ECE 551

Digital Design And Synthesis

Fall '21

Lecture 2

- Commenting & Meta-stability
- Verilog Syntax (literals, 4-value, strength vectors)
- Dataflow Verilog
- Handy Miscellaneous

1

Administrative Matters

- Tutorial sessions
  - Tuesday 6:30 – 8:00 **for those in Dis. 302 only** in B555 Linux lab.
  - Wednesday 6:30 – 8:00 **on General channel of MS Teams**
  - Wednesday 8:00 – 9:30 **on General channel of MS Teams**
  - Can also do on your own
- If you are a windows user install ModelSim locally
- OR...look at video on how to remote to Unix
- Watch Videos on mid & final sections of Lecture02
  - Quiz on these lectures next Monday in class.
- Homework #1
  - Due Next week by beginning of class

2

2

Comments in Verilog

- Commenting is important
  - In industry many other poor schmucks are going to read your code
  - Some poor schmuck (perhaps you 4 years later) are going to have to reference your code when a customer discovers a bug.
- The best comments document why you are doing what you are doing, not what you are doing.
  - Any moron who knows verilog can tell what the code is doing.
  - Comment why (motivation/thought process) you are doing that thing.

3

3

Commenting in Verilog

```
always @(posedge clk)

begin
  Sig_FF1 <= Sig          // Capture value of Sig Line in FF
  Sig_FF2 <= Sig_FF1;     // Flop Sig_FF1 to form Sig_FF2
  Sig_FF3 <= Sig_FF2;     // Flop Sig_FF2 to form Sig_FF3
end

// start_bit is ~Sig_FF2 & Sig_FF3
assign start_bit = (~Sig_FF2 && Sig_FF3) ? 1'b1 : 1'b0;
```

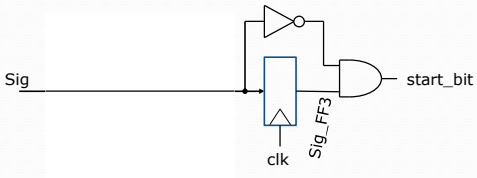
(Read with sarcasm)

"Thanks for the commenting the code pal. It tells me so much more than the verilog itself".

4

4

Consequence of Bad Commenting



The code synthesizes as shown above

Being the next arrogant engineer reading the code and the comments I realize there is an optimization to be made.

There is no need for the two flops up front. So, I delete them.

5

5

Commenting in Verilog

```
always @(posedge clk)
/***********************************************
* Sig is asynchronous and has to be double flopped  *
* for meta-stability reasons prior to use ************
***********************************/
begin
   Sig_FF1 <= Sig;
   Sig_FF2 <= Sig_FF1;  // double flopped meta-stability free
   Sig_FF3 <= Sig_FF2;  // flop again for use in edge detection
end

/***********************************************
* Start bit in protocol initiated by falling edge of Sig line *
***********************************************/
assign start_bit = (~Sig_FF2 && Sig_FF3) ? 1'b1 : 1'b0;
```

Can see 2 types of comments.

Comment to end of line is //

Multi line comment starts with /* and ends with */

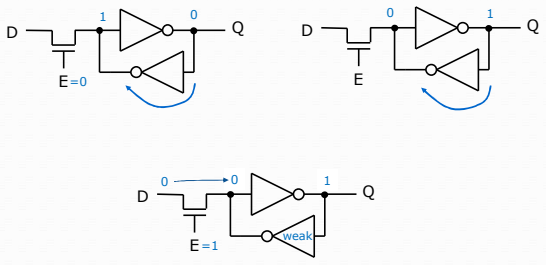
- This is better commenting. It tells you why stuff was done

6

6

Aside: what is meta-stability

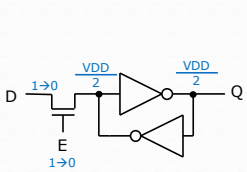
Back to back inverters
Form a "memory loop"
that is central to latch

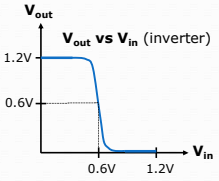


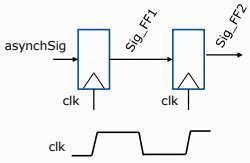
7

7

Aside: what is meta-stability







Yes…the first flop can go meta-stable, but can is stay in that state for a whole clock period?

