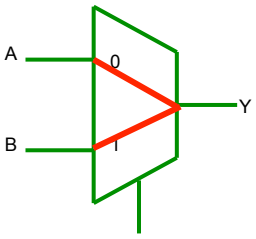
# Multiplexor (aka MUX)
# An example, yet VERY useful circuit!

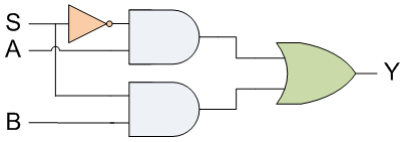



Y = (S) ? B:A;

when S =
0: output A
1: output B

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

Y=S'A+SB






# Simplifying expressions

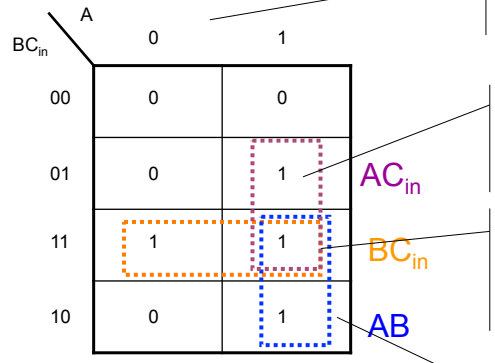
| Input | | | Output | |
|---|---|---|---|---|
| A | B | $C_{in}$ | S | $C_{out}$ |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

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

- Simplification reduces complexity: faster, smaller circuit!

# Karnaugh map

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




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

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

A 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

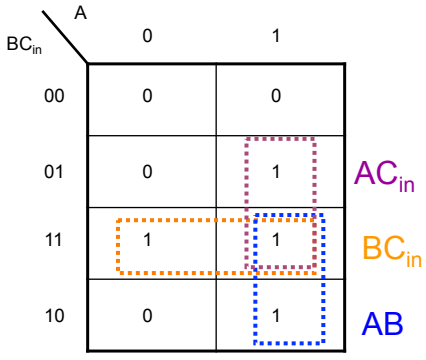
E.g.., $AB'C_{in}$ has a 1

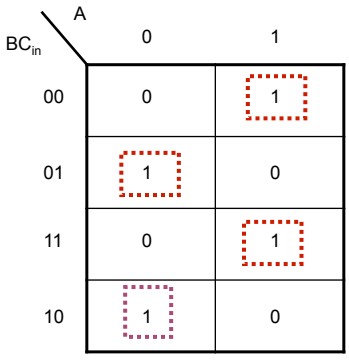
Now, to simplify:
Look for adjacent max rectangular groups with power of 2 elements.
In such a group, some var is {0,1}
Eliminate that variable

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



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




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

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

# Four (or more?) Variables

| AB \ CD | 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**

| AB \ CD | 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

| AB \ CD | 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'**

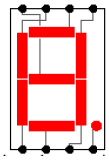
**These terms can now combine**
**Thus, we have B'D'**

