6.205
(aka 6.111)

Combinational Logic

Fall 2022

Administrative

- Pset 01 was just due
- Pset 02 is out at end of class (possibly a few minutes after/I need to check something). Short, due before due Lecture 03 on Thur
- Lab 01 is due Thursday at 10pm
- Lab 02 is out Thursday at 4pm

Variables in Verilog

- We'll use the logic type for our basic variable in 6.205
- It can represent a few different things depending on usage:
  - A "wire"…literally the routed output of some logic
  - A "reg"…a device that can hold a value over time (a form of memory)
- We'll worry about these two "forms" on Thursday

```
logic a; //simple variable (one bit in size)...can only hold 0 or 1
logic a,b,c; //declaring three single bit variables at the same time
```

Arrays in Verilog

- Want variables that can contain more than one bit of information
- Specify the sizing left-to-right like shown
- Can make any size you want, 2, 11, 17 bits
- Don’t feel compelled to use extra bit just because you’ve heard of variables being 32 bits or 16 bits before. Not bound by that structure.

```
logic [7:0] a; //8bit value (also think of this as an array of 8 bits)
logic [31:0] b; //32 bit value
logic [12:0] c,d; //making two arrays, each 13 bits that called c and d
```

Arrays in Verilog

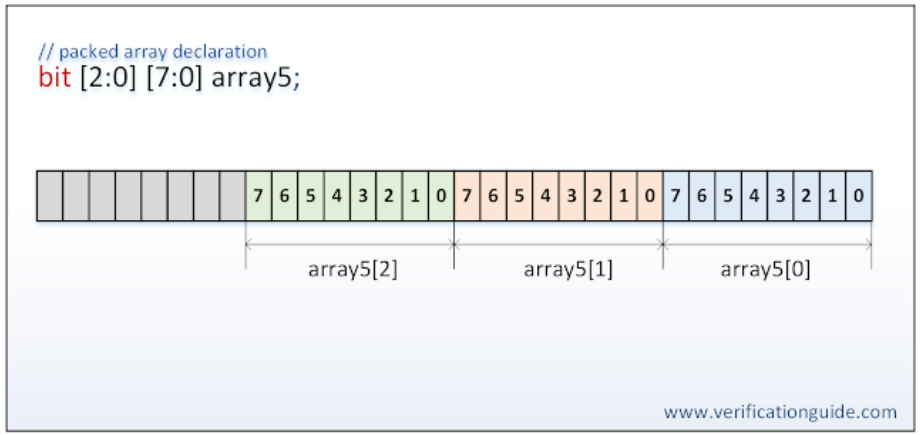
- Can also make "2D" arrays (packed/unpacked):
- The bottom two arrays are similar, but also different:
  - One is "packed"
  - One is "unpacked"
- Packed dimensions are specified before the variable name
- Unpacked dimensions are specified after the variable name

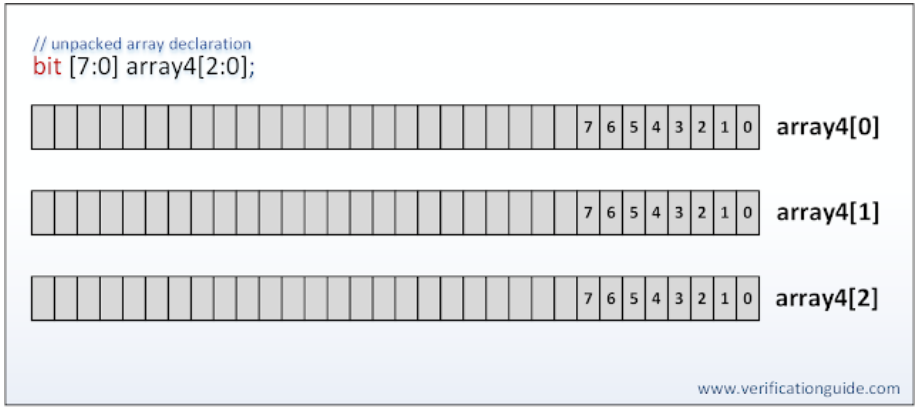
```
logic [7:0] array3;          //8 bit "packed array"
logic [7:0] array4 [2:0]; //three 8 bit unpacked arrays (b[0] not contiguous with b[1])
logic [2:0][7:0] array5 ; //three 8 bit packed arrays
                           //(array5[0] contiguous with array[1])
```