Yes, and a meteor could crash through the roof and kill us all.

Take away: Always double flop asynch signals before "using" them

8

8

Numbers in Verilog

- General format is: <size>'<base><number>
- Examples:
  - 4'b1101 // this is a 4-bit binary number equal to 13
  - 10'h2e7 // this is a 10-bit wide number specified in hex
- Available bases:
  - d = decimal (please only use in test benches)
  - h = hex (use this frequently)
  - b = binary (use this frequently for smaller #'s)
  - o = octal (who thinks in octal?, please avoid)

9

9

Numbers in Verilog

- Numbers can have x or z characters as values
  - x = unknown, z = High Impedance
  - 12'h13x // 12-bit number with lower 4-bits unknown
- If size is not specified then it depends on simulator/machine.
  - Always size the number for the DUT verilog
    - Why create 32-bit register if you only need 5 bits?
    - May cause compilation errors on some compilers
- Supports negative numbers as well
  - -16'h3A // this would be -3A in hex (i.e. FFC6 in 2's complement)
  - I rarely if ever use this. I prefer to work 2's complement directly

10

10

Identifiers (Signal Names)

- Identifiers are the names you choose for your signals
- In a programming language you should choose descriptive variable names. In a HDL you should choose descriptive signal names.
  - Use mixed case and/or _ to delimit descriptive names.
    - ✓assign parityErr = ^serial_reg;
    - ✓nxtState = returnRegister;
  - Have a convention for signals that are active low
    - ✓Many errors occur on the interface between blocks written by 2 different people. One assumed a signal was active low, and the other assumed it was active high
    - ✓I use **_n** at the end of a signal to indicate active low
    - ✓rst_n = 1'b0 // assert reset

11

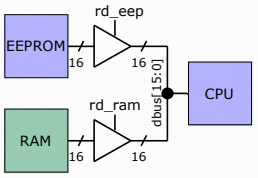
11

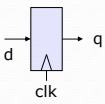
Signal Values & Strength in Verilog

- Signals can have 1 of 4 values

Value:	Meaning:
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown, either uninitialized or in contention
z	High impedance. Not driven, or not connected

Need high impedance for modeling of tri-state busses





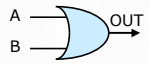
What value does q have prior to any positive clock edge?

It is unknown: x (uninitialized)

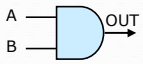
12

12

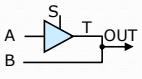
Resolving 4-Value Logic



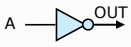
A	B	OUT
0	0	0
0	1	1
1	1	1
0	x	x
0	z	x
1	x	1
1	z	1



A	B	OUT
0	0	0
0	1	0
1	1	1
0	x	0
0	z	0
1	x	x
1	z	x



S	A	T	B	OUT
0	0	z	z	z
0	1	z	x	x
0	x	z	1	1
0	z	z	0	0
1	0	0	1	x
1	0	0	z	0
1	1	1	z	1
1	x	x	z	x
1	z	x	0	x



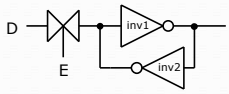
A	0	1	x	z
OUT	1	0	x	x

13

Signal Values & Strength in Verilog

Strength Level	Meaning:
supply	Strongest (like a supply VDD or VSS)
strong	Strong (default if not specified)
pull	Medium strength (like a pullup device)
weak	Weak, like a keeper (sustainer, leaker)
highz	High impedance

- Concept of drive strength supported to model certain CMOS circuits that utilize contention.




```
not                    inv1 (Q,n1)
not (weak0,weak1) inv2 (n1,Q)
```


14

`reg` (or `logic`) in Verilog

- **`reg`** (or **`logic`**) are storage nodes
  - They retain their value till a new value is assigned
  - Unlike a net (**`wire`**) they do not need a driver
  - Can be changed in simulation by assigning a new value
- Signals of type **`reg`** are not necessarily FlipFlops
  - Anything assigned in an *always* or *initial* block must be assigned to a signal of type **`reg`** (or **`logic`**)
  - This inconsistency along with people's inability to keep it straight was part of the motivation for creation of the superset type **`logic`**.
  - Just make you life easier by using type **`logic`**