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

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

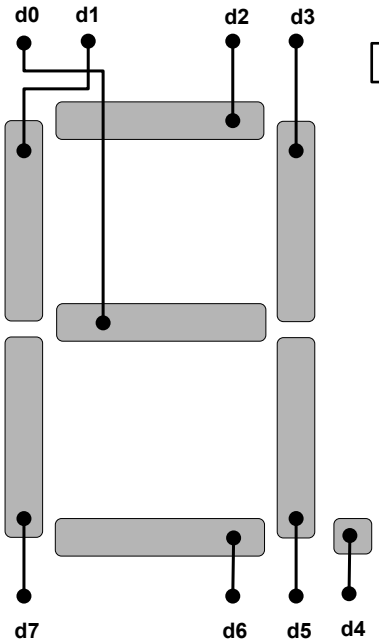
# In-class Example

- 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



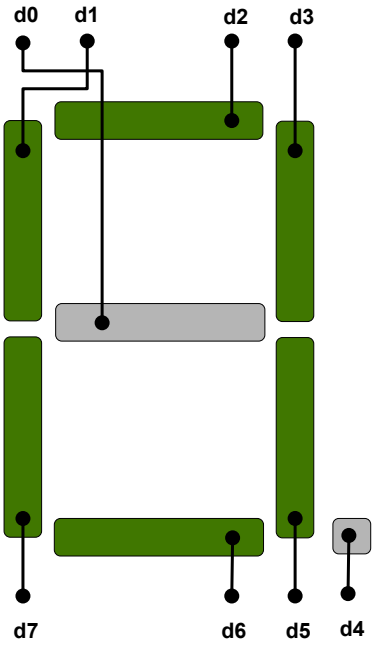
- **Problem**
  - Given a 3-bit number, draw the corresponding numeral
  - E.g., 000 is the numeral 0, 001 is numeral 1 and so forth
- **Solution**
  - Create a Boolean function for each segment. Ignore the decimal point.
  - Boolean function over three inputs for the 3-bit number.
- **Let’s try it!!**






**Segments numbered d0 to d7**

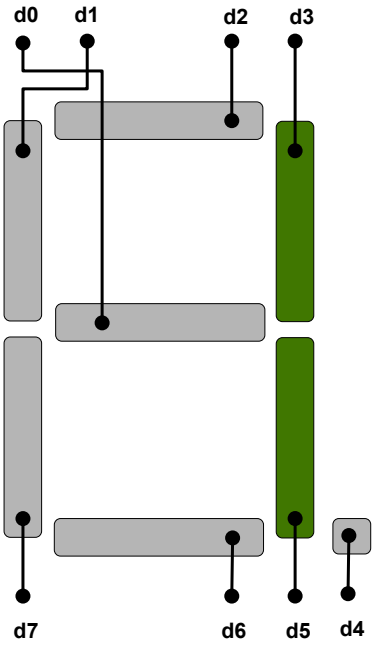
**Hex Digit LED**
**7 segments, 1 decimal point**
**Turn each segment on/off**
**State: 0=OFF, 1=ON**
**“Draw” numbers 0 to 9**




**Numeral 0**
**0 1 1 1 0 1 1 1**

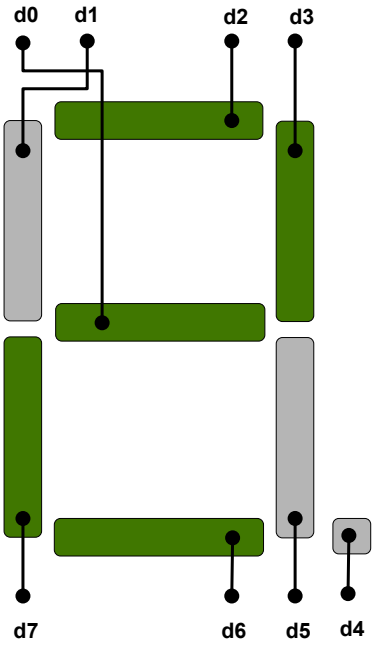
**Hex Digit LED**
**7 segments, 1 decimal point**
**Turn each segment on/off**
**State: 0=OFF, 1=ON**
**"Draw" numbers 0 to 9**






**Numeral 1**
**0 0 0 1 0 1 0 0**

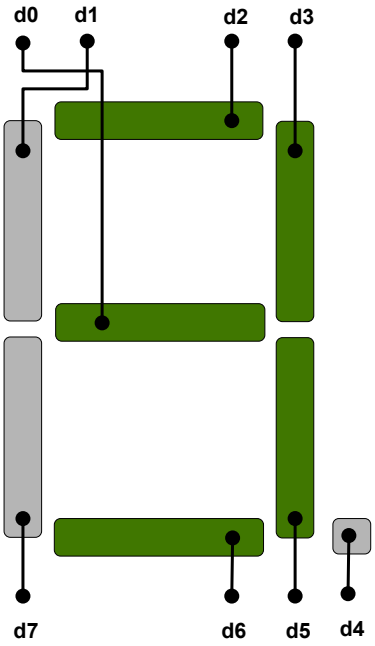
**Hex Digit LED**
**7 segments, 1 decimal point**
**Turn each segment on/off**
**State: 0=OFF, 1=ON**
**"Draw" numbers 0 to 9**




