6.205
(aka 6.111)

Combinational Logic

Fall 2025

Administrative

- Week 1’s content is due tomorrow (Wednesday) night at 11:59pm
- Week 2’s content will come out Thursday after lecture @4pm

Review

1. Have an Idea: “I have three wires of one bit, x,y, and z.
   - I want to treat them collectively as a number…where z is the one’s place, y is the two’s place, and x is the four’s place.
   - If that number is 3, 5, 6, or 7, I want the output bit to be high, else I want it to be low.

**”High” and “Low” refer to the two states in the digital domain*

Review

2. Implement the idea in SystemVerilog:

```
module some_module
  ( input wire x,
    input wire y,
    input wire z,
    output logic out);

    logic [2:0] temp;
    assign temp = {x,y,z};
    always_comb begin
        if (temp>=5)begin
            out = 1'b1; //specify bit
        end else if (temp==3)begin
            out = 1'b1; //specify bit
        end else begin
            out = 1’b0; //specify bit
      end
    end
endmodule
```

This Then gets Turned into a Circuit

- Most synthesis tools will provide some sort of intermediate visualization if you want.
- Yosys (an open toolchain) can do this somewhat easily.
- This site here: https://digitaljs.tilk.eu/ is built on Yosys and let's you see what pops out
- For example module from previous page yields…

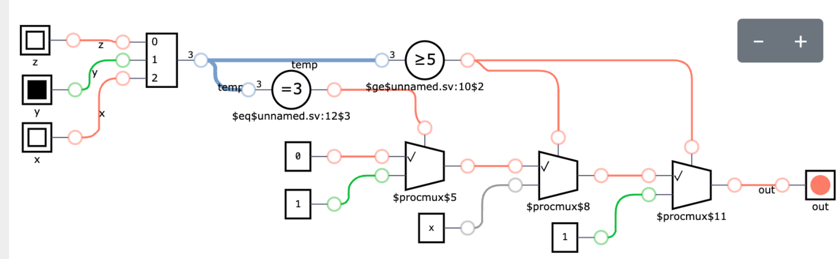
Equivalent Schematic

Run | 25080

Setup | I/O | unnamed.sv

```
module some_module
  ( input wire x,
    input wire y,
    input wire z,
    output logic out);

  logic [2:0] temp;
  assign temp = {x,y,z};
  always_comb begin
    if (temp>=5)begin
      out = 1'b1; //specify bit
    end else if (temp==3)begin
      out = 1'b1; //specify bit
    end else begin
      out = 1'b0; //specify bit
    end
  end
endmodule
```



This can be helpful, but isn't the full story

- In reality when this gets built on an FPGA, additional steps will be taken to reduce it into more primitive functional expression, which we can helpfully visualize with a truth table or sum of products.
- For this example, it would look like this:

x	*y*	*z*	*output_1*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

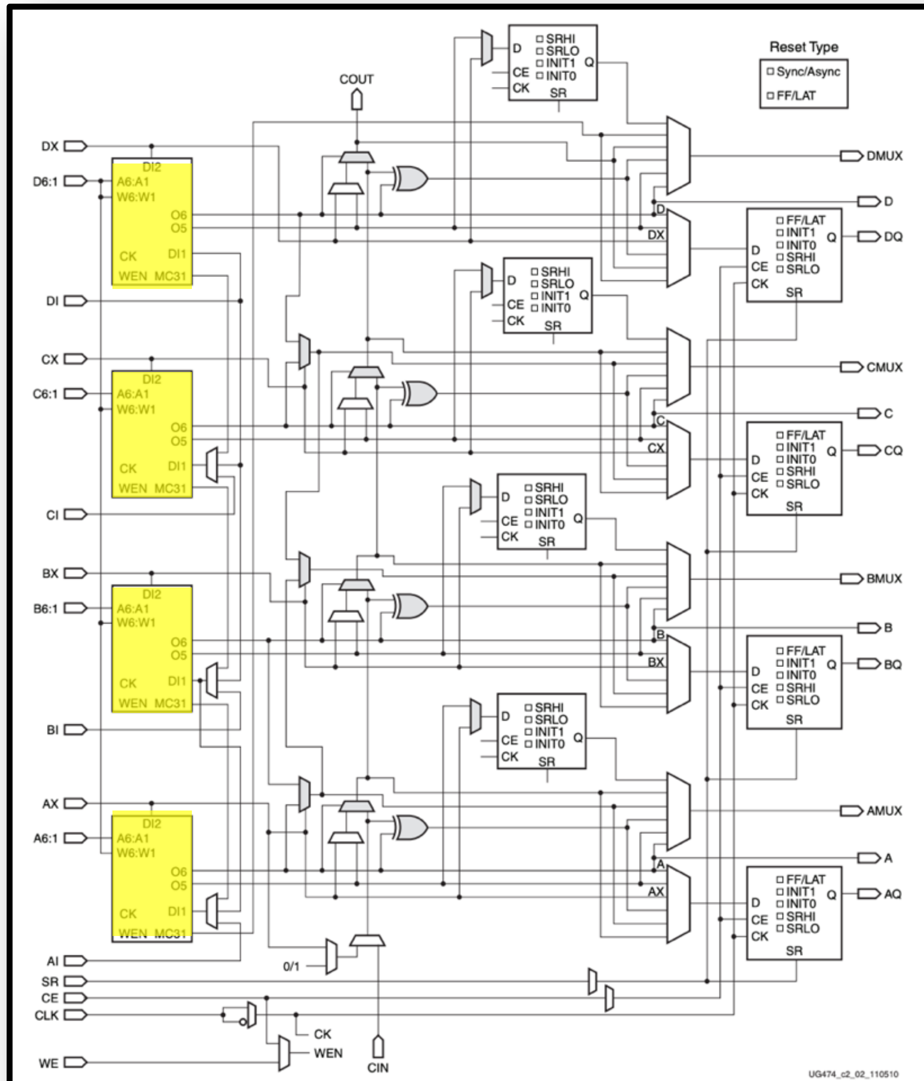
Review

x	y	z	$output\_1$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

- These truth tables are important because they are effectively the ”code” that are used to program the fundamental units of the FPGA, the CLBs and associated wiring

Xilinx Logic Blocks

- Our FPGAs have about 8500 of these →
- Called "Logic Slices"
- Each slice has <u>four</u> CLBs ("**C**onfigurable **L**ogic **B**locks") that form the clay from which we sculpt our digital functions



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)
- Right now we're not super worried about "reg"s

```
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
```

There are other Data Types

- SystemVerilog the language has other datatypes
- There are int’s, shorts, etc…all with signed/unsigned versions…we’ll leave them be for a little bit!
- However when we do things with loops we’ll use int’s to help us iterate!

- For now just use **logic** variables.
  - Can be any size
  - By default unsigned (we’ll worry about signed-ness in future weeks)

Multibits 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

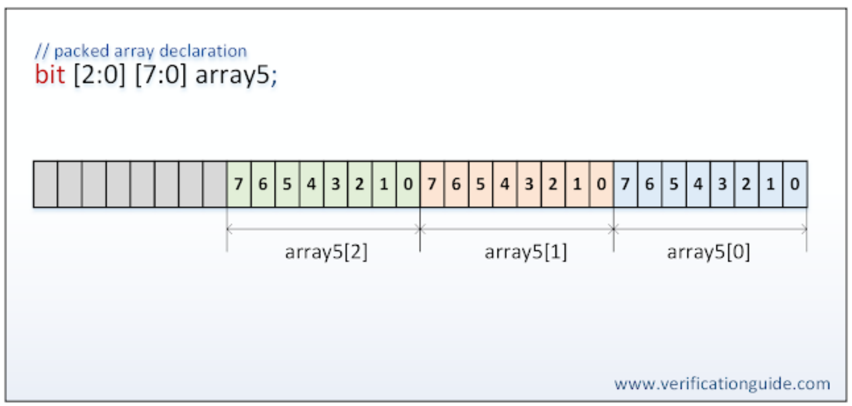
- Can also make “2D” arrays (packed/unpacked):
- The bottom two arrays are similar, but also different:
  - One is “packed”
  - One is “unpacked”
- Packed dimensions are specified before the variable name
- Unpacked dimensions are specified after the variable name