15

Vectors in Verilog

- Vectors are a collection of bits (i.e. 16-bit wide bus)

```
////////////////////////////////////////////////////////////////
// Define the 16-bit busses going in and out of the ALU //
////////////////////////////////////////////////////////////////
logic [15:0] src1_bus,src2_bus,dst_bus;
```

```
////////////////////////////////////////////////////////////////
// State machine has 8 states, need a 3-bit encoding //
////////////////////////////////////////////////////////////////
reg [2:0] state,nxt_state;
```

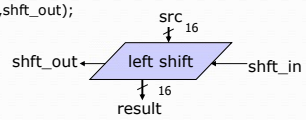
- Bus ≠ Vector (how are they different?)

16

Vectors in Verilog

- Can select parts of a vector (single bit or a range)

```
module lft_shift(src,shft_in,result,shft_out);

input [15:0] src;
input shft_in;
output [15:0] result;
output shft_out;

////////////////////////////////////////////////////////////////////////
// Can access 15 LSB's of src with [14:0] selector                    //
// { , } is a process of concatenating two vectors to form one        //
////////////////////////////////////////////////////////////////////////
assign result = {src[14:0],shft_in};

assign shft_out = src[15];          // can access a single bit MSB with [15]

endmodule
```



17

17

Concatenation { }

```
module off_corr(
  input [15:0] raw_sensor,          // raw signed 16-bit sensor reading
  input [7:0] offset,               // +/- offset to correct sensor reading
  output [15:0] corr_sensor         // corrected sensor reading
);

  assign corr_sensor = raw_sensor + offset;

endmodule
```

- What is wrong with this implementation?
- This is effectively the same as:

```
assign corr_sensor = raw_sensor + {8'h00,offset};
```

18

18

Concatenation { } (fixing it by sign extension)

```
module off_corr(
  input [15:0] raw_sensor,          // raw signed 16-bit sensor reading
  input [7:0] offset,               // +/- offset to correct sensor reading
  output [15:0] corr_sensor         // corrected sensor reading
);

  assign corr_sensor = raw_sensor + {offset[7],offset[7],offset[7],offset[7],
                                     offset[7],offset[7],offset[7],offset[7],offset};

endmodule
```

- Sign extension is a replication of the MSB's right?
- There must be a better way!

19

19

Replication {N{sig}} (sign extension)

- A digit (N) can be a placed in front of a set of { } to replicate whatever scalar or vector quantity is inside the curly braces.
- {2'b00,{3{2'b10}},2'b11} = 10'b0010101011

```
module off_corr(
  input [15:0] raw_sensor,          // raw signed 16-bit sensor reading
  input [7:0] offset,               // +/- offset to correct sensor reading
  output [15:0] corr_sensor         // corrected sensor reading
);

  assign corr_sensor = raw_sensor + {{8{offset[7]}}, offset};

endmodule
```

sign extension
(NOTE: the curly braces around the replicated quantity)

20

20

Other Handy Uses of { }

```
assign signedDiv8 = {{3{value[15]}},value[15:3]};   // ASR 3-bits

assign signedDiv8 = (value>>>3);                    // watch it!

assign rotated = {value[0],value[15:1]};            // RR 1-bit
```

Can also do concatenation on left hand side.

```
module add(
  input [15:0] A_op, B_op,          // operands to be summed
  input Cin,                        // carry in bit
  output [15:0] sum,                // 16-bit sum
  output Cout                       // carry out
);

  assign {Cout,sum} = A_op + B_op + Cin;      // could be 17-bit result

endmodule
```


21

Concatenation in port list

```
module add_concatenate(out, a, b, c, d);
   input [7:0] a;
   input [4:0] b;
   input [1:0] c;
   input d;
   output [7:0] out;

   add8bit(.sum(out), .cout(), .a(a), .b({b,c,d}), .cin(Cin));

endmodule
```