Un/Packed Arrays

- Packed means:
  - Whole structure is continuous
  - Like a subdivided larger array
- Unpacked means:
  - Separate/not continuous

```
logic [7:0] array3;          //8 bit "packed array"
logic [7:0] array4 [2:0]; //three 8 bit unpacked arrays (b[0] not contiguous with b[1])
logic [2:0][7:0] array5 ; //three 8 bit packed arrays
                           //(array5[0] contiguous with array[1])
```





Get familiar with the Three Bases

- Get somewhat fluent with the three bases.
- It will make life easier!

Denary	Binary	Hex
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Values in Verilog

- Good practice to always specify values in the following form: **S'Txxxx_xxxx** where
  - **S** is the size of the number (in bits)
  - **'** is the single quote marker
  - **T** is the numerical base you're specifying the value in
    - b for binary (0,1)
    - d for decimal (0,1,2,3,4,5,6,7,8,9)
    - h for hex (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F)
  - **xxxx_xxxx** are your values
    - The _ is ignored in evaluation
    - use _ to make more readable
    - Don't need to use _ but is really nice

Values in Verilog

- Some examples:

```
10'b0101_0101_00; //10 bit size of value...
10'b1; //10 bit value but only lsb specified...so this is saying 10'b0000_0000_01;
12'hF0F; //12 bits..this would be 12'b1111_0000_1111;
9'hF0F; //9 bits so 9'b1_0000_1111; top three cut off since we said only 9 long
15; //assumed to be an 32 bit integer by default:
    //              'b0000_0000_0000_0000_0000_0000_0000_1111;
```

Assignments

- Consider these:

```
logic a, b, c, d, e;
assign a = 1'b1; //best practice shows you mean to make this 1 bit
assign b = 0;
assign c = 1;
assign d = 15;
assign e = a && b;
```

- What values will all five variables have?

Assignments II

- What about arrays?

```
logic [7:0] a, b, c;
assign a = 8'b1010_1010; //good!
assign b = 16'hF0F0; //fine, but the top eight bits won't get stored
assign c = 32; //fine, but has: 8'b0010_0000 in it (surprise?)
```

- Watch out for size!
- Arrays have a size…you try to fit something too large in…it will get cut off (lsb's will get preference)

Assignments III

- What if we'd like to merge arrays?:

```
logic [7:0] a, b, c;
assign a = 8'b1010_1010; //good!
assign b = 16'hF0F0; //fine, but the top eight bits won't get stored
assign c = 32; //fine, but has: 8'b0010_0000 in it (surprise?)
logic [15:0] d;
logic [7:0] e, f;
assign d = {a,b}; //16'b1010_1010_1111_0000
assign e = {a[3:0], b[3:0]}; //has 8'b1010_0000;
assign f = {a,b}; //will have: 8'b1111_0000;
```

- Index into them however you want

Assignments IIIb

- What about this?

```
logic [2:0] e;
assign e = {1,1,1};
```

Other Ways to Assign (Implicit)

- Can also assign values upon declaration of variables in Verilog (implicit declaration as opposed to explicit with the assigns):

```
logic a = 1'b1; //same as assign a= 1'b1;
logic b = 1'b0;
logic [3:0] c = 4'b1010;
```

But be careful!!!

```
logic [3:0] d = 4'b1100;
assign d = 4'hF;
//might error out...might "choose for you"
```

- Be careful! Can't assign twice! This is not software! Higher up on page does not mean "earlier"

Other Ways to Assign (always_comb)

- You can also assign values/set relationships inside of a block known as always_comb
- Don’t need to use assign here:

```
logic a, b, c;
assign a= 1'b1;
assign b = 1'b0;
assign c = a^b;
//alternatively could do:
always_comb begin
  a = 1'b1;
  b = 1'b0;
  c = a^b;
end
```

Why Use an Always_comb?

