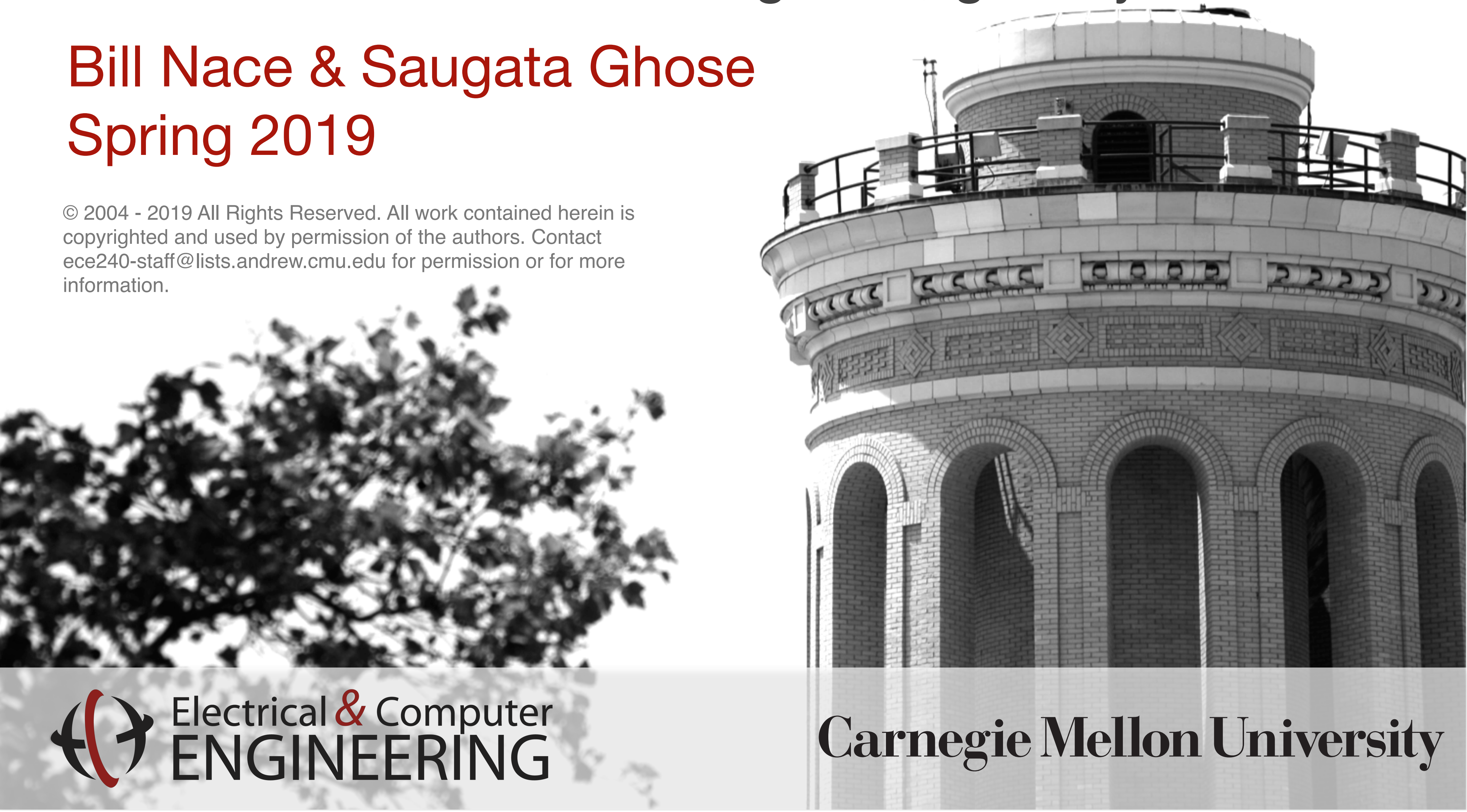
L6: Muxes and other Components

18-240: Structure and Design of Digital Systems

Bill Nace & Saugata Ghose

Spring 2019



Electrical & Computer ENGINEERING

Carnegie Mellon University

18-240: *Where* are we...?

- 1 Handout: Lecture Notes
- Midterm 1 -- the countdown begins

Week	Date	Lecture	Reading	Lab	HW
3	1/29	L4 Automated Logic Minimization		Lab 1	HW 2
	1/31	L5 Synthesizable SystemVerilog	LDVUS 2 - 2.3 HH 4.2-4.3		
	2/1	R2 Recitation			
4	2/5	L6 Combinational Components	HH 2.8, 4.5	Lab 2	HW 3
	2/7	L7 Numbers and Arithmetic	HH 1.4, 5.1-5.3		
	2.8	R3 Recitation			
5	2/12	L8 Comb. Logic Wrap-up	LDVUS 2.7 HH 1.7, 2.5-2.6	Lab 3	HW 4
	2/14	L9 Flip-flops and FSMs	HH 3.4		
	2/15	R4 Recitation			
	2/19	**Midterm 1**		No Lab	HW 5
	2/21	L10 FSM Design	LDVUS 3 - 3.3 HH 3.4, 4.4		
	2/22	R5 Recitation			

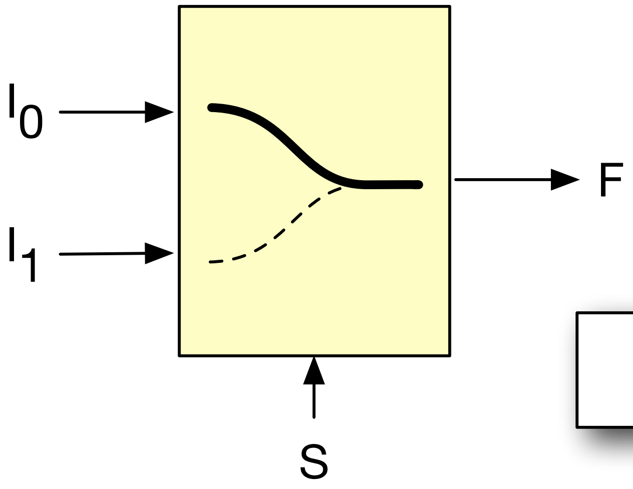
Today: Structured Logic Realization

- What you know so far…
  - Kmaps and 2-level SOP for smallish (random) logic functions
  - QM can be used too — or a computer tool based on QM
- What you don't know…
  - Other ways to approach large designs
  - Partitioning a design into smaller, more manageable chunks
  - Using pre-designed components
    - as parts of larger designs
    - as configurable devices
- Today
  - Multiplexers (and factoring big Boolean expressions)
  - Decoders (and more factoring…)
  - Other useful combinational components: Comparators, Demultiplexers

Multiplexers

Multiplexers (aka Mux)

- Consider 1-bit 2:1 Multiplexer
  - Truth table is given
  - Can you write Boolean equation for output?

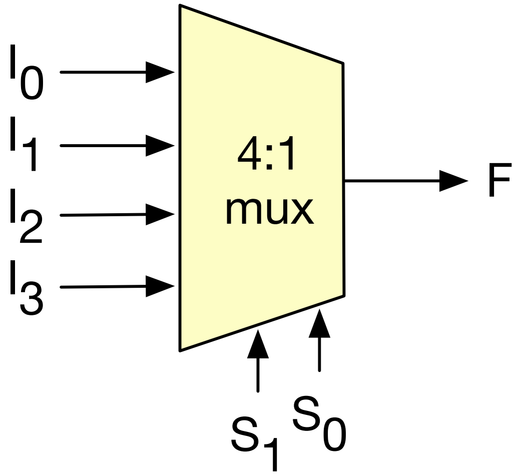


S	I_1	I_0	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

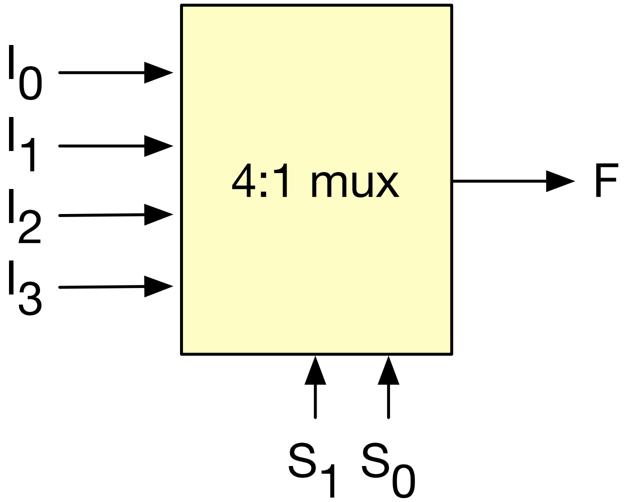
Mission:

Multiplexers: Drawing Conventions

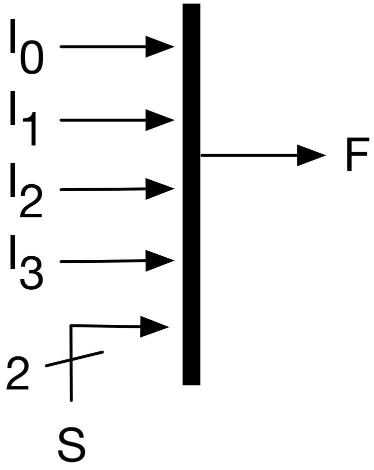
Normal



Plain

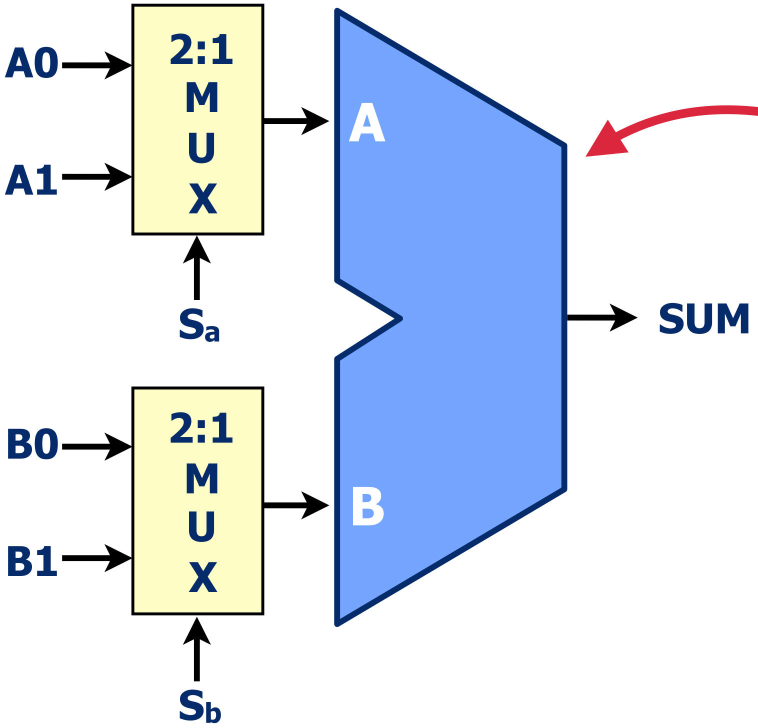


Minimalist



- Specifications
  - Size: how many different inputs (2^n) can be selected?
  - Bit-Width: how many bits wide is each input?
- In general
  - 2^n data inputs, n select lines (also called control inputs), 1 output
    - ✦ a 2^n:1 or “2^n by 1” mux
  - Select lines form binary index used to choose (select) an input

Multiplexers: Example usage



A shared "functional unit," in this case, a binary adder

Depending on Sa and Sb, we can add: A0+B0, A0+B1, A1+B0, or A1+B1

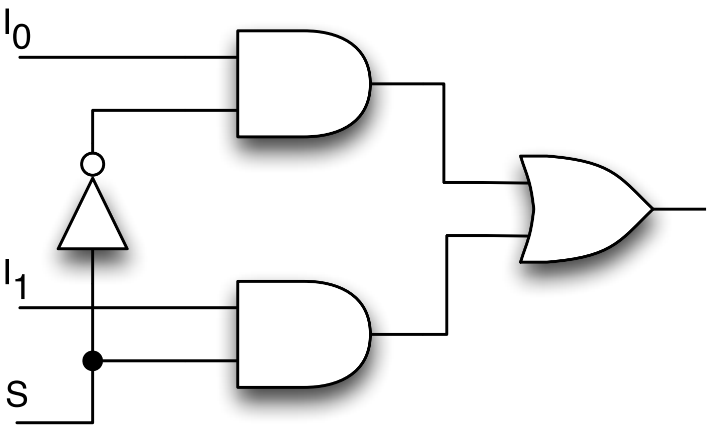
If inputs (A0...B1) are 16-bit variables, then the muxes are 16-bit, 2:1 muxes

Multiplexers: Gate-level

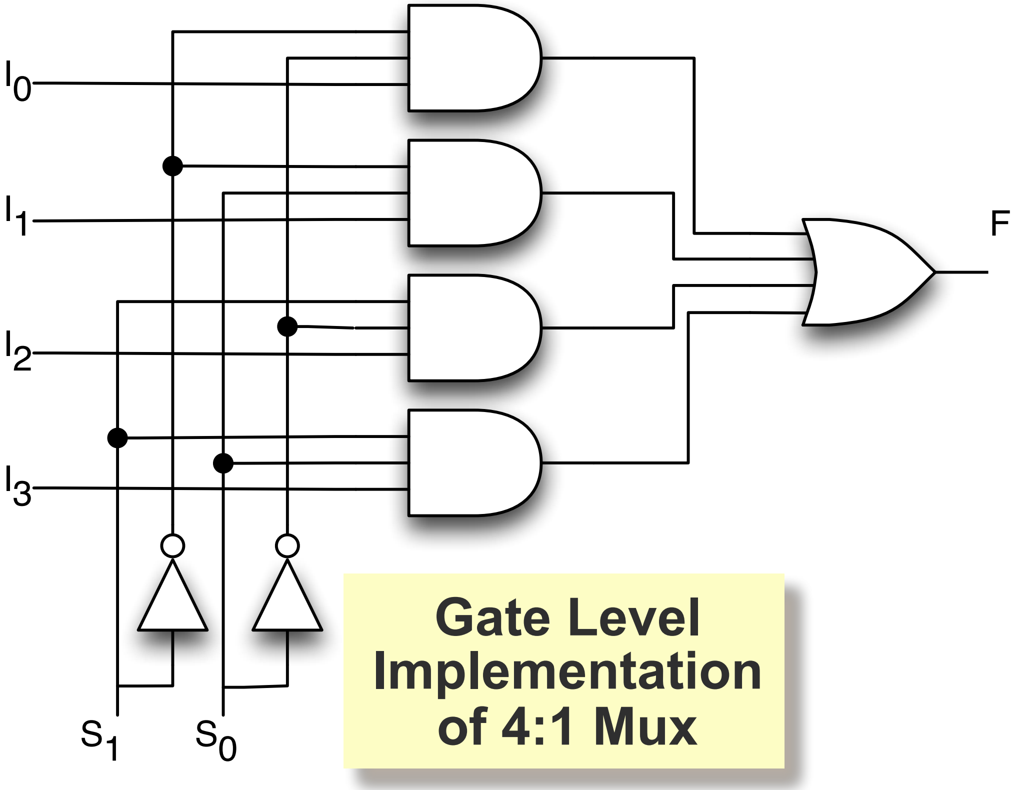
- What's inside the Mux? gate-level implementation?
  - All Muxes have a very stylized, easy to recognize form

$F = S'\,I_0 \;+\; S\,I_1$

$F = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$

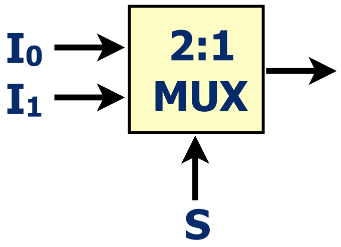


Gate Level Implementation of 2:1 Mux

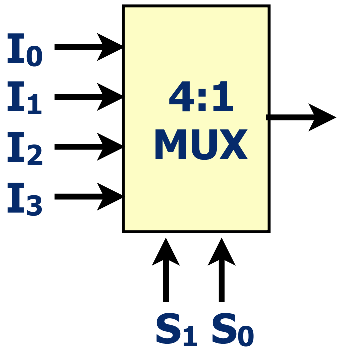


Gate Level Implementation of 4:1 Mux

Multiplexers: Ditto for Bigger Ones…



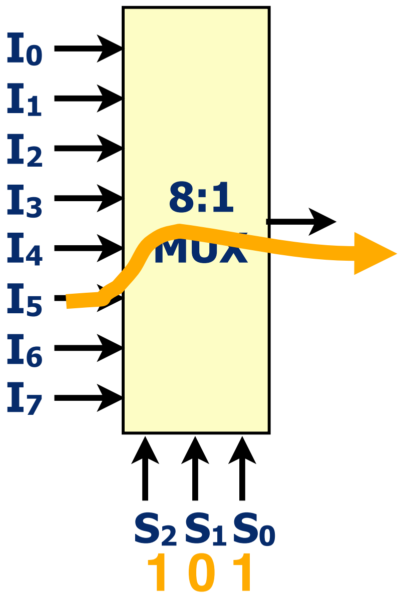
$F = S' I_0 + S I_1$



$F = S_1'S_0' I_0 + S_1'S_0 I_1 + S_1 S_0' I_2 + S_1 S_0 I_3$

Whoa! There’s a minterm expansion in each of these expressions

The minterms of the *select* input(s) are generated



$1 \cdot I_5$

$$F = S_2'S_1'S_0'I_0 + S_2'S_1'S_0I_1 + S_2'S_1S_0'I_2 + S_2'S_1S_0I_3 + S_2S_1'S_0'I_4 + S_2S_1'S_0I_5 + S_2S_1 S_0'I_6 + S_2S_1S_0I_7$$