- Vector concatenation is not limited to assign statements. In these examples it is done in a port connection of a module instantiation.

```
module rippleCarryAdder8(
    input  wire [7:0] A,
    input  wire [7:0] B,
    output wire Cout,
    output wire [7:0] S,
);
    wire [6:0] CarryBits;
    fulladder adders[7:0](.A(A), .B(B), .Cin({CarryBits, 1'b0}), .Cout({Cout, CarryBits}), .S(S));
endmodule
```

Example from Aaron Cohen's Verilog Guide

22

End of part I of Lecture02

23

Dataflow Verilog

- The continuous assign statement
  - It is the main construct of Dataflow Verilog
  - It is deceptively powerful & useful
- Generic form:

```
assign [drive_strength] [delay] list_of_net_assignments;

Where:
list_of_net_assignment ::= net_assignment [{,net_assignment}]
& Where:
Net_assignment ::= net_lvalue = expression
```

OK…that means just about nothing to me…how about some examples?

24

Continuous Assign Examples

- Simplest form:

```
// out is a net, a & b are also nets
assign out = a & b;          // and gate functionality
```

- Using vectors

```
logic [15:0] result, src1, src2;     // 3 16-bit wide vectors
assign result = src1 ^ src2;         // 16-bit wide XOR
```

- Can you implement a 32-bit adder in a single line?

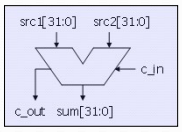
```
wire [31:0] sum, src1, src2;     // 3 32-bit wide vectors
assign {c_out, sum} = src1 + src2 + c_in;     // wow!
```

25

25

Continuous Assign (continued)

- If RHS result changes, LHS is updated with new value
  - Constantly operating ("continuous")
  - It's ***hardware!***
  - What makes it powerful?
    - RHS can use operators: (+, - ,&, |, ^, ~, >>, <<, >>>, ...)



➔ `assign {c_out,sum} = src1 + src2 + c_in;`

26

26

Lets Kick up the Horse Power

- You thought a 32-bit adder in one line was powerful. Lets try a 32-bit MAC…

Design a multiply-accumulate (MAC) unit that computes
Z[31:0] = A[15:0]*B[15:0] + C[31:0]
It sets overflow to one, if the result cannot be represented using 32 bits.

```
module mac(output Z [31:0], output overflow,
           input [15:0] A, B, input [31:0] C);
```

27

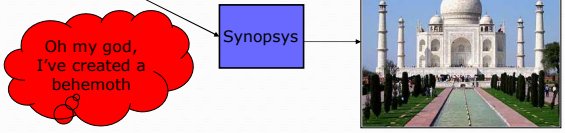
27

Lets Kick up the Horse Power

```
module mac(output Z [31:0], output overflow,
           input [15:0] A, B, input [31:0] C);
  assign {overflow, Z} = A*B + C;
endmodule
```

I am a brilliant genius. I am a HDL coder extraordinaire. I created a 32-bit MAC, and I did it in a single line.

`assign {overflow, Z} = A*B + C;`



28

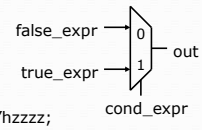
28

Conditional Operator

- This is a favorite!
  - The functionality of a 2:1 Mux
  - **assign** out = conditional_expr **?** true_expr **:** false_expr;

Examples:

```
// a 2:1 mux
assign out = select ? in1 : in0;

// tri-state bus
assign src1 = rf2src1 ? mem[addr1] : 16'hzzzz;

// Either true_expr or false_expr can also be a conditional operator
// lets use this to build a 4:1 mux
assign out = sel[1] ? (sel[0] ? in3 : in2) : (sel[0] ? in1 : in0);
```



29

Conditional assign (continued)

Examples: (nesting of conditionals)

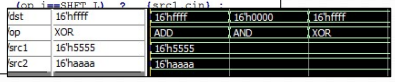
```
localparam add   = 3'b000
localparam and   = 3'b001
localparam xor   = 3'b010
localparam shft_l = 3'b011
localparam shft_r = 3'b100

// an ALU capable of arithmetic,logical, shift, and zero
assign {cout,dst} = (op==add)    ?   src1+src2+cin :
                    (op==and)    ?   {1'b0,src1 & src2} :
                    (op==xor)    ?   {1'b0,src1 ^ src2} :
                    (op==shft_l) ?   {src1,cin} :
                    (op==shft_r) ?   {src1[0],src1[15],src1[15:1]} :
                                     17'h00000;
```