- Can let you be more expressive, particularly when more complicated relationships need to be expressed!
- For example, can now do if/else logic cleanly

```
logic [3:0] a, b, c; //three four bit values!
always_comb begin
  if (a==4'b1010)begin
    c = 4'b1; //(0001)
  end else if (b==4'b0000)begin
    c = 4'b1010;
  end else begin
    c = 4'b0000;
  end
end
```

Why Use an always_comb?

- Always-family blocks also are analyzed in order if you use (=) assignments…Example:

```
logic [3:0] a, b, c; //three four bit values!
always_comb begin
  a = 4'b1010;
  a = a+b;
  a = a+c;
end
```

- Is the same as:

```
assign a = 4'b1010 + b + c;
```

Case Statement

- Need to do in an always block:

```
logic [8:0] a;
logic [1:0] b;
//make b 0, if a is 'b1111_0000
//make b 1, if a is 'b1010_0001
//make b 2, if a is 'b0000_1000
//else b is 3
always_comb begin
  case(a)
    8'hF0 : b = 2'b0;
    8'hA1 : b = 2'b1;
    8'h08 : b = 2'b10;
    default : b = 2'd3;
  endcase
end
```

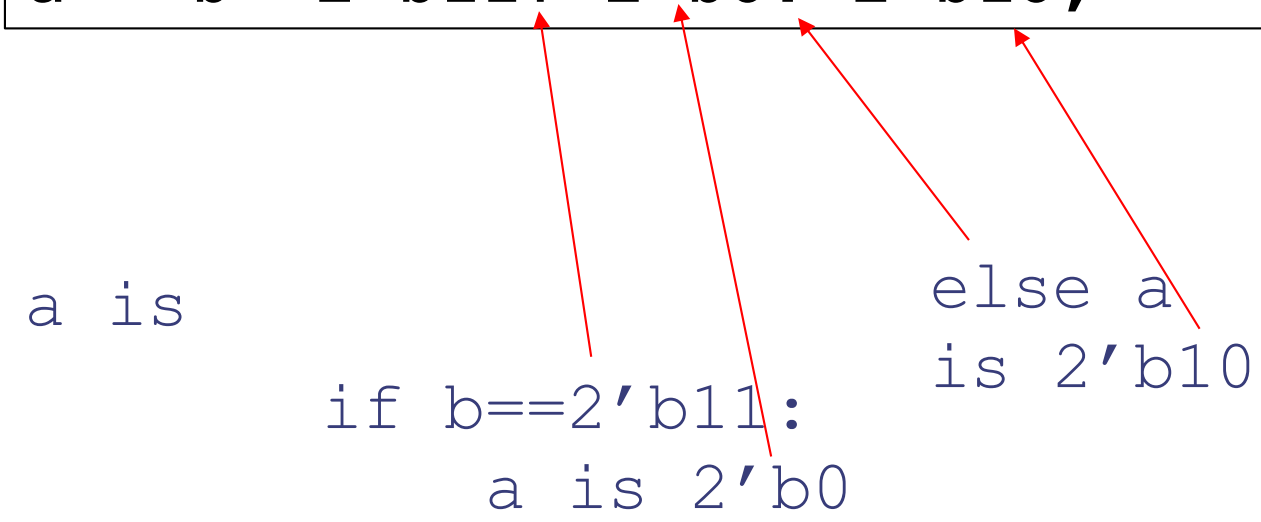
- Use these in place of long-chained if/else statements that are checking same variable
- Always have a default case! (safe, good practice)
- There is no fall-through in Verilog (no need for break statements like in C/C++)

Ternaries

- See these a lot in Verilog
- One-line if/else/if chains done on right side of assignment:

```
logic [1:0] a, b;
```

```
a = b==2'b11? 2'b0: 2'b10;
```



Ternaries

- Can also be done outside always_comb in regular assignment statements:

```
logic [1:0] a, b;
assign a = b==2'b11? 2'b0: 2'b10;
//if b is 2'b11, a is 0, else it is 2'b10
```