**Numeral 2**
**1 0 1 1 0 0 1 1**

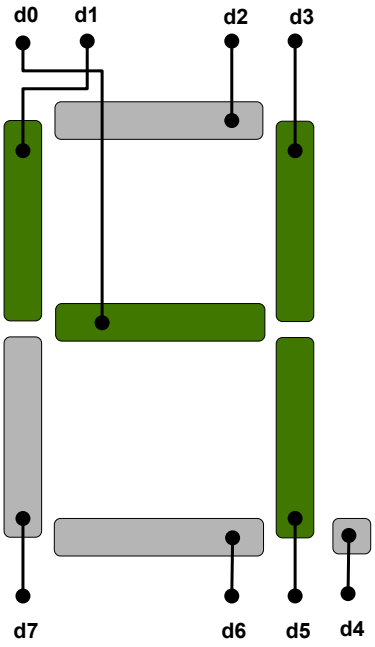
**Hex Digit LED**
**7 segments, 1 decimal point**
**Turn each segment on/off**
**State: 0=OFF, 1=ON**
**"Draw" numbers 0 to 9**






**Numeral 3**
**1 0 1 1 0 1 1 0**

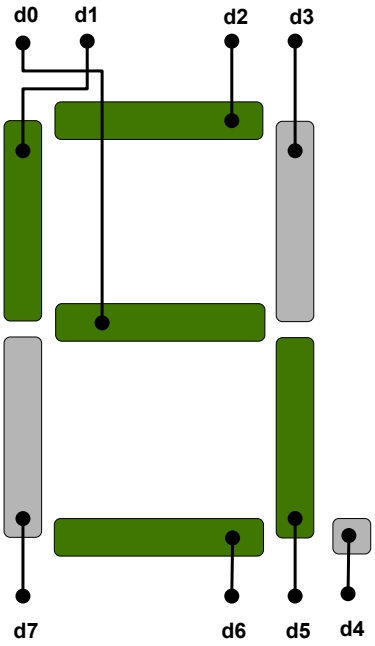
**Hex Digit LED**
**7 segments, 1 decimal point**
**Turn each segment on/off**
**State: 0=OFF, 1=ON**
**"Draw" numbers 0 to 9**




**Numeral 4**
**1 1 0 1 0 1 0 0**

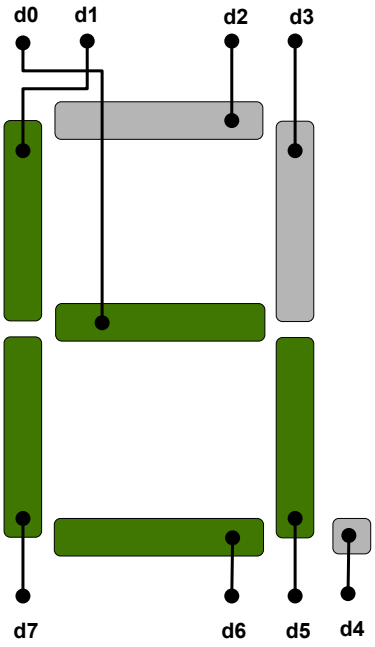
**Hex Digit LED**
**7 segments, 1 decimal point**
**Turn each segment on/off**
**State: 0=OFF, 1=ON**
**"Draw" numbers 0 to 9**






**Numeral 5**
**1 1 1 0 0 1 1 0**

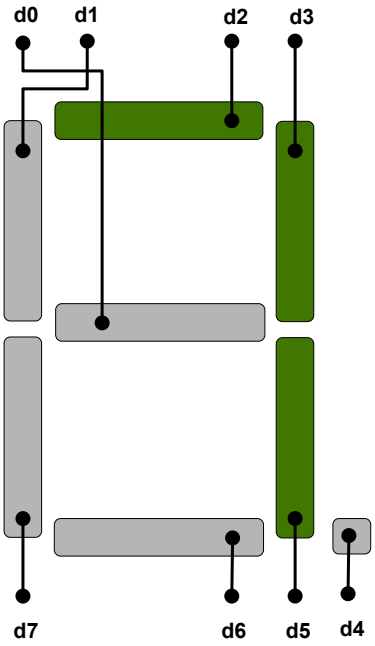
**Hex Digit LED**
**7 segments, 1 decimal point**
**Turn each segment on/off**
**State: 0=OFF, 1=ON**
**"Draw" numbers 0 to 9**