This can be very confusing to read if not coded with proper formating

30

Aside: enum type for readability/debugability

```
module simp_alu(src1,src2,cin,op,dst,cout);

typedef enum logic [2:0] {ADD, AND, XOR, SHFT_L, SHFT_R} op_t

input [15:0] src1,src2;     // input busses
input cin;                  // carry in
input [2:0] op;             // operation select

output [15:0] dst;          // result of ALU
output cout;                // carry output

op_t op_i;

assign op_i = op_t'(op);

// an ALU capable of arithmetic,logical, shift, and zero
assign {cout,dst} = (op_i==ADD)     ?   src1+src2+cin :
                    (op_i==AND)     ?   {1'b0,src1 & src2} :
                    (op_i==XOR)     ?   {1'b0,src1 ^ src2} :
                    (op_i==SHFT_L)  ?   {src1,cin} :
endmodule
```



System verilog allows for the use of enumerated types.

Does not make the code more readable, but does enhance debug because of waveform view.

31

Operators: Arithmetic

- Much easier (and better) than structural!

*	multiply	**	exponent
/	divide	%	modulus
+	add	-	subtract

- Some of these don't synthesize
- Also have unary operators +/- (pos/neg)
- Understand bitsize!
  - Can affect sign of result
  - Is affected by bitwidth of BOTH sides

    assign prod[7:0] = a[3:0] * b[3:0]

32

Operators

Operator Category:	Operators:	Notes:
Shift	>>, <<, >>>	>>> only works of operand and dest signal declared as **signed**
Relational	<,>,<=,>=	Will default to unsigned comparison if operands are not declared as **signed**
Equality	==,!=, ===, !==	=== and !== compare x's and z's explicitly not as don't care. Use in testbenches.
Logical	&&, ||, !	Build clause for **if** or ternary check
Bitwise	&, |, ^, ~	Applies bit by bit, ~ is unary

33

33

Reduction Operators (remember these…they are slick)

- Reduction operators reduce all the bits of a vector to a single bit by performing an operation across all bits.
  - Reduction AND
    - ✓ **assign** all_ones = &accumulator; // are all bits set?
  - Reduction OR
    - ✓ **assign** not_zero = |accuumulator; // are any bits set?
  - Reduction XOR
    - ✓ **assign** parity = ^data_out; // even parity bit

34

34

Saturation Example

- Assume we have a 16-bit signed **sum** and want to saturate it to a 12-bit signed value for use in "downstream" calculations.

```
assign sat12 = (!sum[15] && |sum[14:11]) ? 12'h7FF :
               (sum[15] && !&sum[14:11]) ? 12'h800 :
               sum[11:0];
```

vs

```
assign sat12 = (~sum[15] & |sum[14:11]) ? 12'h7FF :
               (sum[15] & ~&sum[14:11]) ? 12'h800 :
               sum[11:0];
```

35

35

Latches with Continuous Assign

- What does the following statement imply/do?

```
assign q_out = enable ? data_in : q_out;
```

  - It acts as a latch. If enable is high new data goes to q_out. Otherwise q_out maintains its previous value.
  - Ask yourself…It that what I meant when I coded this?
  - It simulates fine…just like a latch

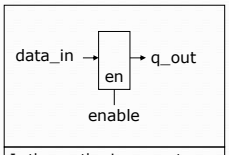
36

36

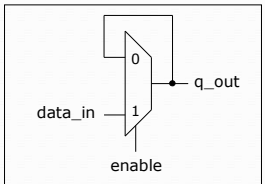
Latches with Continuous Assign

```
assign q_out = enable ? data_in : q_out;
```

- How does it synthesize??



Is the synthesizer smart enough to see this as a latch and pick a latch element from the standard cell library?



Or is it what you code is what you get? A combinational feedback loop. Still acts like a latch. Do you feel comfortable with this?

