



Lab 3 Tutorial

What this lab requires

- More practice with Verilog, and some new Verilog syntax.
 - Learn how devices like multiplexers can be made in multiple ways.
- Implementing some logical devices.
 - Full adder circuit, ALU.
- Learning some hierarchical design.
 - Mux + adders = ALU.

Exploring Verilog

- Verilog can express logic beyond the basic gate-by-gate specification.
 - Typically in `always` blocks
- Example: `case` statements
 - Not the same as case statements in other languages!
 - **case** statements in Verilog provide output behaviour for all possible input values
 - Like specifying the output for all minterm cases
 - How would you do a multiplexer with this?

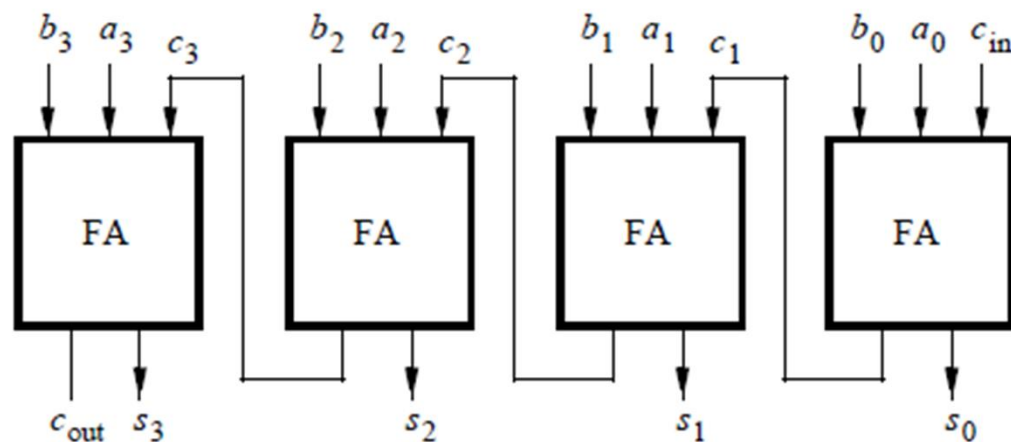
Part 1: Mux + case statement

- Use `case` statement and `always` block to implement a 7-to-1 multiplexer.
 - See the example of a `hex_decoder` from class.
- Key points:
 - Combination circuits w/ an `always` block.
 - Use `*` in the **sensitivity list** of your `always` block.
 - Don't forget the **default** case! Or else the tool will need to have memory (which we haven't done yet).
 - New storage term: **reg** (used similar to `wire`)
 - Use `wire` with assignment statement outside `always` blocks, and `reg` within them.

Part 2: Ripple Carry Adders

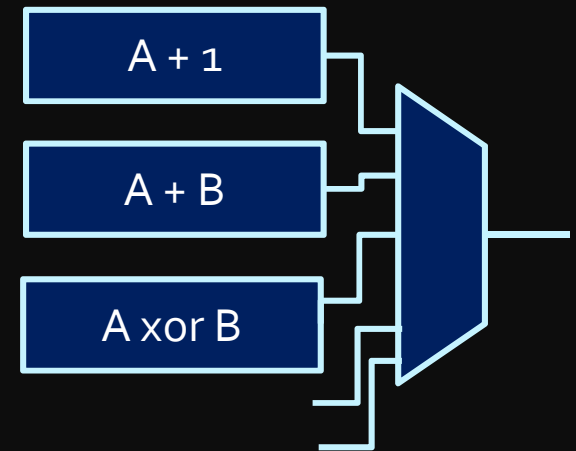
- Implement a Ripple Carry Adder by connecting (chaining) four full-adders together.
 - Must use hierarchical design!

b	a	c_i	$c_o \ s$	
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



Part 3: ALU

- The ALU (Arithmetic Logic Unit) uses a mux to choose a single output value from a series of input values.
 - At the end of each input wire is the output wire from a logical device that performs an arithmetic operation.
 - Remember how primitive gate design worked (back when we first introduced Verilog), when assembling this hierarchical design.
- Once implemented with general inputs A and B , connect this simple ALU to the switches for input and the LEDs and 7-segment display (in hex) for output.
 - Recommended that you reuse the modules you made in Lab 2.
 - The handout includes syntax for some new (and potentially useful) operators



Some New Verilog Operators

Verilog Operators

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	multiply	two
	/	divide	two
	+	add	two
	-	subtract	two
	%	modulus	two
	**	power (exponent)	two
Logical	!	logical negation	one
	&&	logical and	two
		logical or	two
Relational	>	greater than	two
	<	less than	two
	>=	greater than or equal	two
	<=	less than or equal	two
Equality	==	equality	two
	!=	Inequality	Two
		case equality	two
		case inequality	two

Bitwise Operators

- **Bitwise** Operators (see Table 1 from Lab2)
 - If you use a bitwise operator with two n-bit operands, the result is also an n-bit vector.

Table 1: Verilog Operators

	bitwise OR
&	bitwise AND
~	bitwise negation
^	bitwise XOR

- For example:
 - $3'b101 \& 3'b011 \rightarrow 3'b001$
- More general mathematical notation:
 - $(X_{n-1}X_{n-2} \dots X_1X_0 \& Y_{n-1}Y_{n-2} \dots Y_1Y_0)$ results in $W_{n-1}W_{n-2} \dots W_1W_0$ where W_i is $(X_i \& Y_i)$ for every i in $[0, n-1]$.
 - You can use any of these bitwise operators in place of **&**.

Reduction Operators

- **Reduction Operators** have same symbol as bitwise, but
 - they take **a single** multi-bit operand, and
 - they result in a **single-bit vector**.
- For example:
 - $(\& \ 3'b101)$ results in $1'b0$
- General mathematical notation:
 - $(\& \ X_{n-1}X_{n-2} \dots X_1X_0)$ results in W_0 where W_0 is $(X_{n-1} \ \& \ X_{n-2} \ \& \ \dots \ \& \ X_1 \ \& \ X_0)$.
 - Think of this as feeding all n bits of X into a single n -input AND gate with output W_0
 - You can use any of the reduction operators in place of $\&$.

Replication and Concatenation

- The binary value 011 (3 in decimal) is the same as 0011 or 000000011.
 - Adding zeros in the most significant bits of a positive or an unsigned number does NOT change the number being represented!
- Example:
 - If the output of a module is 3-bits and you want to feed it to a 5-bit input of another module, you'd need to use both replication & concatenation!
 - See Background section on Lab3 handout.

More Complex Force Commands

```
force {<signal_name>} <initial value> <initial time>, <new value> <new time>  
-repeat <repeat time> -cancel <cancel time>
```

```
force {a} 0 0, 1 20 -repeat 40  
force {b} 0 0 ns, 1 40 ns -r 80
```

- What's happening here:
 - The waveform for *a* (see first *force* command) starts low (logic-0) @ 0ns
 - At 20ns, it goes high (changes to logic-1).
 - It stays high until 40ns when this process repeats
 - (i.e., it goes back to 0 until 60ns; at 60ns it goes back to 1 until 80ns and so on..)
 - *a* is represented by a square wave with a period of 40ns
 - The square waveform for *b* has double the period of *a*.
- Don't forget to follow these force commands with **a run command** (e.g., run 160ns)!

Declaring a vector (of wires):

- What you might be tempted to do:
 - `wire multibit_vector [3:0];`
- The syntax you should use:
 - `wire [3:0] multibit_vector;`
- Additional notes:
 - You can specify a part of this vector's bits (a subset of wires) by writing: `multibit_vector[2:1]`