Lab 3 Tutorial

Exploring Verilog

- Verilog can express logic beyond the basic gate-by-gate specification.
  - Typically in `always` blocks
- Example: `case` statements make cases that are mutually exclusive
  - 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?

after compilation is same as previous implementation

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.

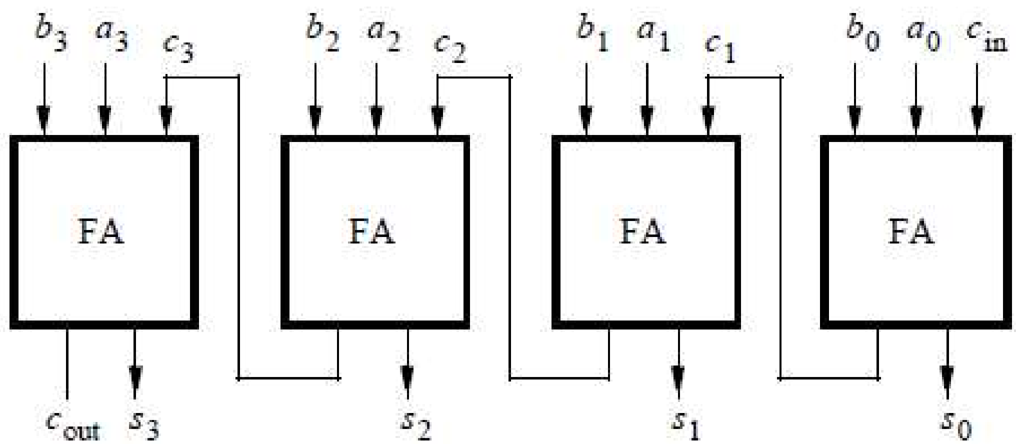
- Key points:
  - Combination circuits w/ an `always` block.
    - Use * in the sensitivity list of your `always` block.
  - Don't forget the `default` case! Or else the tool will need to have memory (which we haven't done yet).
  - New storage term: `reg` (used similar to `wire`)
    - Use `wire` with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

reg - register

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

- Implement a simple ALU (Arithmetic Logic Unit) and display inputs/ouputs to LEDs and 7-segment display (in hex).
  - You should reuse work you did in Lab 2.
  - Uses mux to implement addition, subtraction, inversion, etc.
    - More Verilog operators! ☺

Useful Verilog Operations

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

bit by bit

- For example:
  - `3'b101 & 3'b011` → `3'b001`

    3 digit with base b(inary)

- More general mathematical notation:
  - $(X_{n-1}X_{n-2}..X_1X_0 \ \& \ Y_{n-1}Y_{n-2}..Y_1Y_0)$ results in $W_{n-1}W_{n-2}..W_1W_0$ 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:
  - $(\& \ X_{n-1}X_{n-2}..X_1X_0)$ results in W_0 where W_o is $(X_{n-1} \ \& \ X_{n-2} \ \& \ ... \ \& \ X_1 \ \& \ X_0)$.
    - 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 `00000011`.
  - 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
```

so period of 40ns

- What's happening here:
  - The waveform for `a` (see first `force` command) starts low (logic-`0`) @ `0`ns
  - At `20`ns, it goes high (changes to logic-`1`).
  - It stays high until `40`ns when this process repeats
    - (i.e., it goes back to 0 until `60`ns; at `60`ns it goes back to `1` until `80`ns and so on..)
    - `a` is represented by a square wave with a period of `40`ns
  - 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)!

Specifying a vector:

- What you might want to do:
  - `wire multibit_vector [3:0];`
- The you should do:
  - `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]`