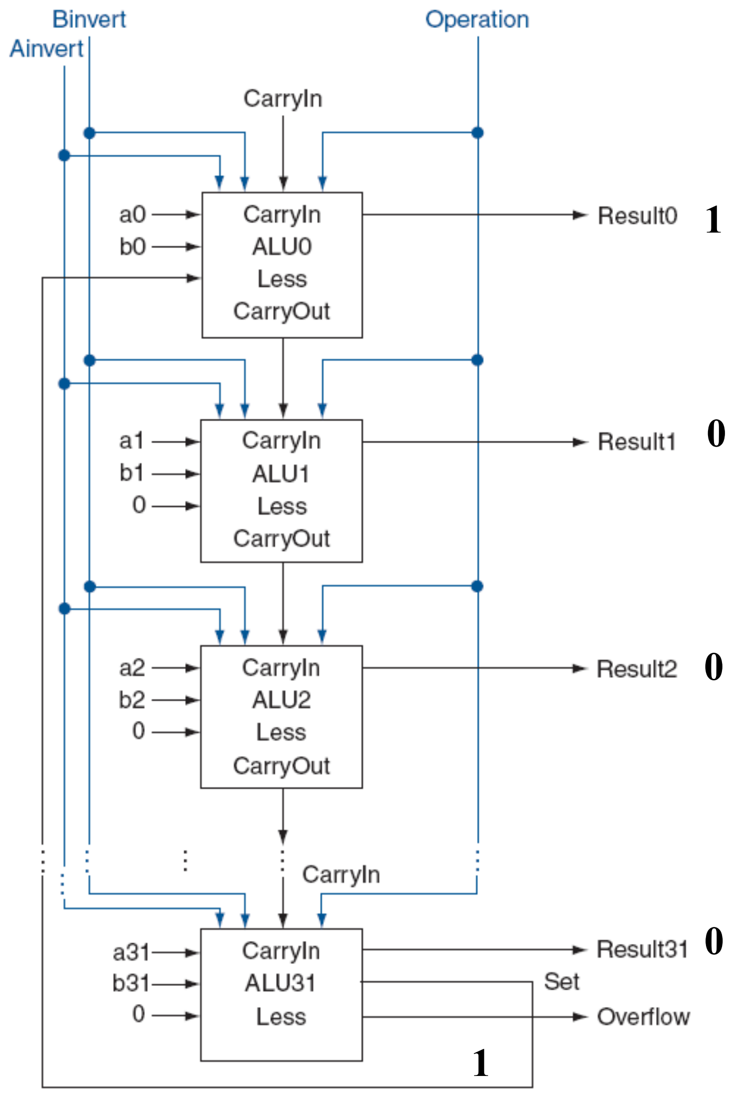
## Implementing SLT (set-less-than)

**SLT uses subtraction**
slt \$t0,\$t1,\$t2
\$t1<\$t2: \$t1-\$t2 gives negative result
set is 1 when negative

**Setting the control**
perform subtraction (Cin=1,Binvert=1)
select Less as output (Operation=3)
ALU31's Set connected to ALU0 Less

**Consider**
**Suppose \$t1=10 and \$t2=11**

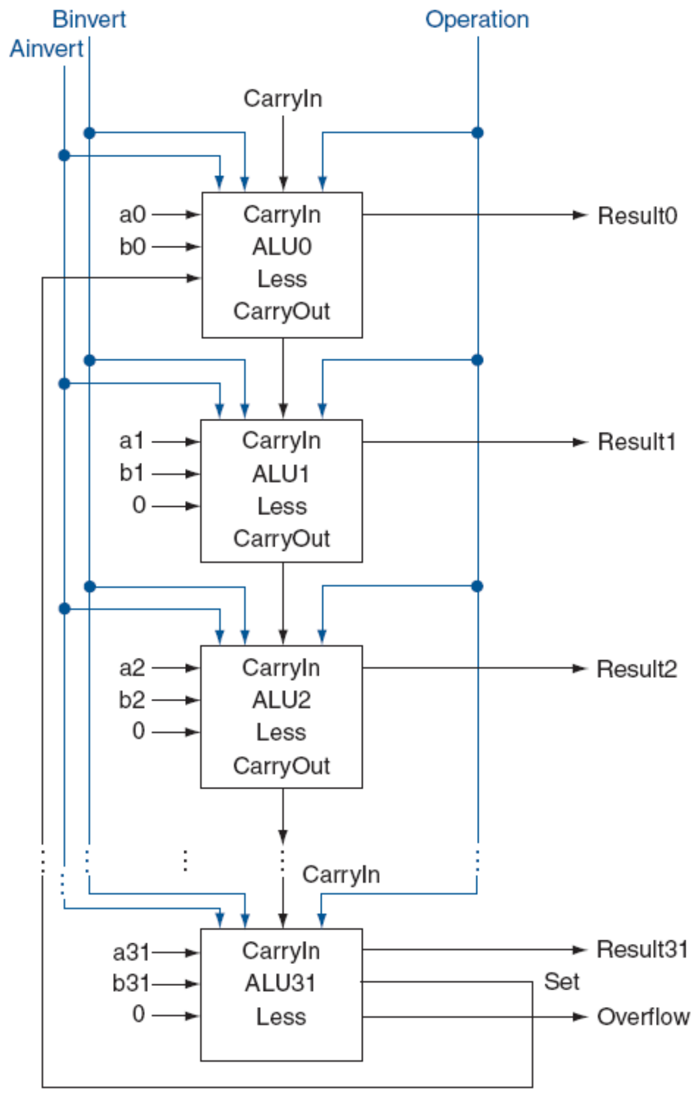
**\$t1 - \$t2 = -1 = 1111…1 binary**
**\$t0 = 0000...1**



# Implementing SLT (set-less-than)

**<u>SLT uses subtraction</u>**
slt \$t0,\$t1,\$t2
\$t1<\$t2: \$t1-\$t2 gives negative result
set is 1 when negative

**<u>Setting the control</u>**
perform subtraction (Cin=1,Binvert=1)
select Less as output (Operation=3)
ALU31's Set connected to ALU0 Less

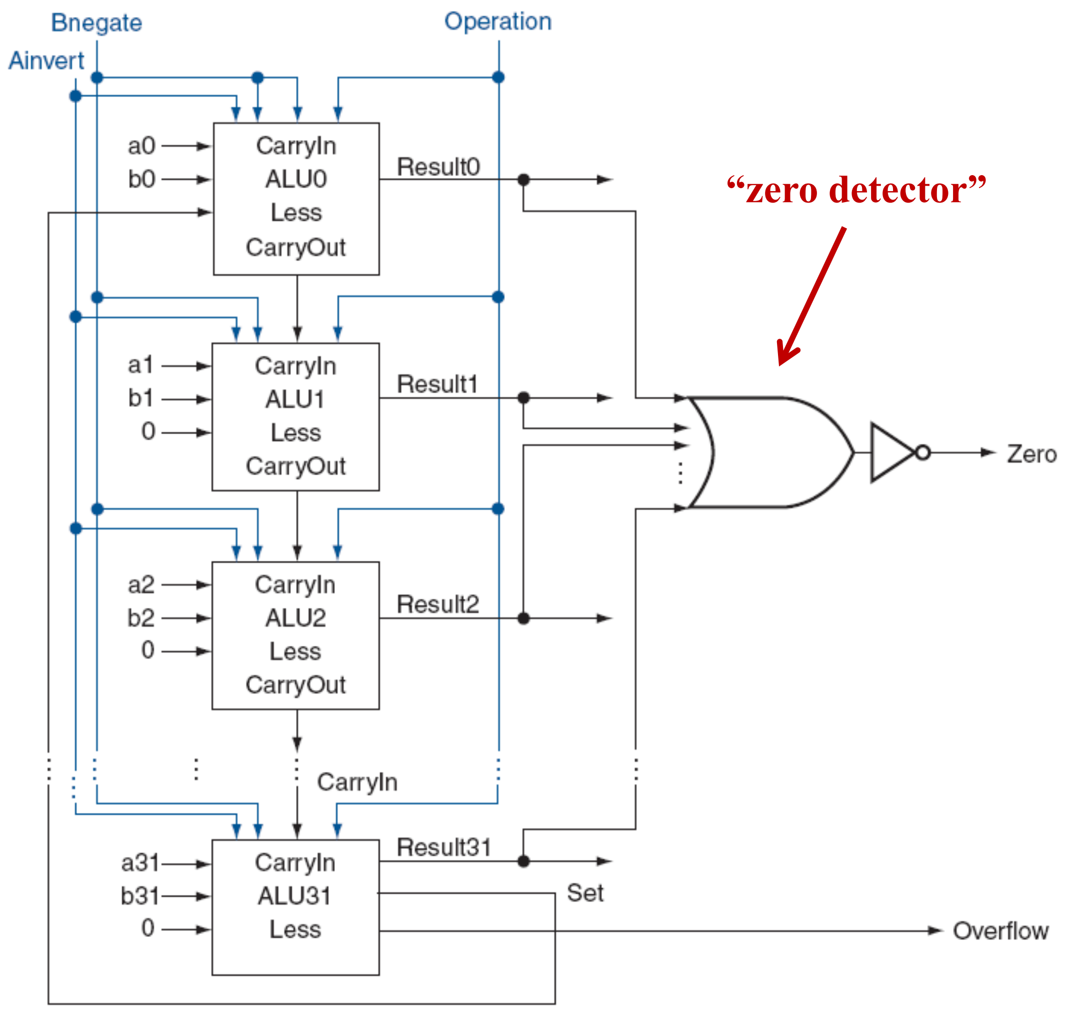


Why do we need Set? Could
we use just the Result31?

# Supporting BEQ and BNE

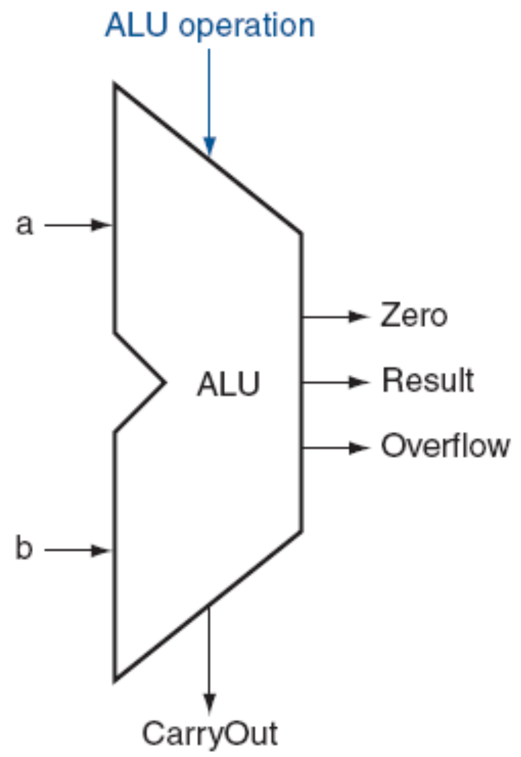
**<u>BEQ uses subtraction</u>**
beq $t0,$t1,LABEL
perform $t0-$t1
result=0 ➔ equality

**<u>Setting the control</u>**
subtract (Cin=1,Binvert=1)
select result (operation=2)
detect zero result



“zero detector”

## Abstracting ALU



- Note that ALU is a combinational logic