37

37

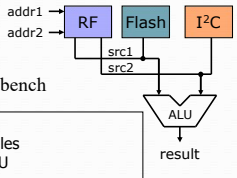
End of part II of Lecture02

38

38

Arrays & Memories

- Can have multi-dimentional arrays
  - **reg** [7:0] mem[0:99][0:3]; // what is this?
- Often have to model memories
  - RF, SRAM, ROM, Flash, Cache
  - Memories can be useful in your test bench




```
input [4:0] addr1, addr2;
input rd,wrt;              // read and write enables
output [15:0] src1,src2;   // source busses to ALU

reg [15:0] reg_file[0:31]; // Register file of 32 16-bit words
.
.
assign src1 = (rd) ? reg_file[addr1] :   // transfer addressed register
              16'hzzzz                   // otherwise back off bus
.
```

39

39

Parameters

- Parameters can make your code more generic/flexible.
  - Defined locally to the module
  - Can be overridden (passed a value in an instantiation)
  - There is another method called **defparam** (don't ever use it) that can override them
- **localparam** → more local than parameter
  - Can't be passed a value
  - **defparam** does not modify

40

40

Parameter Example

```
module adder(a,b,cin,sum,cout);

parameter WIDTH = 8; // default is 8

input [WIDTH-1:0] a,b;
input cin;
output [WIDTH-1:0] sum;
output cout;

assign {cout,sum} = a + b + cin

endmodule
```

```
module alu(src1,src2,dst,cin,cout);
  input [15:0] src1,src2;
  ...
  ////////////////////////////////
  // Instantiate 16-bit adder //
  ////////////////////////////////
  adder #(16) add1(.a(src1),.b(src2),
                   .cin(cin),.cout(cout),
                   .sum(dst));

  ...
endmodule
```

Instantiation of module can override a parameter.

41

41

Parameter Example (ANSI header & named params)

```
module paramReg
  #(parameter WIDTH = 8, CLK2Q = 0)
(
  input [WIDTH-1:0] D,
  input clk,
  input rst_n,
  output reg [WIDTH-1:0] Q
);


  always @(posedge clk, negedge rst_n)
    if (!rst_n)
      Q <= {WIDTH{1'b0}};
    else
      Q <= #CLK2Q D;

endmodule
```

This example uses multiple parameters

If you are using the ANSI declaration style then you declare your parameters here. **NOTE:** this is even before the opening paren

Much like connect by name is clearer and less error prone than by reference order, same can apply to passing of parameters.

```
paramReg #(.WIDTH(16),.CLK2Q(1)) iDUT(.clk(clk), .rst_n(rst_n),
          .D(Din), .Q(Qmon));
```

42

42

Parameter Example (use of $clog2 for address width)

```
module dualPortDEPTHx16(clk,we,waddr,raddr,wdata,rdata);

  parameter DEPTH = 1536;

  input clk;                  // RAM clock.
  input we;                   // active high write enable
  input [$clog2(DEPTH)-1:0] waddr;      // address width dependent on DEPTH
  input [$clog2(DEPTH)-1:0] raddr;      // 11-bit addr for DEPTH = 1536
  input [15:0] wdata;         // data to write
  output reg [15:0] rdata;    // data being read

  reg [15:0] mem [DEPTH-1:0];

  always @(posedge clk) begin
    if (we)
      mem[waddr] <= wdata;
    rdata <= mem[raddr];
  end

endmodule
```

```
dualPortDEPTHx16 #(.DEPTH(3072)) iDP(.clk(clk),.we(we),.waddr(waddr),
                                     .wdata(wdata),.rdata(rdata));
```

- $clog2(3072) will result in 12, so address width is [11:0]

43

43

localparam

```
localparam IDLE =    2'b00;   // idle state of
localparam CONV=     2'b01;   // in this st          nverting
localparam ACCUM =   2'b10;   // in th               raging samples
  .
  .
  .
case (state)
  IDLE : if (start_conv) nx
         else            n
  CONV : if (gt)
         else            NV;
  .
  .
  .
```

Not really a good example! System verilog introduced the enumerated type which is better for state encoding

- Re mming's paper on use of define & param. Posted on web under "reference"

44

44

localparam

- Perhaps a better example of the use of **localparam** would be for defining the value of a constant used in your code that might be subject to change.

```
localparam P_COEFF = 4'h6;     // P_term coefficient
localparam D_COEFF = 7'h38;    // D_term coefficient
```

- This example shows use of a **localparam** for a couple of coefficients used in the calculations for a PID controller.

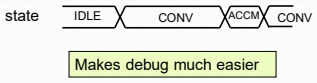
45

45

System Verilog…Better Yet

- System verilog adds an enumerated type.

```
typedef enum reg [1:0] { IDLE, CONV, ACCM } state_t;

state_t state, nxt_state;                // declare state and nxt_state signals
```



Makes debug much easier

```
case (state)
  IDLE : if (strt_cnv) begin
       clr_dac = 1;
       clr_smpl = 1;
       nxt_state = CONV;
  end
  CONV : if (!gt) begin
       inc_dac = 1;
       nxt_state = CONV;
       . . .
```

46

46

Implicit Net Declarations

- When wire is used but not declared, it is *implied*

```
module majority(output out, input a, b, c);
   assign part1 = a & b;
   assign part2 = a & c;
   assign part3 = b & c;
   assign out = part1 | part2 | part3;
endmodule
```

- Defaults to single bit wire
- Breaks compatibility with some tools
- Disable by using `` `default_nettype none `` directive
  - ✓Helps to avoid wasting time on typos

47

47

Don't Take My Word for it (take Aaron Cohen's)

Do this before you start any file. I cannot emphasize this enough.

Add this line to the top of your file, before your module is declared:

```
`default_nettype none
```

Add this line to the bottom of your file, after your endmodule declaration:

```
`default_nettype wire
```

Sign Extender, with a typo

```
1  module signExtender(
2      input  wire [3:0] A,
3      output wire [15:0] A_ext
4  );
5      wire MSB;          // logic works too, instead of wire, in System Verilog
6      assign MSB = A[3];
7
8      assign A_ext = {{12{msb}}, A}; // Repetition/concatenation, explained later on
9  endmodule
```

Do you see the problem? On line 8, instead of using the wire declared as "MSB", it says "msb" in lowercase. Verilog will, by default, infer that you meant to declare some net called msb, and it will use that instead of MSB. The inferred net msb is not driven elsewhere, so A_ext will include 12 bits of undriven HiZ impedance, instead of the most significant bit of A as intended on MSB.

48

48

Useful System Tasks

- $display ➔ Like printf in C. Useful for testbenches and debug

```
$display("At time %t count = %h",$time,cnt);
```

- $stop ➔ Stops simulation and allows you to still probe signals and debug
- $finish ➔ completely stops simulation, simulator relinquishes control of thread.
- Also useful is `include for including code from another file (like a header file)

49

49

Can Mix Styles In Hierarchy!

```
module Add_half_bhv(c_out, sum, a, b);
   output reg sum, c_out;
   input a, b;
   always @(a, b) begin
      sum = a ^ b;
      c_out = a & b;
   end
endmodule
```

```
module Add_full_mix(c_out, sum, a, b, c_in) ;
   output sum, c_out;
   input a, b, c_in;
   wire w1, w2, w3;
   Add_half_bhv AH1(.sum(w1), .c_out(w2),
                    .a(a), .b(b));
   Add_half_bhv AH2(.sum(sum), .c_out(w3),
                    .a(c_in), .b(w1));
   assign c_out = w2 | w3;
endmodule
```

50

50

Hierarchy And Scope

- Parent cannot access "internal" signals of child
- If you need a signal, must make a port!

Example:
Detecting overflow

Overflow =
cout XOR cout6

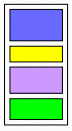
Must output
overflow or cout6!

```
module add8bit(cout, sum, a, b, cin);
   output [7:0] sum;
   output cout;
   input [7:0] a, b,cin;
   wire cout0, cout1,… cout6;
   FA A0(cout0, sum[0], a[0], b[0], cin);
   FA A1(cout1, sum[1], a[1], b[1], cout0);
   …
   FA A7(cout, sum[7], a[7], b[7], cout6);
endmodule
```

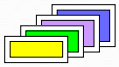
51

51

Hierarchy And Source Code



- Can have all modules in a single file
  - Module order doesn't matter!
  - Good for small designs
  - Not so good for bigger ones
  - Not so good for module reuse (cut & paste)
- Can break up modules into multiple files
  - Helps with organization
  - Lets you find a specific module easily
  - Good for module reuse (add file to project)



52

52