Ternaries

- Can also chain ternaries

```
always_comb begin
  if (a==4'b1010)begin
    c = 4'b1; //(0001)
  end else if (b==4'b0000)begin
    c = 4'b1010;
  end else begin
    c = 4'b0000;
  end
end
```

- Is the same as:

```
logic [3:0] a, b, c; //three four bit values!
assign c = a==4'b1010 ? 4'b1 : b==4'b0000 ? 4'b1010 : 4'b0000;
```

- Or since we're in a C-style language:

```
logic [3:0] a, b, c; //three four bit values!
assign c = a==4'b1010 ? 4'b1
         : b==4'b0000 ? 4'b1010
         : 4'b0000;
```

Where to Create Variables

- Variables are things that exist physically
- Always blocks are meant to describe action.
- You cannot declare variables in an always block
- As much as possible try to declare at top (with nice comments)
- And implement logic (assign, always_comb, etc) below it

Parameters

- Parameters allow us more flexibility in programmatically describing our designs:

```
localparam GOOD = 8'b1111_1111; //not changeable
localparam STATE_SIZE = 8;
parameter BAD = 8'b1111_0000; //changeable (see in a little bit how/where)
logic [STATE_SIZE-1:0] state; //made size of state variable based on param
logic [1:0] output;
always_comb begin
  case(state)
    GOOD : output = 2'b11;
    BAD : output = 2'b00;
    default : output = 2'b10;
endcase
end
```

Apply more meaningful names to values in certain contexts of program

Allow us to describe variable attributes using common adjustable values

Parameters

- `localparam` is local to the module it exists in
- `parameter` is local, but (depending on context), can be a configuration setting (see in a minute)
- Always CAPITALIZE so they are easy to spot
- Parameters can be based on other parameters!

```
parameter NUM_CHICKENS = 167;
parameter CHICKEN_WIDTH = $clog2(NUM_CHICKENS);
logic [CHICKEN_WIDTH-1: 0] chicken_counter;
```

- $clog2 is a Verilog math operator run at compile time
- Other Verilog math functions here:
  https://www.chipverify.com/verilog/verilog-math-functions

Modules

- Just like the idea of functions in software! Wrap up functionality in a reusable and "instantiable" blob

```
module not_gate (input wire x, output logic y);
  assign y = !x;
endmodule
module main_module();
  logic a,b;
  assign a = 1'b1;
  not_gate ng1 (a,b); //ng1 is name of instance
endmodule
```

Specify input/out variables and attributes (like size)

Do your operations

Make an instance of your module (name it) and use it

Declare instance like: `module_name instance_name (arg0,arg1,…);`

Modules Good Practice

- I try to append "_in" and "_out" on the module inputs and outputs to make more readable.
- Upon declaration you can also specify the ports explicitly. For modules with lots of inputs/outputs this makes tracing/debugging much easier!

```
module thing(input wire [3:0] a_in,b_in,
             output logic [3:0] c_ou);
  always_comb begin
    if (a==4'b1010)begin
      c = 4'b1; //(0001)
    end else if (b==4'b0000)begin
      c = 4'b1010;
    end else begin
      c = 4'b0000;
    end
  end
endmodule
//somewhere else you have main module of code:
module main();
  logic [3:0] q,r,s; //three four bit values!
  thing thing_0(.a_in(q),.b_in(r),.c_in(s));
endmodule
```

Parameterized Modules

- We mentioned parameters previously. They can be used to make flexible modules:

```
module add_constant #(parameter TO_ADD = 12)
  (input wire [7:0] val_in, output logic [7:0] val_out);
  assign val_out = val_in + TO_ADD;
endmodule
module top();
  logic[7:0]a,b,c,d;
  assign a = 8'd11;
  assign c = 8'b100;
  add_constant ac_0 (.val_in(a), .val_out(b));
  add_constant #(.TO_ADD(5)) ac_1 (.val_in(c), .val_out(d));
  //value of b?
  //value of d?
endmodule
```