**Numeral 6**
**1 1 0 0 0 1 1 1**

**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**

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

| 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 |

**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**

| 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 |

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

**Function d5**

| i2 \ i1, i0 | 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 |



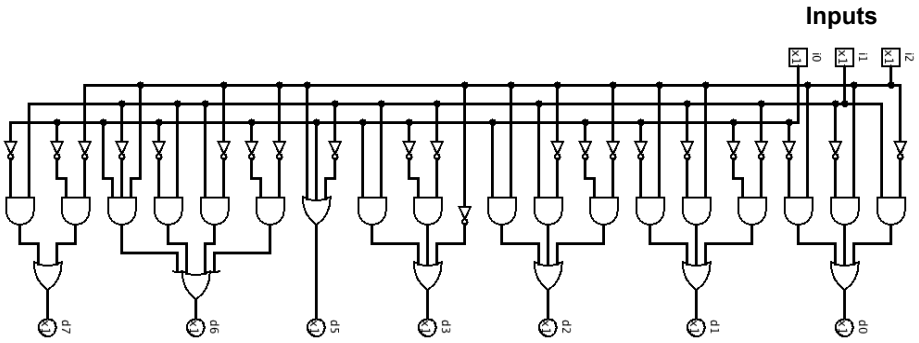
**Function d5**

| i2 \ i1, i0 | 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



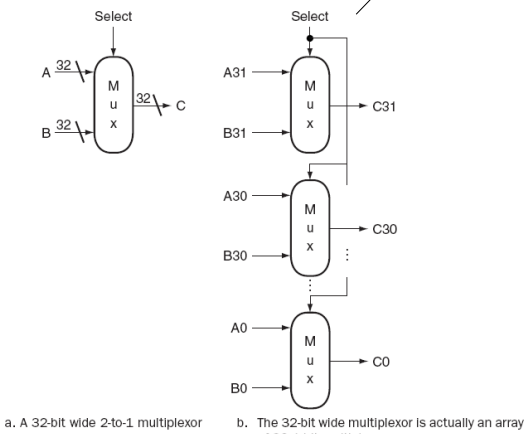

**Outputs to the LED hex digit**

**See example: LEDhexdigit.circ**



# A 32-bit MUX

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

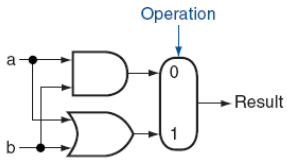


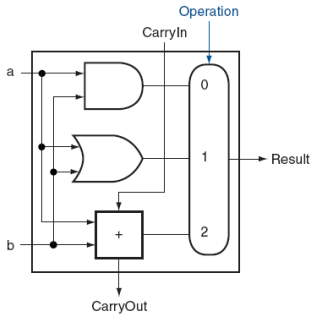

a. A 32-bit wide 2-to-1 multiplexor

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

# Building a 1-bit ALU

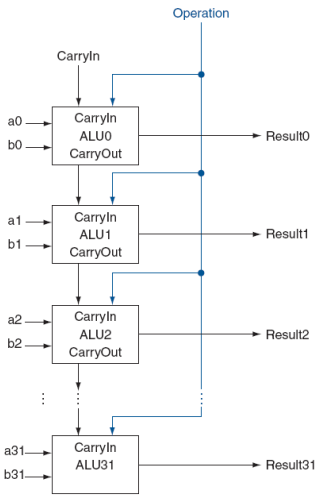
- ALU = arithmetic logic unit = arithmetic unit + logic unit