Carnegie Mellon

Multiplexers: Use as Logic

- 2^n:1 mux can implement n-variable function (easy)
  - Just a lookup table; put TT row k value on k-th mux input

Example: F(A,B,C) = m0 + m2 + m6 + m7

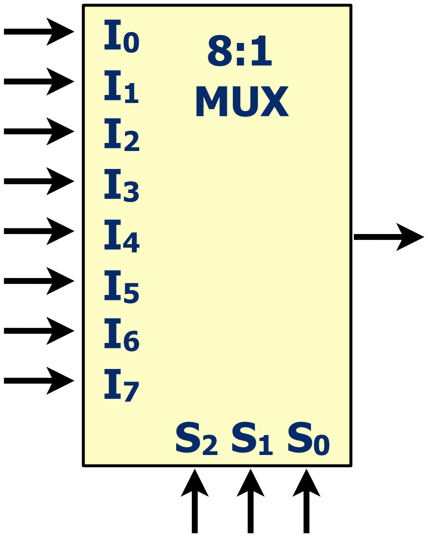
= A' B' C' + A' B C' + A B C' + A B C

= A' B' C' (1) + A' B' C (0) + A' B C' (1) + A' B C (0)

+ A B' C' (0) + A B' C (0) + A B C' (1) + A B C (1)

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

"Lookup Table"



Put Functions on the Inputs

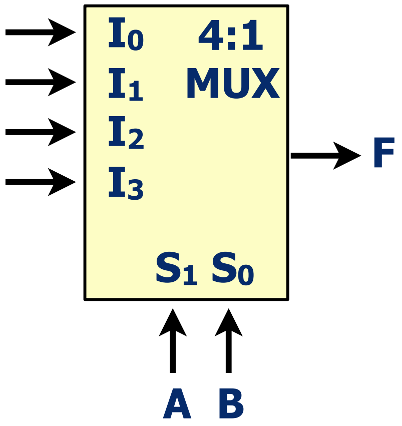
- 2^{n-1}:1 mux can implement n-variable function (subtle)
  - Still a lookup table; but each mux input is now a *function*

Example: $F(A,B,C) = m0 + m2 + m6 + m7$

$$= A'\,B'\,C' + A'\,B\,C' + A\,B\,C' + A\,B\,C$$

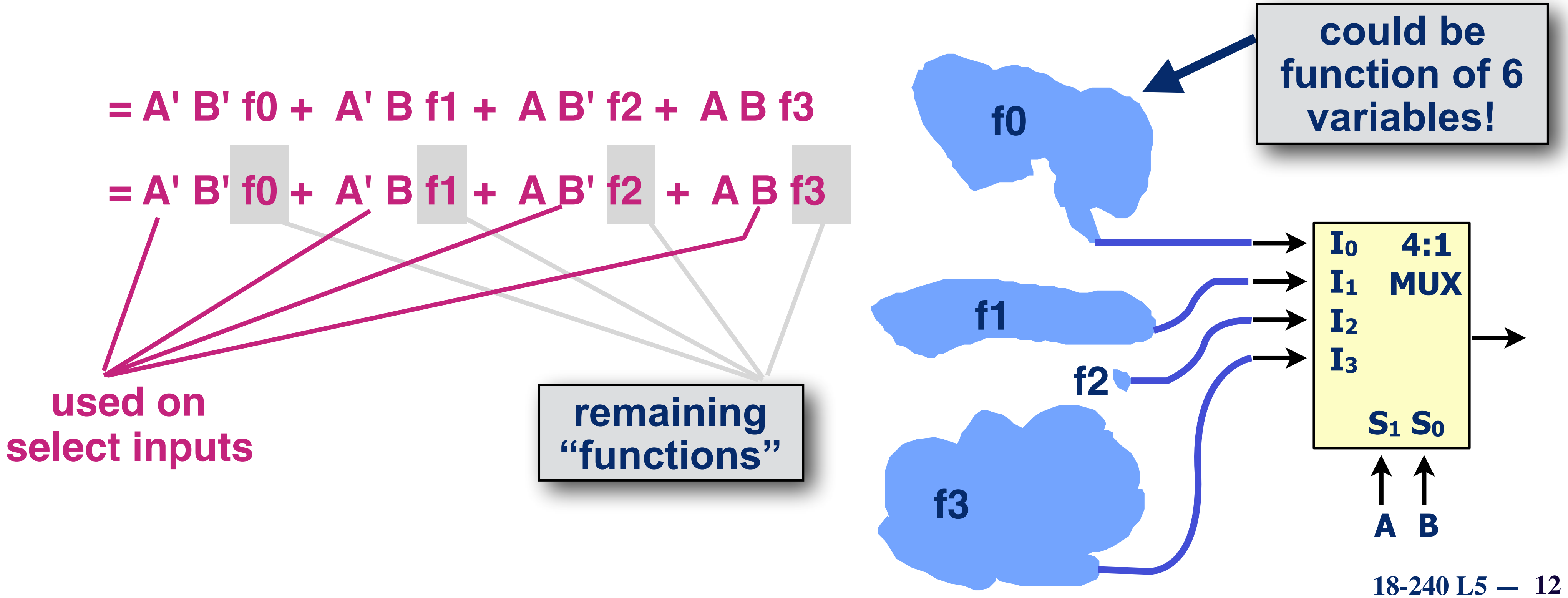
$$= A'\,B'\,(C') + A'\,B\,(C') + A\,B'\,(0) + A\,B\,(1)$$

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



Why is this useful?

- Big functions
  - Input C from the previous slide could be a bigger function
  - Suggests a method for implementing functions of many variables
  - That is, factor the function so that
    - ✦ some of the variables are select inputs to a mux
    - ✦ the remaining functions are data inputs to the mux



$= A' B' f0 + A' B f1 + A B' f2 + A B f3$

$= A' B' f0 + A' B f1 + A B' f2 + A B f3$

used on select inputs

remaining “functions”

could be function of 6 variables!

Shannon Expansion

- So, how do we *factor* the eqn?
  - Suppose we have a function $F(x_1, x_2, \ldots, x_n)$
  - We define a new function if we set one of the x_i=constant
  - Example: $F(x_1, x_2, \ldots, x_i=1, \ldots, x_n)$
  - Example: $F(x_1, x_2, \ldots, x_i=0, \ldots, x_n)$
- Easy to do by hand

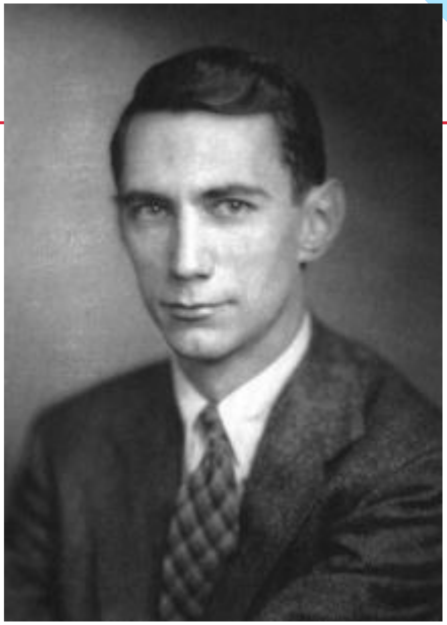
$F(a,b,c) = ab + ac' + b(a'c + c')$

$F(a=1,b,c) =$ □

$F(a=0,b,c) =$ □

- Important to remember that result is a ***new function***
  - Note that new function no longer depends on chosen variable

