**I) Literals**

<size >'<base><number>

All of the following are the same:

**8'b10101111**

**8'hAF**

**{4'hA,4'b1111}**

**{4'hA,{4{1'b1}}}**

**II) Module declaration**

//Rule: Use separate file for each module (Filename should be module\_name.v):

**module module\_name (A, B, C);**

**input A;** // Inputs to the module

**input [2:0] B;** //Convention: one per line

**output C;** // Output of the module

**...**

**endmodule**

**III)Module instantiation**

**//**In file outer\_module.v

**module outer\_module (in1, in2, out);**

**input in1;**

**input [2:0] in2;**

**output out;**

**module\_name md (.A(in1), .B(in2), .C(out));**

**...**

**endmodule**

//Rule: Use port mapping. The port A of module module\_name is connected to wire in1 in module outer\_module.

**IV) wire vs reg**

a) Rules for usage within module

See also VI

 - LHS of “continuous assign” should be a wire

 - LHS of “procedural assign” should be reg.

b) Rules for ports

 - Ports are by default of type wire (you can override this by declaring it as reg)

 - Input ports cannot be declared as reg (Input ports are always wires)

 - Output ports can be either reg or wire

c) Rules for connecting ports while instantiating

 - When instantiating a module, an output port should be connected to a wire (cannot be connected to reg)

 - When instantiating a module, an input port can be connected to either a reg or a wire

**V) Sequential logic**

 - Rule: To create a flip-flop, instantiate the provided dff.v module.

 - Rule: Do not code sequential logic in any other way.

**dff d0 (**

**.q(flop\_out),**

**.d(flop\_in),**

**.rst(reset),**

**.clk(clock)**

**);**

**VI) Combinational logic**

a) Instantiate the provided logic gates (course webpage)

b) assign statement **(Continuous assign)**

**wire result;**

**assign result = s ? A : B;**

//result – must always be wire (refer above)

//s, A, B – can be wire or reg

c) **Procedural assign** ( blocking = or non-blocking <=)

**reg result;**

**reg err;**

**always @(s or A or B)**

**begin**

**casex(s)**

 **1'b1:**

 **begin**

 **result = A;**

 **err = 1'b0;**

 **end**

 **1'b0:**

 **begin**

 **result = B;**

 **err = 1'b0;**

 **end**

 **default:**

 **begin**

 **result = 1'bx;**

 **err = 1'b1;**

 **end**

**endcase**

**end**

//result, err – must always be reg (refer above).

//s, A, B – can be wire or reg

**VII) parameter**

- Parameterized module definition (in register.v)

**module register(out, in, wr\_en, clk, rst);**

**parameter WIDTH = 1;**

**output [WIDTH-1:0] out;**

**input [WIDTH-1:0] in;**

**input wr\_en;**

**input clk;**

**input rst;**

**dff\_en bits[WIDTH-1:0] (**

 **.q(out),**

 **.d(in),**

 **.en(wr\_en),**

 **.clk(clk),**

 **.rst(rst)**

**);**//dff\_en should instantiate the provided dff.v module

**endmodule**

- Instantiating a parameterized module: with default value of the parameter

**register r0 (............);**

- Instantiating a parameterized module: override the default value

**register #(32) r1 (............);**

**register #(1) r2 (............);**

**VIII)Array instantiation**

The dff instantiation in the example above is an array instantiation. It instantiates many flops with names bits[0], bits[1], ..... bits[n]. Note how the wire 'out' is split across many different instances. Also, the wire 'wr\_en' is connected to multiple ports (one each of each instantiation).

**IX) define**

- Keep defines in a separate file (modname\_config.v):

**`define LAST\_VALUE 4'b1010**

**`define WIDTH 4**

- Include the above file in verilog file (modname.v):

**`include "modname\_config.v"**

**module modname(.........);**

**.......**

**wire [`WIDTH-1:0] carry;**

**assign wire = ~carry & `LAST\_VALUE;**

**.......**

**endmodule**

**X) Allowed keywords**

assign, module, endmodule, input, output, wire, define, parameter

**XI) Keywords allowed with stipulations**

case, casex, reg, always, begin, end

a) case, casex:

 - Have items for all possible combinations. Use default and err should be asserted in default.

 - All outputs of case statement should be assigned in all case items.

 - All nets used in RHS of all assigns within case statement and all nets used as the compare value in case statement should be specified in the sensitivity list.

b) reg:

 - Can only be used to specify outputs of case/casex statement.

c) always, begin, end:

 - Can only be used to introduce case/casex statement.

**XII) Allowed Operators :**

\*In list below, shift operators should have the second argument as constant. (x<<4 is allowed whereas x<<y is not allowed)

~m Inversion

m & n Bitwise AND

m|n Bitwise OR

m^n Bitwise XOR

m~^n Bitwose XNOR

&m ReductionAND

~&m Reduction NAND

|m Reduction OR

~|m Reduction NOR

^m Reduction NOR

~^m Reduction XNOR

m==n Equality

m!=n Inequailty

m===n Identity

m!==n Not Identical

m << const Shift left by const bits

m >> const Shift right by const bits

condition ? m : n Ternary

{m, n} concatenation

{m {n}} replicate n (m times)

**XIII) Testing your design**

a) Design file template to be provided for HW2-HW6 and for the project (in file foo.v);

**module foo (in, out, clk, rst, err);**

**...**

**endmodule**

b) \_hier.v file to be provided for HW2-HW6 and for the project. (foo\_hier.v):