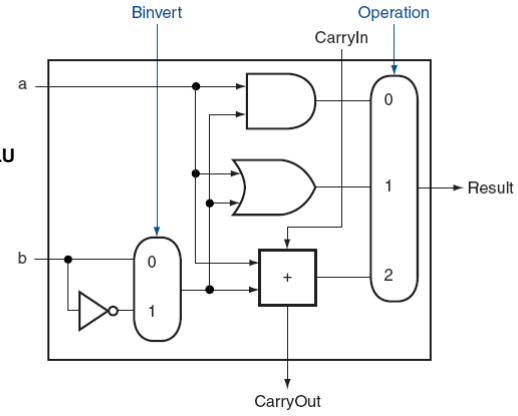



# Building a 32-bit ALU



# Implementing "sub"

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



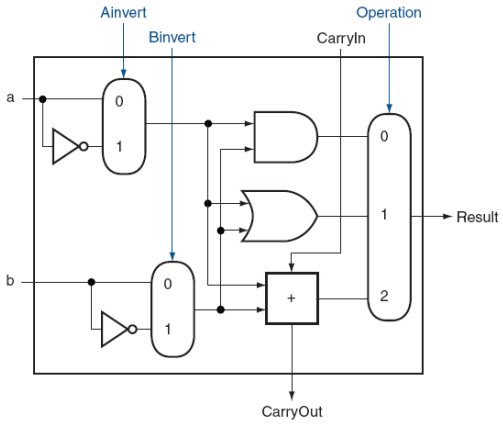



# Implementing NAND and NOR

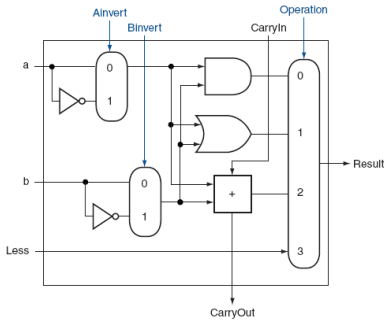
**NOR:**
**NOT (A OR B)**
**by DeMorgan's Law:**
**(NOT A) AND (NOT B)**

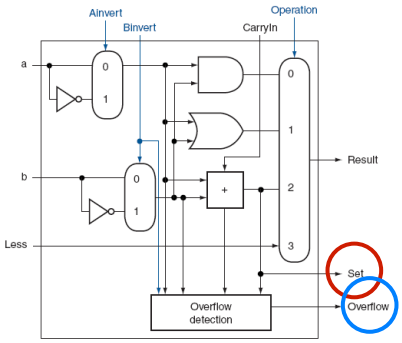
**Thus,**
**Operation=0,**
**Ainvert=1,**
**Binvert=1**

**And, NAND???**



# Implementing SLT (set-less-than)




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

**1-bit ALU for bit 31**

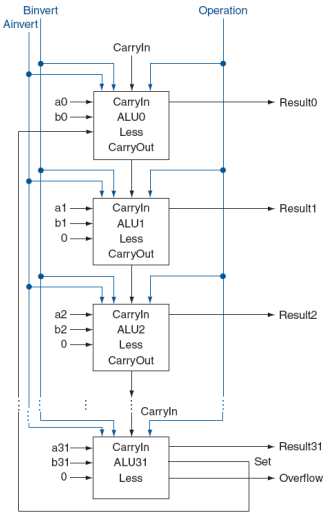


# Implementing SLT (set-less-than)

**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

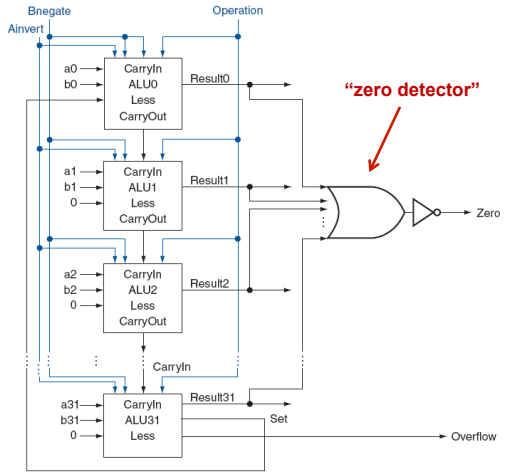
Why do we need Set? Could we use just the Result31?



# Supporting BEQ and BNE

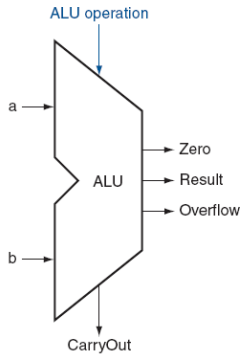
**BEQ uses subtraction**
beq $t0,$t1,LABEL
perform $t0-$t1
result=0 ➔ equality

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






# Abstracting ALU




- Note that ALU is a combinational logic