Shannon Expansion



- Turns out to be incredibly useful idea
  - These new functions are called Shannon cofactors (or just cofactors) of the original function
  - Shannon Cofactor with respect to x_i
    - ✦ Write $F(x_1, x_2, \ldots, x_i=1, \ldots x_n)$ as F_{xi}
    - ✦ Write $F(x_1, x_2, \ldots, x_i=0, \ldots x_n)$ as $F_{xi'}$
    - ✦ Also can write as just $F(x_i=1)$ $F(x_i=0)$ (which is easier to type)
- Shannon Expansion Theorem
  - Given any Boolean function $F(x_1, x_2, \ldots, x_n)$ and any x_i in the variables F( ) depends on,

F( ) can be represented as

$F(x_1, x_2, \ldots, x_n) =$ □

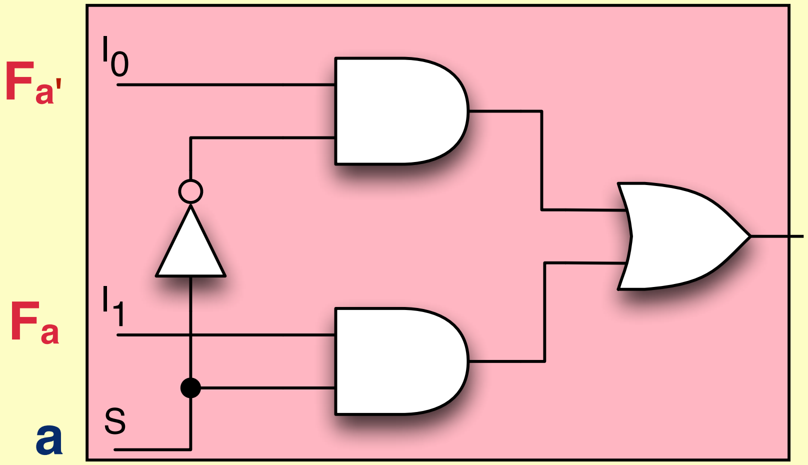
What this looks like

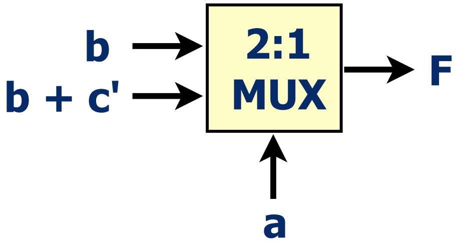
- For F(a,b,c) = ab + ac' + b(a'c + c')

$F(a=0) = F_{a'} = b$
$F(a=1) = F_{a} = b + c' + bc'$
$= b + c'$

Putting it back

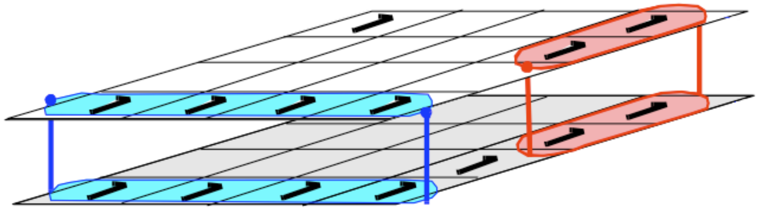
$F = a'F_{a'} + aF_{a}$
$= a'(b)+a(b+c')$
$= a'b+ab+ac'$
$=b+ac'$ (min sop)
$=b(1+c')+ac'$
$=b+bc'+ac'$
$=b(a+a')+bc'+ac'$
$=ab+a'b+bc'+ac'$
$=ab+ac'+b(a'+c')$
$=ab+ac'+b(a'c+c')$

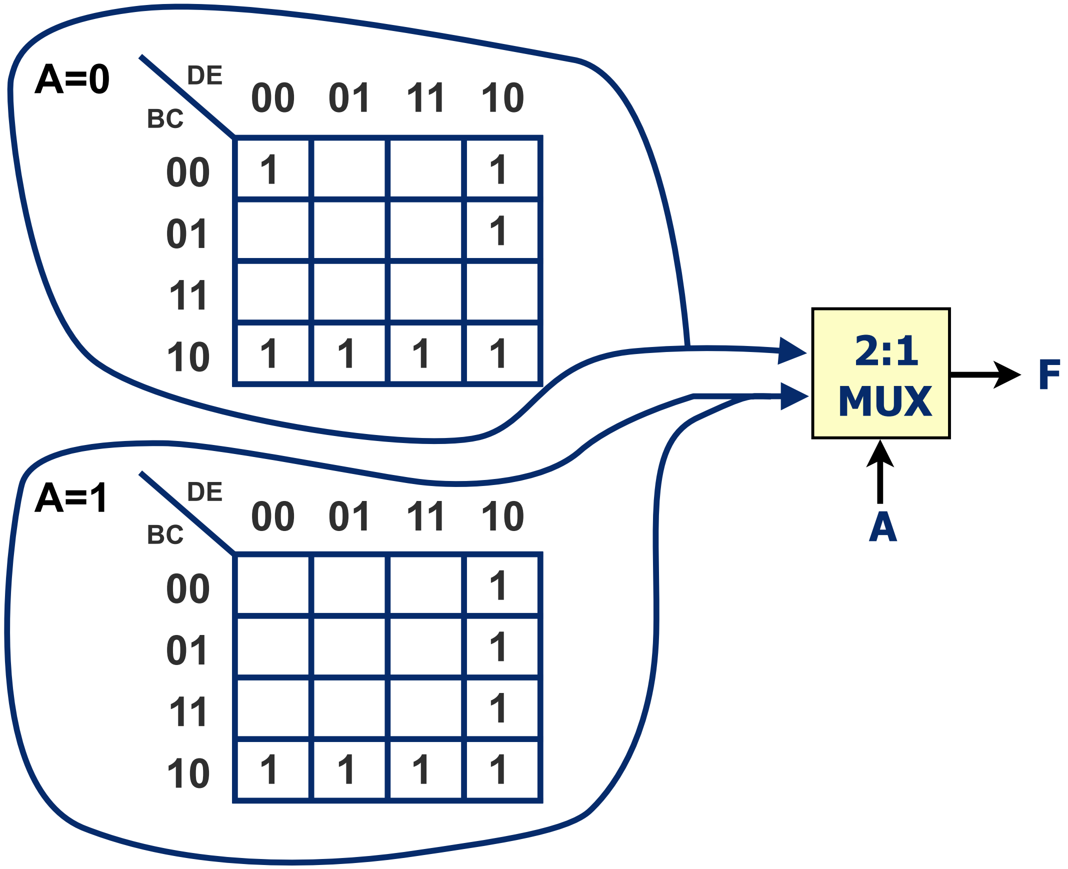




Another view

- Cofactors of this 5-variable function
  - Clearly, it's important to pick the variable to cofactor around. Why? (What if all prime implicants are independent of the chosen variable)
  - If we chose A, then draw kmap with A as the first variable so F_A and $F_{A'}$ are the top and bottom Kmaps





A=0

BC \ DE	00	01	11	10
00	1			1
01				1
11				
10	1	1	1	1

A=1

BC \ DE	00	01	11	10
00				1
01				1
11				1
10	1	1	1	1

Which variable to factor?

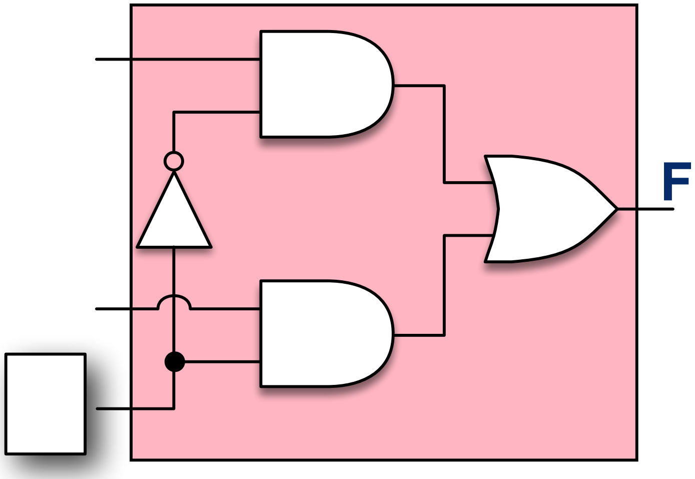
- The factoring variables affect complexity
  - A heuristic: pick a variable that shows up a lot in both true and complement form to zero out product terms in the expansion
- Example…

$F = b + a'c'd + acd'$

If you pick a to factor around:

$F_a =$

$F_{a'} =$



Which variable to factor?

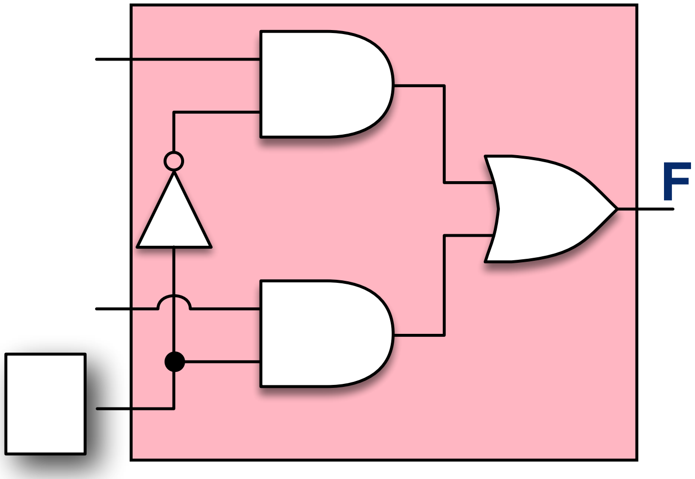
- Same example…

$F = b + a'c'd + acd'$

If you pick b to factor around:

$F_b =$ □

$F_{b'} =$ □



Advanced Boolean Algebra

- You can do this for more than one variable at a time
  - Shannon Cofactor with respect to x_i and x_j
    - ✦ Write F(x_1, x_2, …, x_i=1, …, x_j=0, …, x_n) as $F_{xi\ xj'}$
    - ✦ Ditto for any number of variables x_i, x_j, x_k, …
  - Now, the expansion is around the 4 combinations of the 2 variables
    - ✦ Before we wrote:

      $F(x_1, x_2, \ldots, x_n) = x_i \cdot F_{xi} + x_i' \cdot F_{xi'}$

    - ✦ Now, we can write:

      $F(x,y,z,w) =$ □

    - ✦ Note that F_{xy} = F(x=1, y=1, z, w) is a Boolean function of z and w
  - We've removed 2 variables
    - ✦ could remove even more

Cofactor Summary

- Why is this cool?
  - We have taken a *big* function and factored it into *smaller* ones
  - The standard engineering approach of divide and conquer
- These examples are way too small
  - You only get some feel of how the Boolean functions split up
  - And, what you might do if confronted with:
    - ✦ a **big** function, especially when good factorization exist
    - ✦ no CAD tools
- Left to another course (18-760)
  - Detail, insight, expansion about what we just did
    - ✦ A recursive approach that just keeps factoring the remaining functions until the resulting cofactor is something reasonable
    - ✦ Which variables do you expand around?

2:1 Mux: SystemVerilog

- You’ve seen this before

```
module multiplexer
 (input  logic sel, i0, i1,
  output logic f);

  assign f = (sel) ? i1 : i0;

endmodule : multiplexer
```

```
module multiplexer
 (input  logic sel, i0, i1,
  output logic f);

  always_comb begin
    f = i0;
    if (sel)
      f = i1;
  end

endmodule : multiplexer
```

- Also, multi-bit mux

```
module multiplexer
 (input  logic       sel,
  input  logic [7:0] i0, i1,
  output logic [7:0] f);

  assign f = (sel) ? i1 : i0;

endmodule : multiplexer
```

Still 2:1, just bigger sized items to choose from

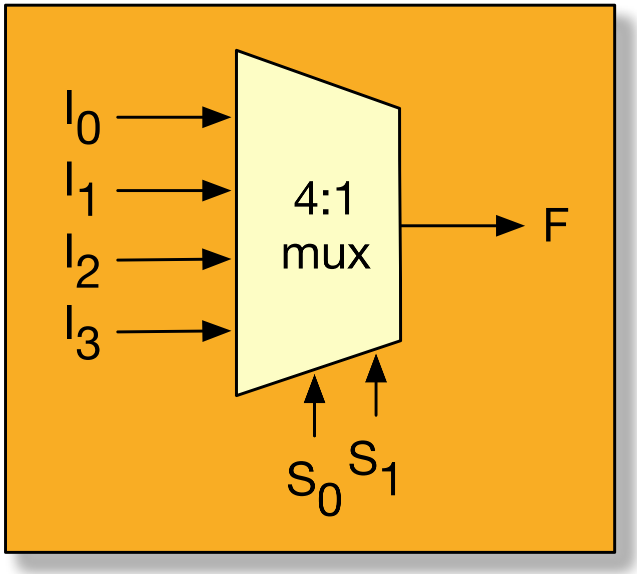
2^{sel}:1 Mux: SystemVerilog

- Choice can also be over more inputs

```
module multiplexer_4to1
 (input  logic [1:0] sel,
  input  logic [3:0] i,
  output logic f);

  always_comb
    case (sel)
      2'b00 : f = i[0];
      2'b01 : f = i[1];
      2'b10 : f = i[2];
      default : f = i[3];
    endcase

endmodule : multiplexer_4to1
```

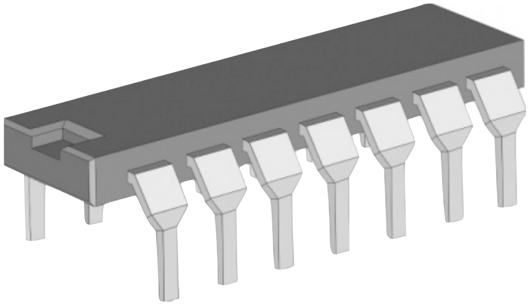


Gonna get tiring for many inputs

There must be a better way!

Standard Components

- Mux is an example of a standard component
  - Used to make bigger designs
- Historically, available as a "chip" or integrated circuit (IC)
  - Small Scale ICs = SSI = ~ 10 gates/chip
    - ✦ AND, OR gates, etc
  - Medium Scale ICs = MSI = ~ 100 gates/chip
    - ✦ Comparators, Multiplexers, Decoders, Adders, ...
  - Large Scale ICs = LSI = ~ 1000-10,000 gates/chip
  - Very Large Scale ICs = VLSI = 100,000+ gates/chip

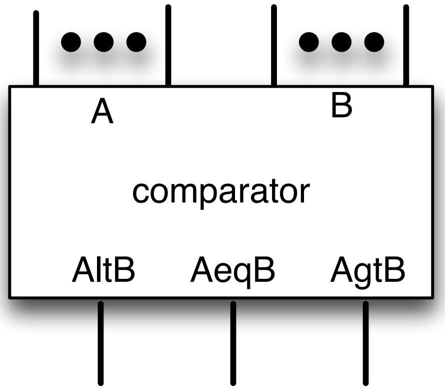


- Nowadays, components available in SystemVerilog libraries and FPGA elements
- Also useful to discuss functionality of designs

Comparators

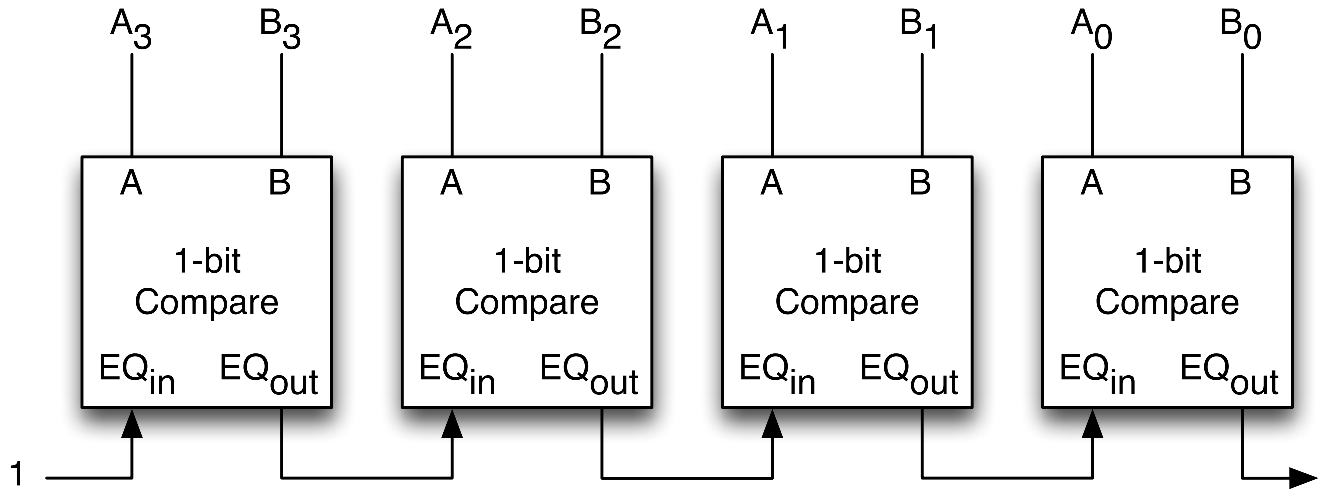
Comparator

- **Mission: Compare two inputs**
- **Inputs: unsigned binary numbers**
  - Signed versions also available
- **Outputs: Single bit signals**
  - Is A = B?
  - Is A < B? Is A > B? Available on "Magnitude Comparator"
- **1-bit comparator?**



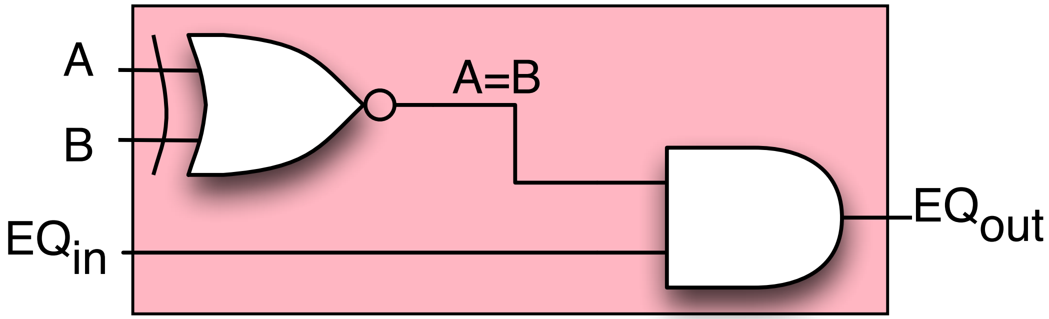
Multi-bit Comparator

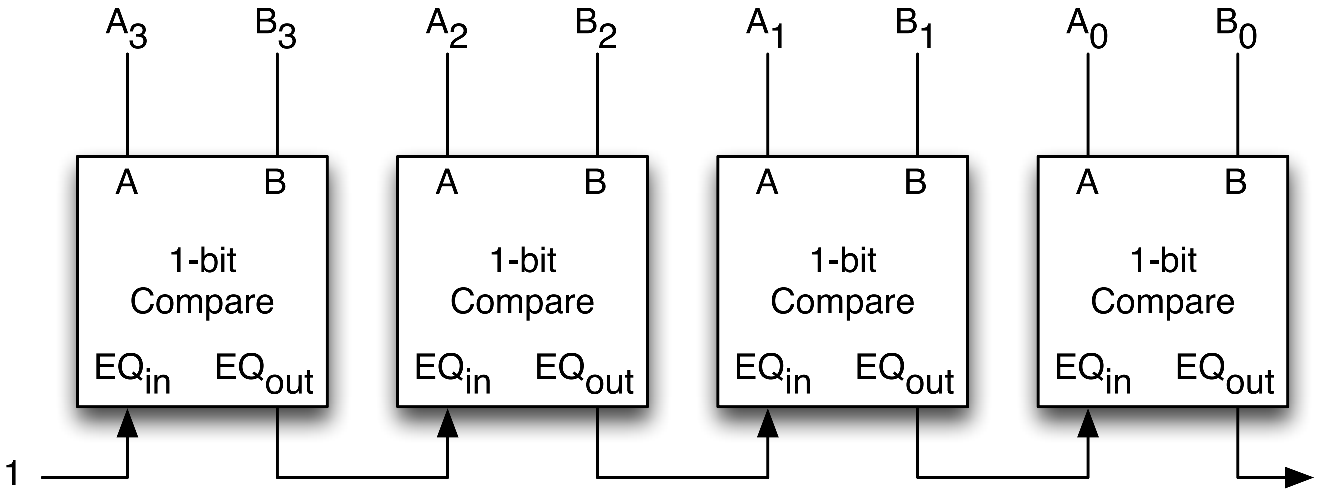
- How do you compare numbers?
- Are these equal? 568743
  568941
  - Start at the left and compare digits until you find a difference
- Iterative circuit
  - Repeated use of a basic module with signals cascading



What's in the box?

- The 1-bit comparison box needs to be able to cascade inputs
  - XNOR isn't good enough
  - EQ_{out} = 1 if (EQ_{in} = 1) and (A = B)



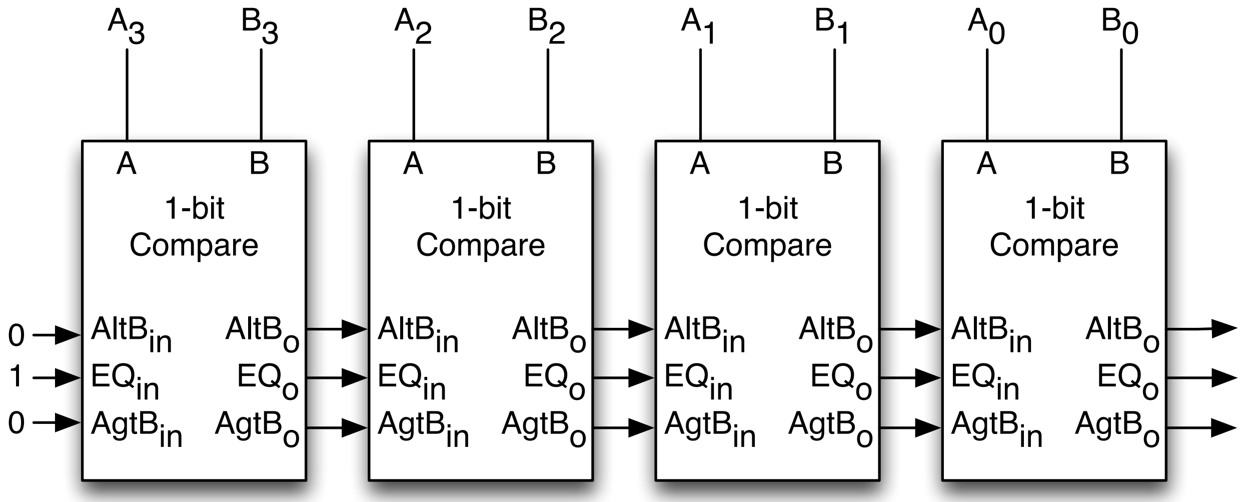


Carnegie Mellon

Multi-bit Magnitude Comparator

- How do you compare numbers?
  - Which is larger? 568743
568941
  - Start at the left and compare digits until you find a difference

AgtB	AEqB	AltB	A	B	AgtB	AeqB	AltB
1	0	0	X	X	1	0	0
0	0	1	X	X	0	0	1
0	1	0	0	0	0	1	0
0	1	0	0	1	0	0	1
0	1	0	1	0	1	0	0
0	1	0	1	1	0	1	0



Bit Slicing

- This design method where you solve a problem for one-bit at a time is very useful
- You can replicate the solution n-times for an n-bit circuit
  - No need to re-design / re-solve / re-debug just because you want a 17-bit comparator
- We will see this bit-slice design approach in several other circuits, primarily arithmetic

Comparator: SystemVerilog

- Again, SV operator does this for you

```
module comparator(
  input  logic [7:0] A, B,
  output logic       AeqB);

  assign AeqB = (A == B);

endmodule : comparator
```

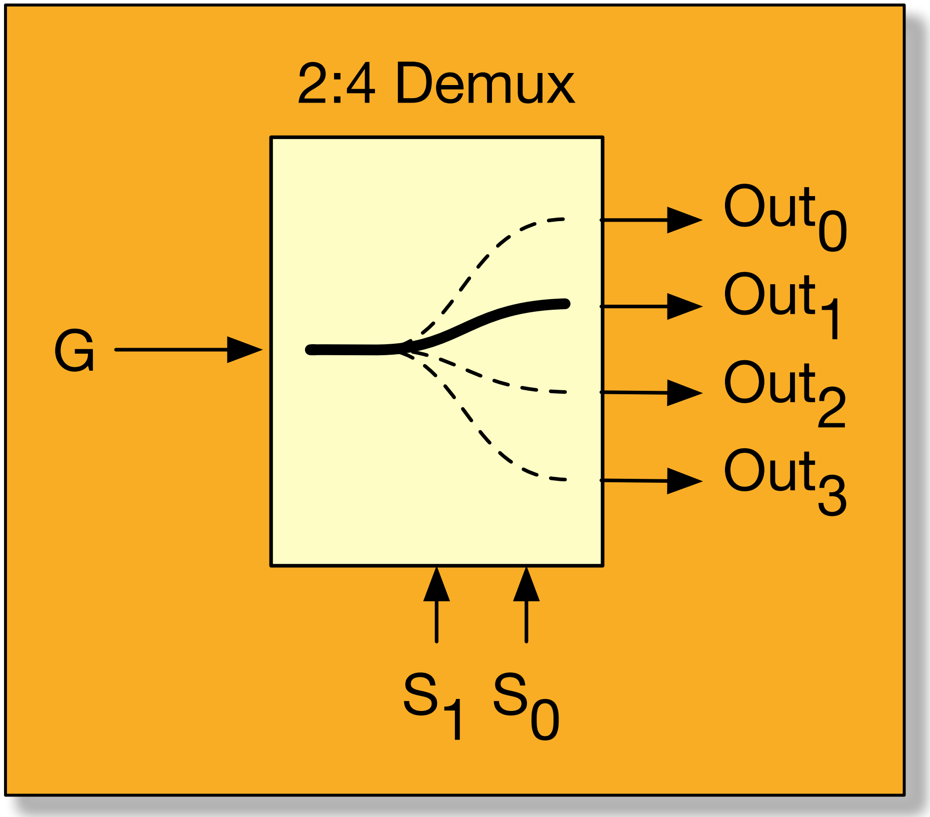
- Magnitude comparator uses comparison operators

```
 assign AltB = (A < B);
 assign AgtB = (A > B);
```

De-multiplexers and Decoders

Demultiplexer: Basic Idea

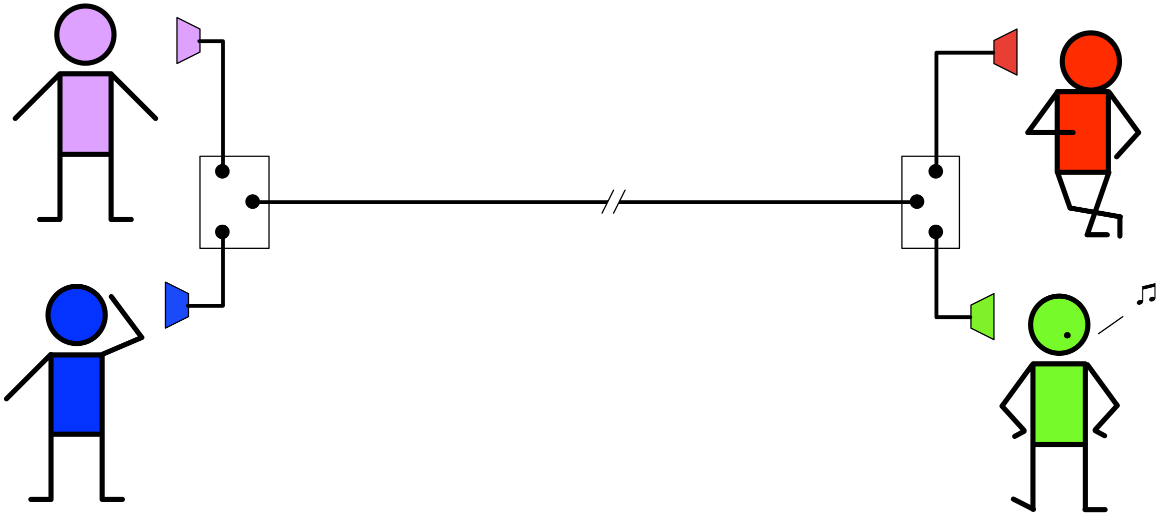
- Opposite of a Mux
- S1, S0 select one of 4 outputs to send G to



- What happens to the other 3 outputs?

Example Use

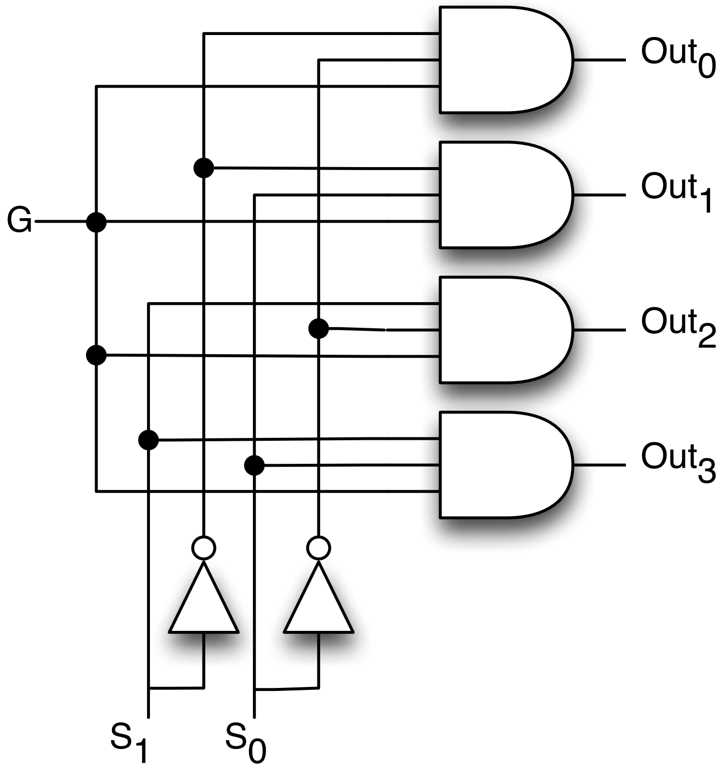
- Telecommunications: When paired with a Mux, can drastically reduce # wires needed to connect *m* sources to *m* receivers



- Can route data to one of *m* circuit elements
  - Often, more efficient to send data to all *m* circuit elements, and have *m*-1 of them ignore the input

Demux Internals

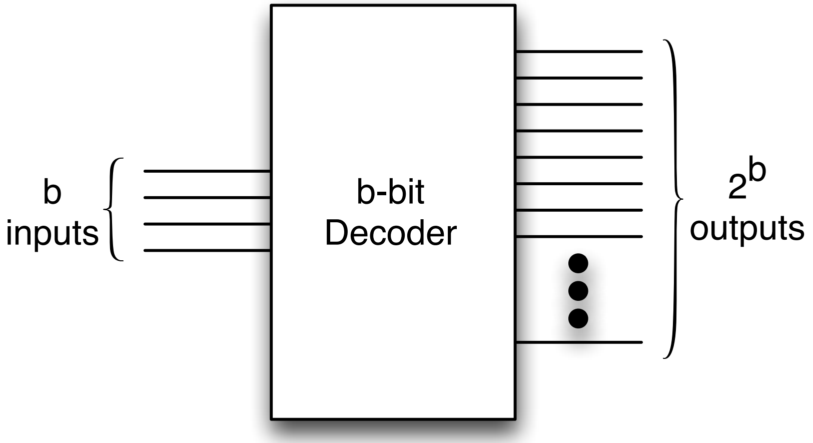
- Each output just needs to check if the select lines apply to it or not



S_1	S_0	G	Out_3	Out_2	Out_1	Out_0
0	0	0	0	0	0	0
0	0	1	0	0	0	1
0	1	0	0	0	0	0
0	1	1	0	0	1	0
1	0	0	0	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	0	0
1	1	1	1	0	0	0

Decoder

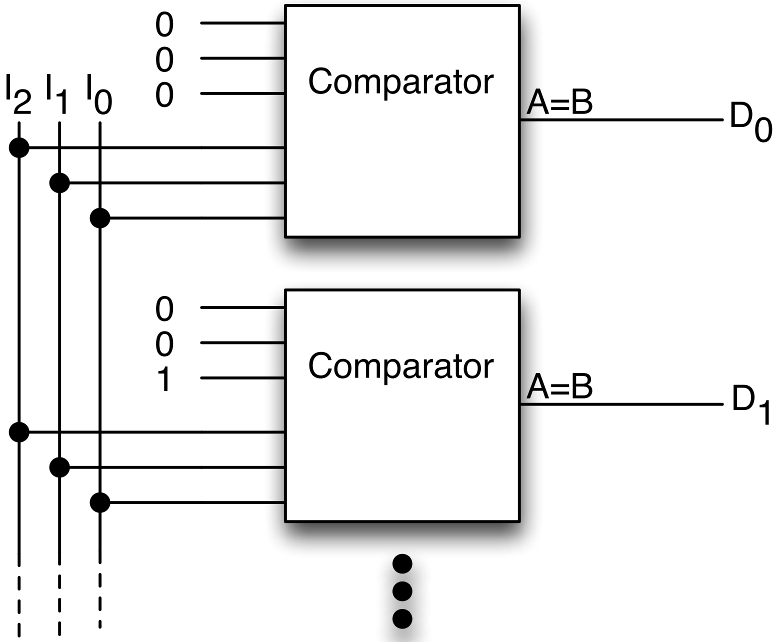
- Mission: Convert one code to another
- Default: Convert binary to "one's hot"
  - One's Hot: only a single output is 1 ("hot") all other are zero
- BCD-to-7Segment Decoder is another example that you will see in Lab 2



I_2	I_1	I_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

Decoder Internals

- **How do you build a decoder?**
  - **Isn't each output active only when the input bits match the code?**



- **This is a perfectly good, "component-based" way of thinking about building circuits**
  - **But, it's wrong**
  - **Too much abstraction, not enough optimization**

Decoder Internals, another try

- Examine the Truth table

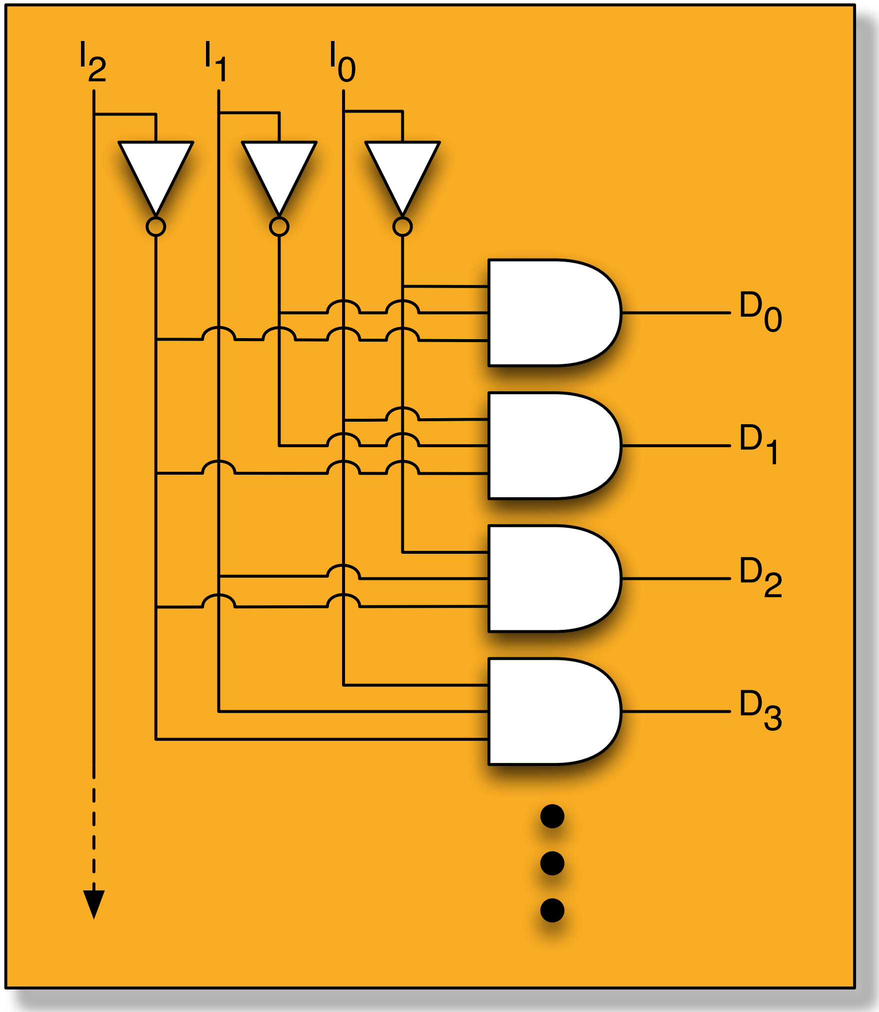
I_2	I_1	I_0	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

- What is D_0?
- D_1?

A b-bit decoder is 2^b AND gates, each with b inputs

Decoder Internal Gate Structure

- Each output of the decoder is an AND gate connected to some combination of the inputs (and their complemented versions)



Decoder Enable

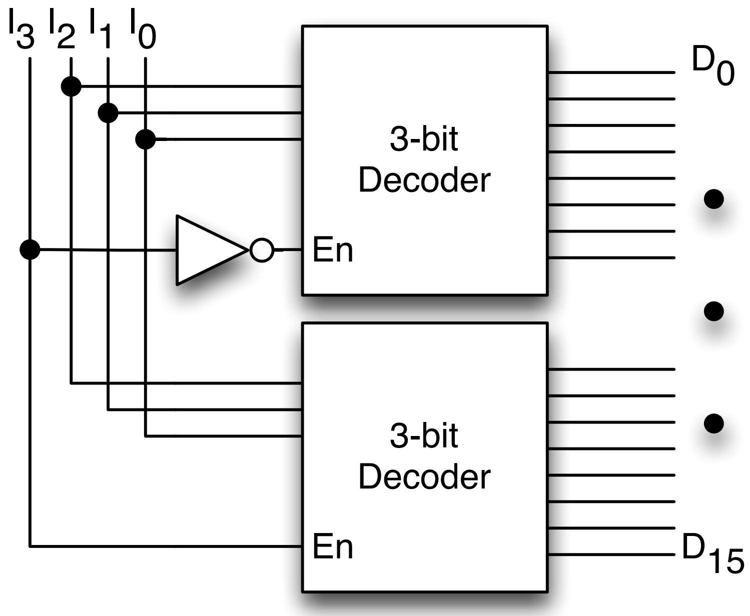
- ***Enable*** **on a component lets you control when it should do its job**
  - **… and when you want it to shut up**

I_2	I_1	I_0	En	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	1	0	0	0	0	0	0	0	1
0	0	1	1	0	0	0	0	0	0	1	0
0	1	0	1	0	0	0	0	0	1	0	0
0	1	1	1	0	0	0	0	1	0	0	0
1	0	0	1	0	0	0	1	0	0	0	0
1	0	1	1	0	0	1	0	0	0	0	0
1	1	0	1	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0
X	X	X	0	0	0	0	0	0	0	0	0

- **Decoder output is always a zero when not enabled**

Cascading Decoders

- Enable lets you build big decoders out of little ones



Decoder vs. Demux

- BTW, you've seen Decoder w/Enable before

I_1	I_0	En	D_3	D_2	D_1	D_0
0	0	1	0	0	0	1
0	1	1	0	0	1	0
1	0	1	0	1	0	0
1	1	1	1	0	0	0
X	X	0	0	0	0	0

2-bit Decoder w/Enable

S_1	S_0	G	Out_3	Out_2	Out_1	Out_0
0	0	0	0	0	0	0
0	0	1	0	0	0	1
0	1	0	0	0	0	0
0	1	1	0	0	1	0
1	0	0	0	0	0	0
1	0	1	0	1	0	0
1	1	0	0	0	0	0
1	1	1	1	0	0	0

2-bit Demultiplexer

Component Summary

- Multiplexer (Mux)
  - Mission: Select an input to put on the output
- Demultiplexer (Demux)
  - Mission: Choose an output to get the input
  - Opposite of a Multiplexer (duh, Demux)
- Comparator
  - Mission: Decide if two multi-bit inputs are the same
  - Magnitude comparator also does “less-than” and “greater-than”
  - Bit-slice or "cascade" design technique
- Decoders
  - Mission: Convert binary to one-hot code
  - Sometimes convert other codes
  - Some have “enable” input so we can build big from little

Summary

- Lots of useful components
  - A standard library of combinational circuits
- Decoders and Multiplexers are essential components of higher level designs
  - Computer architecture is littered with them
- Later in the course, we will find ourselves using these components as building blocks for RTL designs