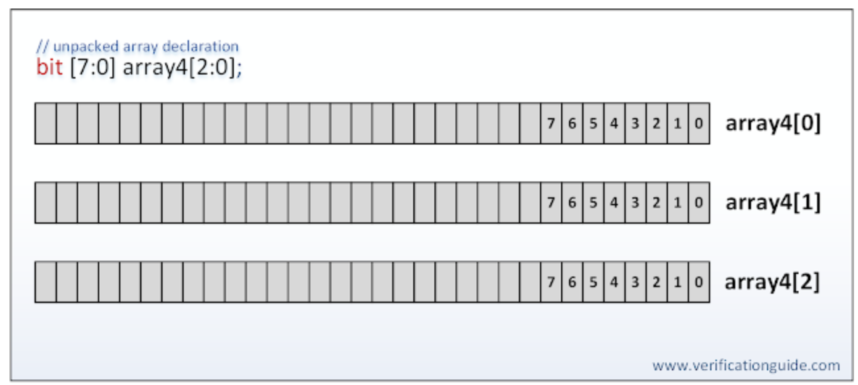
```
logic [7:0] array3;        //8 bit "packed array"
logic [7:0] array4 [2:0]; //three 8 bit unpacked arrays (b[0] not contiguous with b[1])
logic [2:0][7:0] array5 ; //three 8 bit packed arrays
                           //(array5[0] contiguous with array5[1])
```

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 array5[1])
```



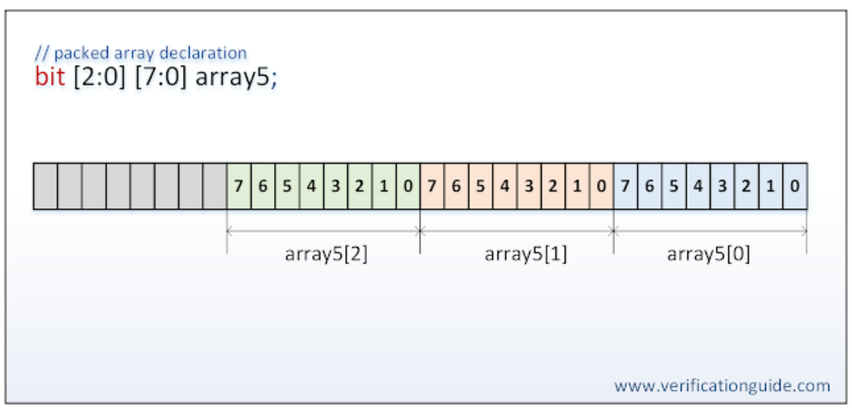


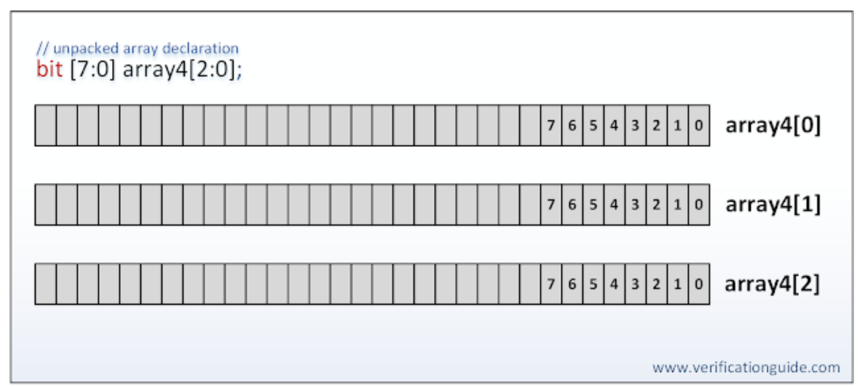
Un/Packed Arrays

- Packed/Unpacked has little meaning beyond the program construct within the Verilog language

```
logic [7:0] array3;       //8 bit "packed array"
logic [7:0] array4 [2:0]; //three 8 bit chunks (unpacked)
logic [2:0][7:0] array5 ; //really just one 24-bit chunk with sub-indexing convenience
```

- Unpacked array: Use to handle the output of three separate adders, for example
- Packed array: Use to represent a string type object, for example (maybe?).





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;
```