Parameterized Modules

- Parameterizable modules are more complicated to write, but their reusability is a great feature
- If a parameter is not specialized upon instantiation, the default is used instead.
- Parameters can be used to influence other parameters in the design

for loops

- For loops (and to a lesser extent while loops) exist in Verilog to more conveniently lay out our hardware.
- There are two general types:
  - Generate for loops (for loops in a generate block)
  - Regular for loops
- Which one works can be confusing* so we'll over it here

*the rules have also changed as Verilog evolves so there can be confusing info on the internet

Regular for loop

- If you are in an always block and just need to replace a bunch of repetitive lines, a for loop can help
- Let's say I had to do some annoying operation a bunch of times with some variables:

```
always_comb begin
  for(integer i =0; i<64; i= i+1)begin
    a[i] = b[i]>c[63-i];
  end
end
```

Generate For loops

- Put a for loop in a generate block.
- Use this any time you need to :
  - Run multiple assign statements
  - Create multiple always_comb, always_ff blocks
- OR:
  - Create multiple instances of a module
  - Create logics
- Need to use a genvar for your iterating variable rather than an integer.
- Can also label your for loops to have access the modules or entities created within

Generate For Loops

An Example:

```
generate
  genvar i;
  for(i=0; i<5; i=i+1)begin: myloop
    logic[31:0] hi;
    assign hi = 32'hAAAAAAAA ^ i;
  end
endgenerate
//outside of generate, those logics can be accessed with:
// myloop[2].hi for example
// this is needed since the logic hi needs more
// specifity than provided otherwise.
```

Wire vs. logic. vs. reg

- `wire` Can only be signal flow ("nets"). From perspective of a module, signals coming into module are conveyed by wires. In other usage, declared wires can only be given values with **assign** statement. A wire can also be associated with combinational logic
- `reg` Ideally represents a flipflop or latch (storage mechanism), but in reality can also turn into a net (in other words a wire)/ combinational logic based on usage (cover more on Thursday in Lec 03). Only given values with **always-family** blocks
- `logic` Can represent all datatypes. Its usage dictates what it ultimately represents (combinational logic). Can be worked with **assign** and **always-family** blocks

Why Logic?

- In addition to allowing us to just use one general type rather than two, the **logic** datatype has stricter protections against multi-driven nets

```
module thing(input wire [3:0] a_in,b_in,
output wire [3:0] c_out);
  //stuff
endmodule
module main_module();
  logic[3:0] a,b,c;
  thing my_thing(.a_in(a), .b_in(b), .c_in(c));
  assign c = 4'b1010; //whoops might make it through (multi-driven net)
endmodule
```

- Logic on output should prevent:

```
module thing(input wire [3:0] a_in,b_in,
output logic [3:0] c_out);
  //stuff
endmodule
module main_module();
  logic[3:0] a,b,c;
  thing my_thing(.a_in(a), .b_in(b), .c_in(c));
  assign c = 4'b1010; //should get caught on synthesis
endmodule
```

6.205 Caveat

- We would like to always wrap our code files in:

```
`default_nettype none
//stuff
//stuff
//stuff

`default_nettype wire
```

- Prevents Vivado from inferring undeclared variables for us:

```
module main_module();
  logic [7:0]a,b;
  assign c = a+b; //vivado infers c, but makes a one bit variable
  //this might/will be a problem!!!
endmodule
```

- Forcing the default nettype to be none (rather than wire) will force an error at synthesis (early) and this is good!
- But need to put back to nettype wire at end since Vivado IP uses that.

Problem Statement

- I want to be able to add up a variable number of 8-bit unsigned integers in hardware.
- Let's say up to 128 values.
- I want to add the entire array up as fast as possible
- How would we do it?

Software Solution?

- Take first value, add to second.
- Then take third, add to that sum
- Then fourth, add to that sum,
- Etc…
- These adds would take some time since they need to be done one after the other.
- A hardware solution can be faster since we can have separate parts doing adds at the same time

Let's build two different adders

- Adder 1 (to get some generate practice):
  - Explicitly lay out a tree-shaped adder module for 8-wide
- Adder 2: A parameterized adder module that works for an arbitrary bit width and number of inputs