**module foo\_hier ( in, out )**

**...**

**clkrst c0( .clk(clk), .rst(rst), .err(err) );**

**foo f0 ( .out(out), .in(in), .clk(clk),**

 **.rst(rst), .err(err) );**

**endmodule**

c) The testbench \_hier\_bench.v file to be developed and submitted by the student (foo\_hier\_bench.v)

**module foo\_hier\_bench;**

**foo\_hier f0 (....);**

**...**

**endmodule**

**XIV) Scripts**

a) Verilog rules check script (Not foolproof)

**vcheck-all.sh**

b) Name convention check script:

**name-convention-check**

c) Command line verilog simulation script:

**wsrun.pl foo\_hier\_bench \*.v**

**wsrun.pl -wave foo\_hier\_bench \*.v**

d) Synthesis script:

**synth.pl-cmd=synth -type=other -top=foo -opt=yes -file=foo.v,foo\_submodule1.v,foo\_submodule2.v**

**XV) Submission rules**

HW2-HW6 and project submissions require absolute compliance with guidelines. **Here is how you can do almost everything right, but still score ZERO:**

 - By tampering with the provided template for foo.v and/or foo\_hier.v

 - By not submitting the provided testbench components foo\_hier.v file and clkrst.v along with the other verilog files.

 - By not submitting the other provided modules like dff.v, not1.v, nand3.v etc. which are required to compile your design.

 - By submitting a tar file with a directory structure not matching the guidelines.

(eg, if you have a wrapper directory over hw1\_1, hw1\_2 and hw1\_3)

(eg, if your verilog files are located within subdirectories inside hw1\_1)

(eg, if you submit hw1\_1, hw2\_2, hw3\_3 when you are asked to submit hw1\_1, hw1\_2 and hw1\_3)

 - By submitting .tar.gz or .zip when you are asked to submit .tar

 - By not running vcheck on verilog files or by not checking the results after running the script.

 - By not turning in .vcheck.out files for each .v file (except the testbench components and provided module.)

 - By forgetting to click on the 'Submit' button in dropbox after clicking the 'Upload' button.

 **XVI) Modelsim waveform viewing/debugging cheats?**Course bonus points abound ☺

**Code example (dyser\_stage.v)**

`include "dyser\_config.v"

// dff\_rn is a d-flip-flop with negative reset

// dff\_rne is a d-flip-flop with negative reset and enable

module stage(

 /\* inputs \*/

 ready\_in, valid\_in, credit\_in, data\_in, clk, rst\_n,

 /\* outputs \*/

 credit\_out, data\_out, valid\_out, ready\_out, err

 );

 parameter ID = 0;

 parameter EDGE = 0;

 input ready\_in;

 input valid\_in;

 input credit\_in;

 input [`DATA\_WIDTH:0] data\_in;

 input clk;

 input rst\_n;

 output credit\_out;

 output [`DATA\_WIDTH:0] data\_out;

 output valid\_out;

 output ready\_out;

 output err;

 // wires and reg

 reg credit\_out;

 reg data\_en;

 reg [`DATA\_WIDTH:0] data;

 reg valid;

 reg ready\_out;

 reg state;

 parameter CN = 1'b0;

 parameter NR = 1'b1;

 // state + next state logic

 wire next\_state;

 dff\_rn state\_ff( .din(next\_state), .q(state),

 .rst\_n(rst\_n) );

 assign next\_state = (state == CN) ? ready\_in ? NR : CN :

 /\*state == NR\*/ credit\_in ? CN : NR;

 // output logic (Mealy)

 always @(state or credit\_in or ready\_in)

 //always @(\*)

 case ({state,credit\_in,ready\_in})

 //for state CN

 3'b0\_0\_0:

 begin

 ready\_out = 1'b0;

 data\_en = 1'b0;

 credit\_out = 1'b1;

 err = 1’b0;

 end

 3'b0\_0\_1:

 begin

 ready\_out = 1'b0;

 data\_en = 1'b1;

 credit\_out = 1'b1;

 err = 1’b0;

 end

 3'b0\_1\_0:

 begin

 ready\_out = 1'b0;

 data\_en = 1'b0;

 credit\_out = 1'b1;

 err = 1’b0;

 end

 3'b0\_1\_1:

 begin

 ready\_out = 1'b0;

 data\_en = 1'b1;

 credit\_out = 1'b1;

 err = 1’b0;

 end

 3'b0\_0\_0:

 begin

 ready\_out = 1'b1;

 data\_en = 1'b0;

 credit\_out = 1'b0;

 err = 1’b0;

 end

 3'b0\_0\_1:

 begin

 ready\_out = 1'b1;

 data\_en = 1'b0;

 credit\_out = 1'b0;

 err = 1’b0;

 end

 3'b0\_1\_0:

 begin

 ready\_out = 1'b1;

 data\_en = 1'b0;

 credit\_out = 1'b0;

 err = 1’b0;

 end

 3'b0\_1\_1:

 begin

 ready\_out = 1'b1;

 data\_en = 1'b0;

 credit\_out = 1'b0;

 err = 1’b0;

 end

 default:

 begin

 //$display("ERROR time: %d", $time );

 ready\_out = 1'b0;

 data\_en = 1'b0;

 credit\_out = 1'b0;

 err = **1’b1;**

 end

 endcase

 // data and valid FF

 dff\_rne\_data\_width data\_ff( .din(data\_in)

 .q(data)

 .en(data\_en)

 .rst\_n(rst\_n));

 dff\_rne valid\_ff( .din(valid\_in)

 .q(valid)

 .en(data\_en)

 .rst\_n(rst\_n));

 assign data\_out = data;

 assign valid\_out = valid;

endmodule