I’d generally recommend against doing this! Because...

```
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 necessarily 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` in `always_comb`:

```
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;
```

Inside an always-type block

- Order of Code *can* matter

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

- The entire block is analyzed and turned into a “hidden” one-liner like this*:

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

*or something…differences aren’t important now.

Case Statement

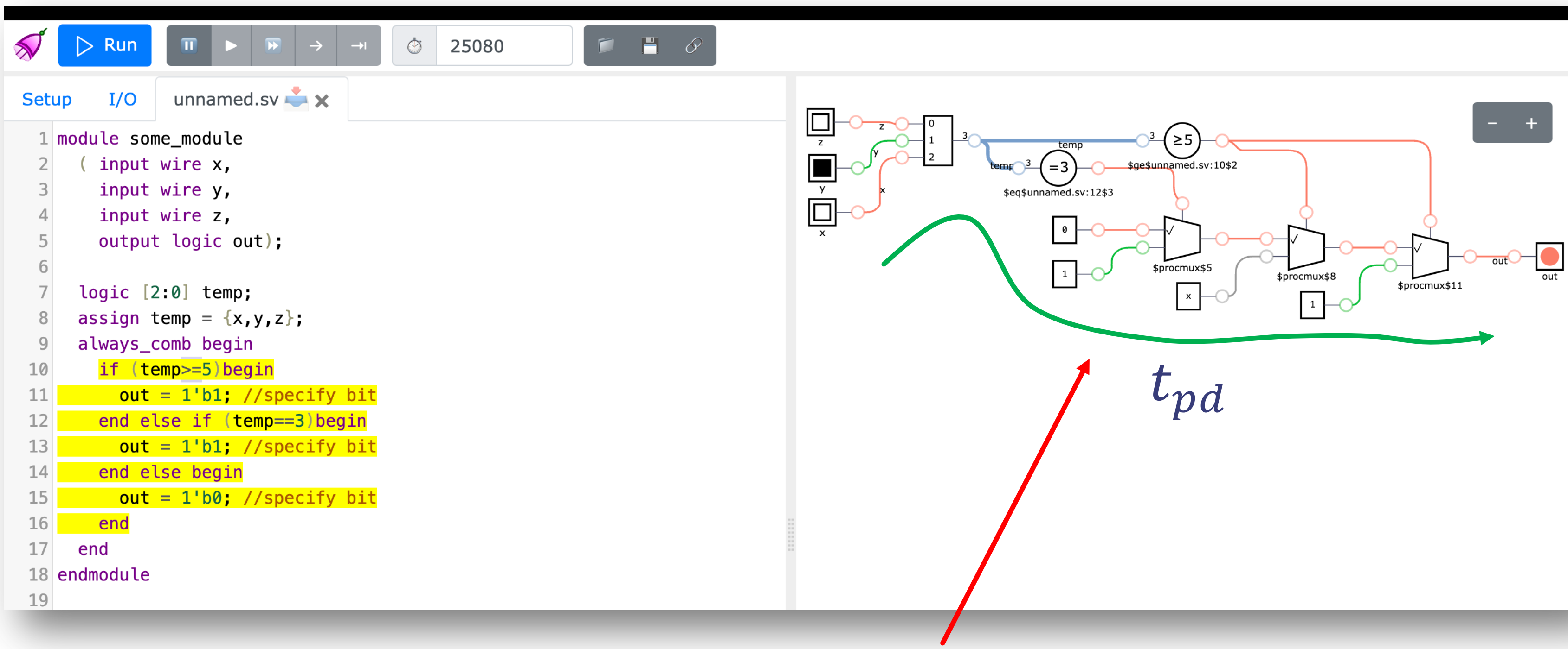
- Need to do in an always block:

```
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++)

Why Try to Avoid Long-Chained If-Else?

