//LSU EE 3755 - Fall 2003 Computer Organization

// Based on 2002

/// Verilog Notes 1 --- Verilog Basics

///Contents

///References

//:P: Palnitkar, ?Verilog HDL?

//:Q:Qualis, ?Verilog HDL Quick Reference Card Revision 1.0?

//:PH: Patterson & Hennessy, ?Computer Organization & Design?

//

// This is what Verilog code looks like.

// **You are not supposed to understand any of this at this time.**

// You will understand this later as semester goes on.

```
//:P
module ripple_carry_counter(q,clk,reset);

output [3:0]  q;
input clk, reset;
T_FF tff0(q[0],clk,reset);
T_FF tff0(q[1],q[0],reset);
T_FF tff0(q[2],q[1],reset);
T_FF tff0(q[3],q[2],reset);

endmodule

module T_FF(q,clk,reset);

output q;
input clk,reset;
wire d;

D_FF dffo(q,d,clk,reset);
not n1(d,q); //not is a Verilog-provided primitive. case sensitive
endmodule

//module D_FF with synchronous reset
module D_FF(q,d,clk,reset);
```

```
output q;
input d,clk,reset;
reg q;

always @(posedge reset or negedge clk)

if (reset)
        q = 1'b0;
// module D_FF with synchronous reset
else
        q = d;
endmodule

// Test bench or Stimulus Block
module stimulus;

reg clk;
reg reset;
wire [3:0] q;

//instantiate the design block
ripple_carry_counter r1(q, clk, reset);

// Control the clk signal that drives the design block.Cycle time =10
```

```
initial
        clk = 1'b0; //set clk to 0
always
        #5 clk = ~clk; // toggle clk every 5 time units

//Control the reset signal that drives the design block
//rest is asserted from 0 to 20 and from 200 to 220.
initial
begin
        reset = 1'b1;
        #15 reset = 1'b0;
        #180 reset = 1'b1;
        #10 reset = 1'b0;
        #20 $finish; //terminate the simulation
end
// Monitor the outputs
initial
        $monitor($time, "Output q = %d",q);
endmodule

//Output of the simulation
//#  0 Output q =0
//# 20 Output q =1
//# 30 Output q =2
```

```
//# 40 Output q =3

//# 50 Output q =4

//# 60 Output q =5

//# 70 Output q =6

//# 80 Output q =7

//# 90 Output q =8

//#100 Output q =9

//#110 Output q =10

//#120 Output q =11

//#130 Output q =12

//#140 Output q =13

//#150 Output q =14

//#160 Output q =15

//#170 Output q =0

//#180 Output q =1

//#190 Output q =2

//#195 Output q =0

//#210 Output q =1

//#220 Output q =2
```

//Typical Ripple Carry Counter Output.

// clock 01010101010101010101010101010100

// q(q0) 00110011001100110011001100110011001100

// q(q1) 00001111000011110000111100000111100

// q(q2) 00000000111111110000000011111111100

// q(q3) 00000000000000001111111111111111100

// if you don't understand any of above so far, it is OK.

//you are not supposed to understand.

///Module

//:P:4.1 Good description.

// Describes a part,

// can be simpler than a gate

// or more complex than an entire computer.

//A Module is the most basic element of Verilog

// It represents a design or part of a design

//Modules have inputs, outputs, and perform ?things? inside

//Modules ?run? concurrently or in parallel.

//A Module is defined once (see example) and can be /instantiated/.

//used by other modules, many times.

//: Example:

//

//A definition of a module with no inputs or outputs, and that does nothing.

```
module my_first_module();
endmodule
// : keywords: module , endmodule

////////////////////////////////////////////////////////////////////////////////////////////////////
///Module Ports
//:P:4.2
// A /port/ is either an input or an output [or an inout] of a moduloe.

//Each port has:
// a name E.g., clock, q, my_input
// a direction : input, output, [or inout]
//
//: Example:
//
//A module with two ports, an input and an output.
// Though it has two ports it still does nothing.
module my_second_module(my_first_output,my_first_input);
        input my_first_input;
        output my_first_output;
endmodule
// :keywords : input, output
        ///more information
//
```

```
//The port declarations must be at the top of a module.
//A module can have any number of ports.
//Each port name must appear near the module name(my_second_module)
// and with a port direction below (input my_first_input).
//The input and output keywords can be followed by any positive of names.
//Inputs and outputs can appear in any order.
// In class outputs will appear first.

//: Example:
//
// A module with four ports.  Also does nothing.

module my_third_module(a,b,c,d);
        input a,c;
        output b;
        output d;
endmodule

////////////////////////////////////////////////////////////////////////////////////////////////////
/// Wire(Nets)
//:P:3.2.2
// Objects of type /wire /(among other things) are used to connect
//items within a module.
//
```

```
//Wires don?t store their values
//Wires may be assigned with an assignment statement


//: Example:
//
//A module with a wire, c, declared. It doesn?t connect to anything.

module my_first_wire_module(a,b);
        output a;
        input b;

        wire c;
endmodule

//: Example:
//
// A module with three wires declared: c,d, and e. None connect
// to anything.

module my_second_wire_module(a,b);
        output a;
        input b;
        wire c,d;
```

```
        wire e;
endmodule
// : Keyword:wire
// Used to declare wires.

// More Information
//
//The wire declarations can appear in many places in a module.
//wire is one of several /net/ data types.

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
//Gates
//: P: 5.1

// A / gate/ is a /primitive/ component.
//
// Some Verilog gates : /and/, /or/, /xor/.
//
//  /primitive/: something that is not defined in terms of something else.

//: Example:
//
//A module implementing : x = ab + c using gates.
```

```
module my_gates(x,a,b,c);
        input a,b,c;
        output x;
        wire ab;
        and and1(ab,a,b);
        or or1(x,ab,c);
endmodule


//#################here comes fig1####################


/// More Information
//
//Gates are instantiated in modules.
//      E.g.: and and1(ab, a, b);
//The instantiation specifies a gate type (and), a name(and1),
//      and a list of port connections (ab, a, b).
//
//It does not matter what order gates are instantiated in. See two examples below.
//
///Gates to be used in class:
//  :Keywords : and . or, not, xor, nand, nor , xnor.
```

```
//:Example:
//
module my_second_gate(x,a,b,c);
        input a,b,c;
        output x;
//Wire declarations can be omitted but don?t . We could have used anything, instead of ?ab?, but
//?ab? helps the person reading this description remember that ?ab? is the output of an AND gate with
inputs
// ?a? and ?b?.
wire ab;
//Code below /instantiates/ an AND gate. Instance name is ?and1?.
//Output is always first, followed by 1 or more inputs. (Gates can have
// any number of inputs.)
and and1(ab, a,b);
or or1(x,ab,c);

endmodule
//: Example:
//
//Same example except for different instantiation order. The
//different order does NOT matter. Choose order for human
//readability.

module my_third_gate(x,a,b,c);
input a,b,c;
```

```
        output x;
wire ab;

or or1(x,ab,c);

and and1(ab, a,b);
endmodule
//:Example:
//
//Two modules implementing an exclusive or gate. The
//first module uses AND, OR, and NOT gates , and is shown to
//illustrate how gates are used . The second module uses
//Verilog?s xor gate.

//################fig.2 comes here############################
module my_xor_module(x,a,b);
        output x;
        input a,b;

        wire na, nb, na_b, a_nb;

        not n1(na,a);
        not n2(nb,b);
```

```
        and a1(na_b,na,b);
        and a2(a_nb,a,nb);

        or o1(x,na_b,a_nb);
endmodule

module my_xor_module2(x,a,b);
        output x;
        input a,b;
        xor x1( x,a,b);

endmodule

////////////////////////////////////////////////////////////////////////////////////////////////////
/// Module Instantiation

//:P:4.1, 4.2
//A module is used by another module by  /instantiating / it.
//
//A module instantiation is similar to a gate instatiation.
//
//:Example:
//
```

```
// An xnor module which uses the xor module defined above and a not gate.
module my_xnor_module(x,a,b);
        input a,b;
        output x;

        my_xor_module x1(y, a,b);
        not n1(x,y);
        //Here a module defined above , my_xor_module, is being used, or
        //instantiated, the name of he instance is x1.
endmodule

///More Information
// The differences between a gate instantiation and a module
//   instantiation should not make a difference in this class.
//

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///Logic Value Set

//:P: 3.2.1
//
// Wires can take on four values:
// 0: Logic 0, false, off
// 1: Logic 1, true, on
```

```
// z: High impedance
//      Nothing driving wire .(E.g. not connected to anything or output hi Z.)
//x: Unknown.
//
//      Simulator cannot determine value.
//      Sometimes this means two sources driving wire.

//:Example:
//Module below generates the four logic values. See comments.
//Assume that input ?a? is either zero or one. The purpose of this
//module is to generate the four physical values for demonstration
//purposes.

module values(v0,v1,vx,vz,a);
        input a;
        output v0,v1,vx,vz;

        wire na;

        not n1(na,a);

        and a1(v0,a,na);  // v0 will be logic 0.

        or o1(v1,a,na);    //v1 will be logic 1.
```

```
        not n2(vx,a);
        not  n3(vx,na);   //vx is driven by a 0 and 1, so its value is x.
                          //vz unconnected, so its value is z.
endmodule

// Testbench for values. Material for this testbench
//not yet covered. Ignore it if you like.

 module demo_values();
        wire v0, v1, vx,vz;
        reg in;
        values d1(v0,v1,vx,vz,in);

        initial begin

                in = 0;

               #5;

                in = 1;

                #5;
         end
```

```
endmodule

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

///Binary Full Adder

//:P: 5.1.3

//A /binary full adder // adds three one-bit integers.

//

// It is usually used as part of  a larger, say 32- bit, adder.


//Inputs : a, b, cin(carry in)

//Outputs:sum,cout

//

//Computes a+b+cin and sets

//   sum to the LSB of a+b+cin and

//   cout to the MSB of a+b+cin.

//  Truth Table


// a b  cin | cout sum

// 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

// cout = a b + a cin  + b cin
// sum = a xor b xor cin

//Won?t use xor here though.

//: Example:
//
//Example of a binary full adder. There will be several other
// example of this circuit.
//###############fig 3  comes here#################
module bfa_structural(sum,cout ,a,b,cin);
        input a, b, cin;
        output sum, cout;

        wire term001,term010, term100,term111;
        wire ab, bc, ac;
        wire na, nb, nc;
```

```
// 1 1 1 | 1 1


// cout = a b + a cin + b cin

// sum = a xor b xor cin


//Won?t use xor here though.


//: Example:
//
//Example of a binary full adder. There will be several other
// example of this circuit.
###############fig 3  comes here#################
module bfa_structural(sum,cout ,a,b,cin);
        input a, b, cin;
        output sum, cout;


        wire term001,ter,010, term100,term111;
        wire ab, bc, ac;
        wire na, nb, nc;


        or o1(sum,term001, term010,term100, term111);
        or  o2(cout,ab,bc,ac);


        and a1(term001, na, nb, cin);
```

```
    and a2(term010, na, b, nc);
    and a3(term100, a, nb, nc);
    and a4(term111, a, b, cin);

    not n1(na,a);
    not n2(nb,b);
    not n3(nc, c);

    and a10(ab,a,b);
    and a11(bc,b,cin);
    and a12(ac,a,cin);

endmodule
```