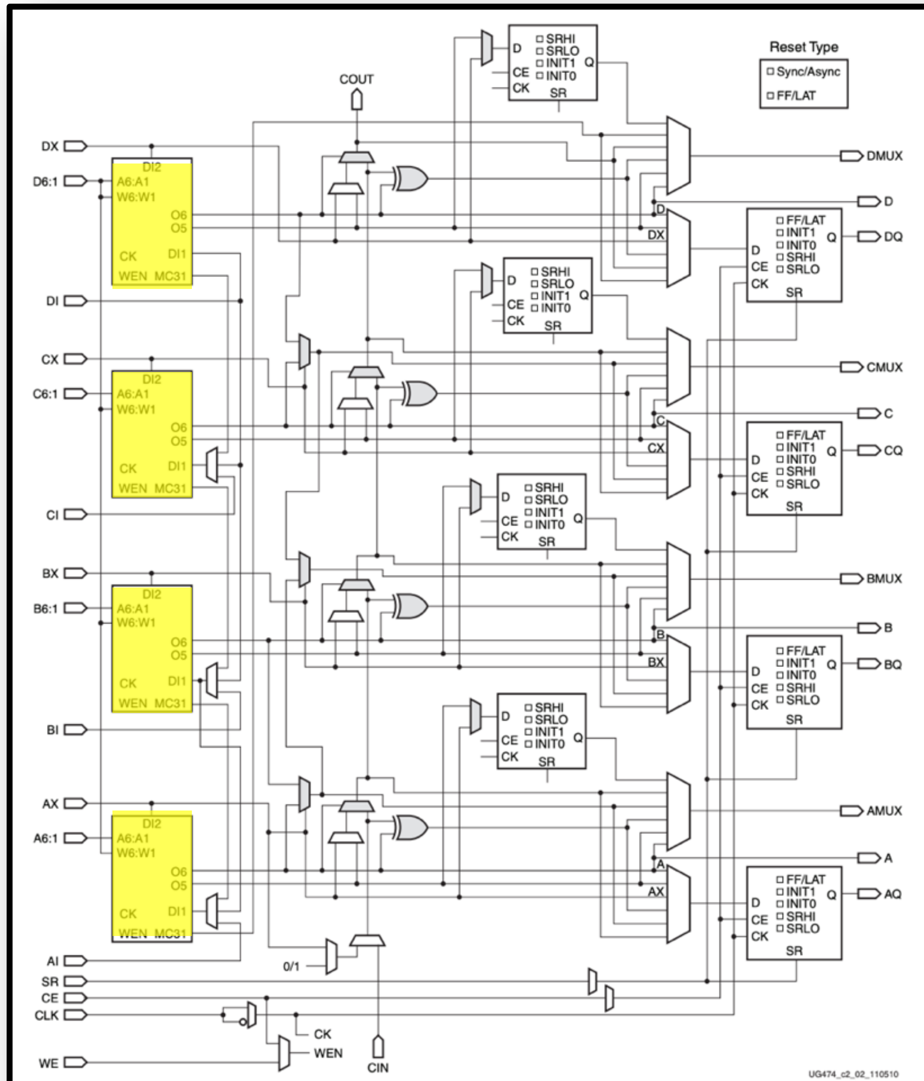
- Because if, else if, else if, else… has a priority encoded in it….a hierarchy of when one decision should be made relative to others.



Can end up having a loooong propagation delay

Case Statements

- Can evaluate to multi-bit multiplexers which are relatively “shallow” pieces of logic with lower propagation delays
- And readily available in FPGA:
- More on Thursday



Always blocks?

- For stuff you write, stick with specific always family blocks:
  - `always_comb`
  - `always_ff` (coming up)
  - `always_latch` (coming up)

What about `always @(*)`

- Historically, there was just one `always` block and you would infer different types of logic (combinational, latch, or sequential) from what was in the parentheses:

```
always @(<sensitivity list>)begin

    //do your stuff here when a change happens
    //to anything specified in sensitivy list

end
```

Simple combinational adder

- For example you would do:

```
always @(x,y)begin
  z= x+y;
end
```

"any time x or y changes, z changes as x+y." This is a purely combinational adder

- Verilog 2001 brought in the "wildcard". Same as above can be done with:

```
always @(*)begin
  z= x+y;
end
```

"any time anything in the block changes, z changes as x+y." This is a purely combinational adder

Consider This Situation

- "I want a combinational circuit that says z = x+y if x is 3."
- Here's my solution:

```
always @(*)begin
  if (x==3)begin
    z = x+y;
  end
end
```

- Problems with this?

Remember what we’re doing

- We are specifying (using HDL) a Boolean function. That function has a finite input space.
- We need to make sure we are specifying how this circuit should work for the *entire* input space:

```
always @(*)begin
  if (x==3)begin
    z = x+y;
  end
end
```

- Code above is saying set z to be x+y when x==3. It says nothing else.
- There is a device that will enable this as stated but it is not combinational!

The Resulting Truth Table

- Let's just assume x is two bits...

$\boldsymbol{x}[\mathbf{1}]$	$\boldsymbol{x}[\mathbf{0}]$	$f(x,y,z)$
0	0	?
0	1	?
1	0	?
1	1	$\boldsymbol{x} + \mathbf{y}$

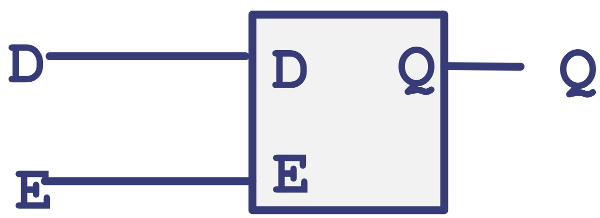
- Not to decide is to decide...
- What you just want it stay the same or something? That's opening up a whole world of issues... Staying the same means you need a concept of time in your truth table...

A part that remembers (starting in Lec 03)

- Talked a little about stateful things in lecture 1... in addition to combinational blocks there are several stateful things too!
- Two big ones!

<u>D Latch</u>

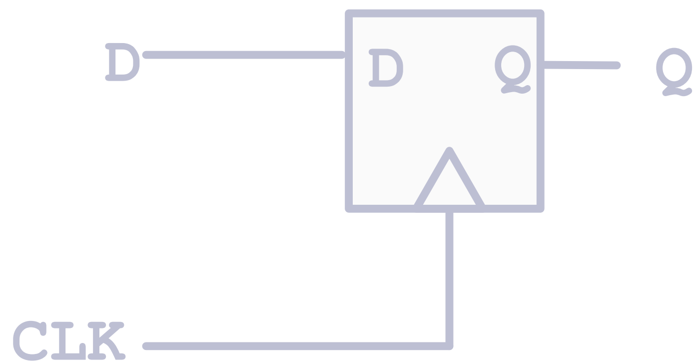
Level-Triggered Sample-and-Hold Device



"store D when E is high"

<u>D Flip-Flop</u>

Edge-Triggered Sample-and-Hold Device



"store D when clk rises"

Missing Input Space

```
always @(*)begin
  if (x==3)begin
    z = x+y;
  end
end
```

- This code fails to specify what to do when x!=3.
- It therefore assumes you want to do nothing.
- A latch will do that:
  - When x==3, set z to be x+y
  - When x!=3, hold the value you already have
- Correct code would be:

```
always @(*)begin
  if (x==3)begin
    z = x+y;
  end else begin
    z = 0;
  end
end
```

A latch is not combinational

- Suddenly your design will have “memory” in it where you never intended.
- This can mess up your simulations and designs!
- Vivado will happily synthesize a latch for you since it’ll think that’s what you want.
- Forcing it to know you want combinational logic (via always_comb) can throw warnings:

```
WARNING: [Synth 8–327] inferring latch for variable
‘z_reg’ [/top_level.sv:12]
```

- It will also ensure there’s less chance of simulation-to-reality variations

To be annoying!!!!

- We are specifying (using HDL) a Boolean function. That function has a finite input space.
- We need to make sure we are specifying how this circuit should work for the ***entire*** input space:
  - Ideally do this explicitly
  - If you do implicitly make sure you're doing so responsibly!
- If you fail to specify your truth table in full, unknown behaviors will exist and wreak havoc

x	*y*	*z*	*output_1*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	?
1	1	0	?
1	1	1	?

Never use `always`
Always use `always_comb`

- When writing your logic:
  - Make sure to cover the entire input space for each variable in its entirety!
  - Do not forget about leftovers:
    - Have a terminal else in case of if/else if chain
    - Have a default case in the case of a case statement
    - Initialize starting values for variables at start of `always_comb` block!
  - Scan output logs from vivado for word “latch”. If there’s any getting inferred, make sure it is because you want them (very unlikely in our class)

In Conclusion

- Tons of legacy code will have them so you should be aware and know how to how to read it and deal with it!

No way ~~drugs~~ **`always@()`** blocks are for loserzz! I've got too much to lose to get mixed up with them.

- For stuff you write, stick with specific always family blocks:
  - `always_comb`
  - `always_ff` (coming up)
  - `always_latch` (coming up)

Other Syntax...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;
```

a is

if b==2’b11:

a is 2’b0

else a

is 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;
```

Ternary style Specification

- One benefit of it is that by its syntactic nature it forces you to have a trailing else:

```
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
```

- You cannot have something like this:

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

- Nice because it forces you to cover your full input space of possibility, avoiding gaps/resulting latch

Competing Assignments

- What if I have two always_comb blocks?

```
always_comb begin
  a = c + e;
end

always_comb begin
  a = d + 5;
end
```

- Only one will be chosen, the other ignored. It will not make a union or merge the two.

Multiple always-type blocks

- It is fine to use values across multiple always blocks or continuous assign statements, but you should only specify them in one and only one location!

```
always_comb begin
  d = a+ 5;
end

always_comb begin
  b = a + 8;
end
```

- Specifying/assigning a variable in multiple always-type blocks is a no-no however

Where to Create Variables

- Variables are things that exist physically
- Always blocks are meant to describe action.
- You *can never* 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 are different than variables.
- Their values can change, but only at the compile-stage.
- At run-time they are constants.
- They allow us to make flexible designs (make an adder that can be 8 bits or 14 bits or whatever)

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 few slides 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/output 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...but you really should use named port convention!

- 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 (.x(a), .y(b)); //ng1 is name of instance
endmodule
```

Specify input/output variables and attributes (like size)

Do your operations

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

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 specify other parameters in the design!

Operator Precedence

- Largely borrowed from C!
- Be careful some of these often feel out of order for people.
- Left/right shift for example!
- For example if:
  - x=100
  - q=8
  - What will y be?

```
assign y = x + q>>2;
```

- 27…not…102

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
! ~ & | ~& ~| ^ ~^ or ^~	logical negation negation reduction AND reduction OR reduction NAND reduction NOR reduction XOR reduction XNOR	logical bit-wise reduction reduction reduction reduction reduction reduction
+ -	unary (sign) plus unary (sign) minus	arithmetic arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
* / %	multiply divide modulus	arithmetic arithmetic arithmetic
+ -	binary plus binary minus	arithmetic arithmetic
<< >>	shift left shift right	shift shift
> >= < <=	greater than greater than or equal to less than less than or equal to	relational relational relational relational
== !=	logical equality logical inequality	equality equality
=== !==	case equality case inequality	equality equality
&	bit-wise AND	bit-wise
^ ^~ or ~^	bit-wise XOR bit-wise XNOR	bit-wise bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

Reduction Operators in Verilog

- Reduction operators act like their bitwise cousins, but are done on a variable rather than between several:

```
logic [7:0] b, d;
logic       a, c;

assign a = |b; //if anything in b is 1, a is 1
assign c = &d; //everything in d needs to b 1
//four others xor and xnor are particularly
useful
```

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal to	relational
<	less than	relational
<=	less than or equal to	relational
==	logical equality	equality
!=	logical inequality	equality
===	case equality	equality
!==	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
^~ or ~^	bit-wise XNOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

for loops

- For loops (and to a lesser extent while loops) exist in Verilog to more conveniently lay out our hardware.
- They are NOT for loops “in time”. They are for loops “in space”
- 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 evolved 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:

```
logic [7:0] b [63:0];
logic [7:0] c [63:0];
logic [63:0] a;

//assume b and c are large enough
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 :
  - create 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
// specificity than provided otherwise.
```

Rule about For Loops

- Inside an always_comb (or always_ff?):
  - Use *regular for loop*

- Want to make multiple assign statements? Or Multiple always-type blocks? or multiple modules?:
  - Use a *generate loop!*

- In both instances, the iterating variables of the loop have no intrinsic hardware meaning…they exist as a helper variable during specification (a copy-paster thingie)

`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. ***DO NOT USE IN 6.205***

- `logic` Can represent all datatypes. Its usage dictates what it ultimately represents (combinational logic or Flip Flops). 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,b,
output wire [3:0] c);
  //some behavior goes here
endmodule

module main_module();
  logic[3:0] a,b,c;
  thing my_thing(.a(a), .b(b), .c(c));
  assign c = 4'b1010; //whoops might miss checks in Vivado (multi-driven net)
endmodule
```

- Logic on output should prevent:

```
module thing(input wire [3:0] a,b,
output logic [3:0] c);
   //some behavior goes here
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
```

So why still use `wire` in module definitions at all?

```
module thing(input wire [3:0] a_in,
             input wire [3:0] b_in,
             output wire [3:0] c_out);
  //stuff
endmodule
```

- Seems excessive...Let's just use logic for everything.
- We would...but...
- This is a thing we do in 6.205 to help us with our Vivado toolchain since it is a picky, picky child.

If we had a module definition like this:

```
module thing(input logic [3:0] a,
             input logic [3:0] b,
             output logic [3:0] c);
  assign c = a ^ b;
endmodule
```

- This would run just fine in all honesty.
- In the scope of this module, what logic ”is” isn’t specified, but by default Verilog interpreters tend to assign the “wire” attribute to unspecified things (like these logic’s would be).

The Problem...

```
module thing(input logic [3:0] a,
             input logic [3:0] b,
             output logic [3:0] c);

  assign d = 4;
  assign c = a ^ b ^ d;
endmodule
```

- This verilog will synthesize perfectly happy for the same reason the verilog on the previous page did. Why???
- d id undeclared, but Verilog just assumes it is a “wire” by default...a one-bit wire. omg why.

To Protect Ourselves...

- we will (and should) often add the a directive at the top of our file to protect against this:

```
`default_nettype none
```

- This forces Verilog to treat everything not explicitly declared as a none entity and working with those ***__will__*** throw an error

The downside of that though...

- All inputs and outputs are of type `logic`
- A `logic` is an abstract type whose physical realization is determined through usage.

```
`default_nettype none
module thing(input logic [3:0] a,
             input logic [3:0] b,
             output logic [3:0] c);
  assign c = a ^ b;
endmodule
```

- In the scope of this module, how these inputs should get their values is never specified so their actual manifestation is left undefined and defaults to a none and an error gets thrown (good!)

So must make the inputs into wires

- All inputs are now wires

```
`default_nettype none
module thing(input wire [3:0] a,
             input wire [3:0] b,
             output logic [3:0] c);
  assign c = a ^ b;
endmodule
```

- In the scope of this module, inputs a and b are explicitly known to be wires (things that convey signals) so there’s no ambiguity
- Still have that great protection against accidental variables since they’ll be none!

But That’s Not Enough

- It just keeps going...
- A lot of Vivado’s source files and modules *rely* on the default nettype being `wire` (I know who would do that?).
- So after our module we need to set things back to defaulting to wire
- So we’ll tack this on at the end of files:

```
`default_nettype wire
```

To Satisfy All of These Issues

- Declare inputs to a module as wires since it will fully specify what they are
- Begin and end all modules with nettype compiler directives. This will protect you from implicit declarations by Vivado

```
`default_nettype none
module thing(input wire [3:0] a,
             input wire [3:0] b,
             output logic [3:0] c);
  assign c = a ^ b;
endmodule
`default_nettype wire
```

Kind of Annoying

- Yeah it is. :/ But this is the sort of thing you deal with when working with a large vendor’s toolchain.

- Or a language that just can’t let go of the past and keeps maintaining various degrees of backwards compatibility back to the 1980s

Let’s build some different adders

- Adder 1 (parameter practice):
  - Add up two variable width values (width is parameterized)
- Adder 2: A parameterized adder module that works for an arbitrary bit width and an arbitrary number of input values
- Adder 3/if time…(to get some generate practice):
  - Explicitly lay out a tree-shaped adder module for 8, 8-wide inputs.
  - Force the structure to be a tree shape: