Lect 20
Verilog HDL

CS221: Digital Design

Dr. A. Sahu
Dept of Comp. Sc. & Engg.
Indian Institute of Technology Guwahati

Outline

- HDL Programming : Verilog HDL
- HDL Rules
- HDL Module and Examples
- HDL levels : Data flow, Structural and Behavioral, UDP
- Testing and Simulation

Boolean Expressions

- Use reserved word **assign**
- AND (&), OR (|) and NOT (~)
- Example:

```
// Boolean Circuit representation
module Boolean Circuit (E,F,A,B,C,D);
        output      E,F;
        input       A,B,C,D;

        assign      E= A| (B&C)|(~B&D);
        assign      F= (~B &C) | (B& ~C & ~D);
endmodule
```


User Defined Primitives

- System primitives: **and, or, nand, xor**
- One way is to define own primitive by a Truth Table….
- Use **primitive** and **endprimitive** to create a UDP
- It is declared with the reserved word **primitive** followed by a name and port list
- One output and it must be listed first in the port listing and following the output declaration

User Defined Primitives

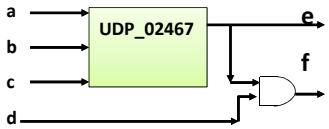
- Any number of inputs, however the order given in the port declaration must be the same as the Table
- The table must start with the reserved word **table** and end with **endtable**
- The values of the inputs are listed in order and separated from output by : the line ends with ;

User Defined Primitives (UDP)

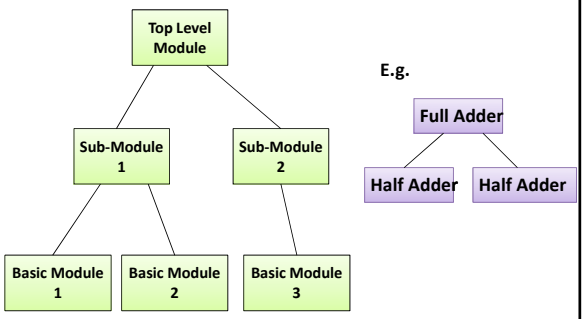
```
primitive UDP_02467 (D,A,B,C);
output      D;
input       A,B,C;
// Truth Table for D= f( A, B ,C ) = Σ m(0,2,4,6,7);
table // A   B   C   :   D   // headers
      0     0   0   :   1;
      0     0   1   :   0;
      0     1   0   :   1;
      0     1   1   :   0;
      1     0   0   :   1;
      1     0   1   :   0;
      1     1   0   :   1;
      1     1   1   :   1;
endtable
endprimitive
```

Calling of UDP

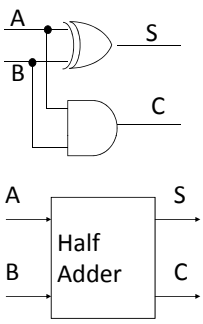
```
// Verilog model: Circuit instantiation of Circuit_UPD_02467
module Circuit_withUDP_02467 (e,f,a,b,c,d);
   output      e,f;
   input       a,b,c,d;

   UDP_02467  (e,a,b,c);

   and         (f,e,d);
endmodule
```



Hierarchical Design



Example: Half Adder




```
module half_adder(S,C,A,B);
output S, C;
input A, B;

wire S, C, A, B;

assign S = A ^ B;
assign C = A & B;

endmodule
```


How to build and test a module

- **Construct a "test bench" for your design**
  - Develop your hierarchical system within a module that has input and output ports (called "**design**" here)
  - Develop a separate module to generate tests for the module ("**test**")
  - Connect these together within another module ("**testbench**")

How to build and test a module

```
module testbench ();
   wire  l, m, n;

   design   d (l, m, n);
   test     t (l, m);

   initial begin
   //monitor and display
      …
```

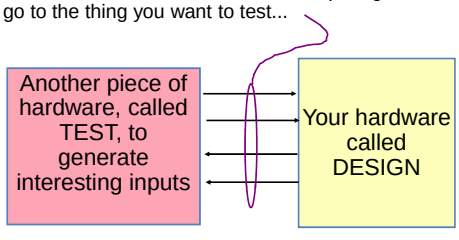
```
module design (a, b, c);
   input    a, b;
   output   c;
   …
```

```
module test (q, r);
   output   q, r;

initial begin
  //drive the outputs with signals
```


Another view of this

- 3 chunks of verilog, one for each of:

TESTBENCH is the final piece of hardware which connect DESIGN with TEST so the inputs generated go to the thing you want to test...



Verilog Examples: TB of HA

Module testAdd generated inputs for module halfAdd and displayed changes. Module halfAdd was the design

```
module tBench;
   wire  su, co, a, b;
   halfAdd  ad(su, co, a, b);
   testAdd  tb(a, b, su, co);
endmodule
```

```
module halfAdd (sum, cOut, a, b);
   output sum, cOut;
   input  a, b;
   xor    #2      (sum, a, b);
   and    #2      (cOut, a, b);
endmodule
```

13

Verilog Examples: TB of HA

```
module testAdd(a, b, sum, cOut);
   input       sum, cOut;
   output      a, b;
   reg         a, b;

   initial begin
      $monitor ($time,,  "a=%b, b=%b, sum=%b,cOut=%b",
                         a, b, sum, cOut);
      a = 0; b = 0;
      #10 b = 1;
      #10 a = 1;
      #10 b = 0;
      #10 $finish;
   end
endmodule
```

14

The test module

- It's the test generator
- $monitor
  - prints its string when executed.
  - after that, the string is printed when one of the listed values changes.
  - only one monitor can be active at any time
  - prints at end of current simulation time

15

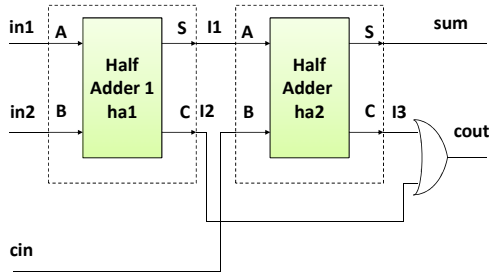
The test module (continued)

- Function of this tester
  - at time zero, print values and set a=b=0
  - after 10 time units, set b=1
  - after another 10, set a=1
  - after another 10 set b=0
  - then another 10 and finish

```
module testAdd(a, b, sum, cOut);
   input       sum, cOut;
   output      a, b;
   reg         a, b;

   initial begin
      $monitor ($time,,  "a=%b,
b=%b, sum=%b,cOut=%b",
                a, b, sum, cOut);
      a = 0; b = 0;
      #10 b = 1;
      #10 a = 1;
      #10 b = 0;
      #10 $finish;
   end
endmodule
```

16

Example: Full Adder



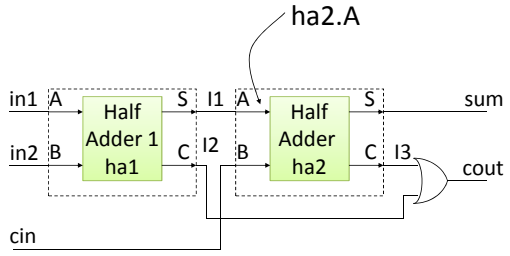
Example: Full Adder

```
module full_adder(sum,cout,in1,in2,cin);
output   sum, cout;
input    in1, in2, cin;
wire sum, cout, in1, in2, cin;
wire I1, I2, I3;

half_adder ha1(I1, I2, in1, in2);
half_adder ha2(sum, I3, I1, cin);

assign cout = I2 || I3;

endmodule
```

Module name

Instance name

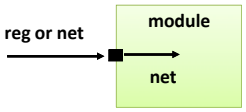
Hierarchical Names



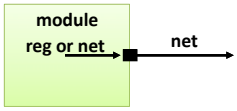
Remember to use instance names, not module names

Port Assignments

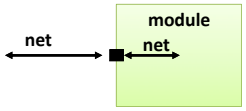
- **Inputs**



- **Outputs**



- **Inouts**



Full Adder and 4 bit adder

$S = A \oplus B \oplus Cin \quad Cout = A.B + B.Cin + A.Cin$

```
module FA(output S, Cout, input A,B,Cin);
        assign  S=A^B^Cin;
        assign  S= (A&B)  | (B &Cin) | (A&Cin);
endmodule
```

```
module 4BitRCA (output[3:0] S, output C4, input[3:0] A,B,
input C0);
        wire C1. C2, C3;
        //Instantiate Chain of Full Adder
        FA      FA0 (S[0], C1,  A[0],  B[0],  C0);
        FA      FA1 (S[1], C2,  A[1],  B[1],  C1);
        FA      FA2 (S[2], C3,  A[2],  B[2],  C2);
        FA      FA3 (S[3], C4,  A[3],  B[3],  C3);
endmodule
```

Continuous Assignements

a closer look

- Syntax: `assign #del <id> = <expr>;`

optional — net type !!

- Where to write them:
  - inside a module, outside procedures
- Properties:
  - **they all execute in parallel**
  - **are order independent**
  - **are continuously active**

Verilog HDL Models

- HDL model specifies the relationship between input signals and output signals
- HDL uses special constructs to describe hardware concurrency, parallel activity flow, time delays and waveforms
- Verilog code for a AND gate

```
module and_gate(y, x1, x2);
input       x1, x2;
output      y;

        and(y, x1, x2);

endmodule
```


Verilog Examples

```
module Add_half (sum, c_out, a, b);
  output        sum, c_out;
  input         a, b;

  wire          c_out_bar;

  xor     G1 (sum, a, b);
  nand    G2 (c_out_bar, a, b);
  not     G3 (c_out, c_out_bar);
endmodule
```

*** the instance name of Verilog primitives is optional.**

Verilog Example: behavioral model

```
module adder_4_RTL (a, b, c_in, sum, c_out);
    output [3:0]    sum;
    output          c_out;
    input [3:0]     a, b;
    input           c_in;

    assign {c_out, sum} = a + b + c_in;
endmodule
```


Behavioral Model - Procedures (i)

- Procedures = sections of code that we know they execute sequentially
- Procedural statements = statements inside a procedure (they execute sequentially)
- e.g. another 2-to-1 mux implem:

Execution Flow

```
begin
if (sel == 0)
      Y = B;
 else
      Y = A;
end
```

Procedural assignments: Y must be reg !!

Behavioral Model - Procedures (ii)

- Modules can contain any number of procedures
- Procedures execute in parallel (in respect to each other)
- And can be expressed in two types of blocks:
  - **initial → they execute only once**
  - **always → they execute for ever (until simulation finishes)**

"Initial" Blocks

Start execution at sim time zero and finish when their last statement executes

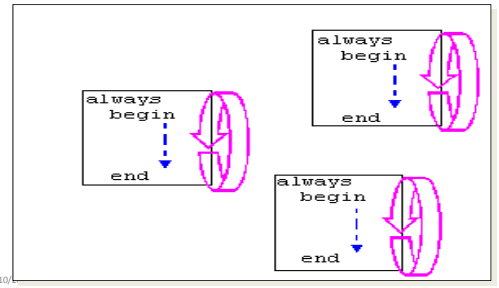
```
module nothing;

initial
 $display("I'm first");

initial begin
 #50;
 $display("Really?");
 end

endmodule
```

Will be displayed at sim time 0

Will be displayed at sim time 50

"Always" Blocks

- Start execution at sim time zero and continue until sim finishes



Events (i)

```
always @(signal1 or signal2 or ..)
begin
        ..
end
```

execution triggers every time any signal changes

```
always @(posedge clk)
begin
        ..
end
```

execution triggers every time clk changes from *0* to *1*

```
always @(negedge clk)
begin
        ..
end
```

execution triggers every time clk changes from *1* to *0*

Examples: always@

```
module half_adder(S, C, A, B);
output      S, C;
input       A, B;

reg         S,C;
wire        A, B;

always @(A or B) begin
        S = A ^ B;
        C = A && B;
end

endmodule
```

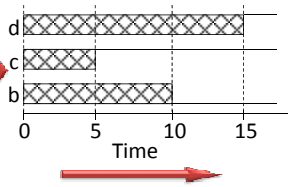

Events (ii) : wait

```
always begin
  wait (ctrl)
  #10 cnt = cnt + 1;
  #10 cnt2 = cnt2 + 2;
end
```

execution loops every time ctrl = 1 (level sensitive timing control)

Timing (i)

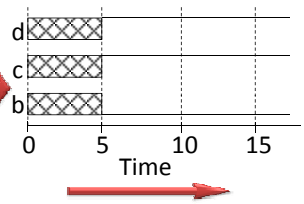
```
initial begin
    #5 c = 1;
    #5 b = 0;
    #5 d = c;
end
```



Each assignment is blocked by its previous one

Timing (ii)

```
initial begin
    fork
    #5 c = 1;
    #5 b = 0;
    #5 d = c;
    join
end
```



Assignments are not blocked here

Procedural Statements: if

```
module mux4_1(out, in, sel);
output      out;
input[3:0]  in;
input[1:0]  sel;

reg         out;
wire [3:0]  in;
wire [1:0]  sel;

always @(in or sel)
    if (sel == 0)
        out = in[0];
    else if (sel == 1)
        out = in[1];
    else if (sel == 2)
        out = in[2];
    else
        out = in[3];
endmodule
```

Procedural Statements: case

```
module mux4_1(out, in, sel);
output      out;
input[3:0]  in;
input[1:0]  sel;

reg         out;
wire [3:0]  in;
wire [1:0]  sel;

always @(in or sel)
    case (sel)
        0: out = in[0];
        1: out = in[1];
        2: out = in[2];
        3: out = in[3];
    endcase
endmodule
```

Procedural Statements: for

```
module count(Y, start);
output [3:0] Y;
input        start;

reg [3:0]    Y;
wire         start;
integer      i;

initial
             Y = 0;

always @(posedge start)
    for (i = 0; i < 3; i = i + 1)
        #10 Y = Y + 1;
endmodule
```


Procedural Statements: while

```
module count(Y, start);
output [3:0] Y;
input        start;

reg [3:0]    Y;
wire         start; integer   i;

initial
             Y = 0;
always @(posedge start)
    i = 0;
    while (i < 3) begin
        #10 Y = Y + 1;
        i = i + 1;
        end
    end
endmodule
```


Procedural Statements: repeat

```
module count(Y, start);
output [3:0] Y;
input        start;

reg [3:0]    Y;
wire         start;
integer      i;

initial
             Y = 0;

always @(posedge start)
     repeat (4) #10 Y = Y + 1;
endmodule
```


Procedural Statements: forever

forever stmt;

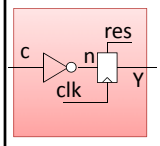
Executes until sim finishes

```
//clock generation in test modules
module test;
reg clk;
initial begin
   clk = 0;
   forever #10 clk = ~clk;
end

other_module1 o1(clk, ..);
other_module2 o2(..,
clk,..);
endmodule
```

T_{clk} = 20 time units

Mixed Model

Code contains various both structure and behavioral styles




```
module simple(Y, c, clk, res);
output   Y;
input    c, clk, res;

reg      Y;
wire     c, clk, res, n;

not(n, c); // gate-level

always @(res or posedge clk)
    if (res)
        Y = 0;
    else
        Y = n;
endmodule
```


System Tasks

Always written inside procedures

- **$display**(“..”, arg2, arg3, ..);
  - much like printf(), displays formatted string in std output when encountered
- **$monitor**(“..”, arg2, arg3, ..);
  - $display(), but .. displays string each time any of arg2, arg3, .. Changes
- **$stop;** → suspends sim when encountered
- **$finish;** → finishes sim when encountered

System Tasks

Always written inside procedures

- **$fopen**("filename");
  - returns file descriptor (integer);
  - Then you can use $fdisplay(fd, "..", arg2, arg3, ..); or $fmonitor(fd, "..", arg2, arg3, ..); to write to file
- **$fclose**(fd); → closes file
- **$random**(seed); → returns random integer;
  - integer as a seed

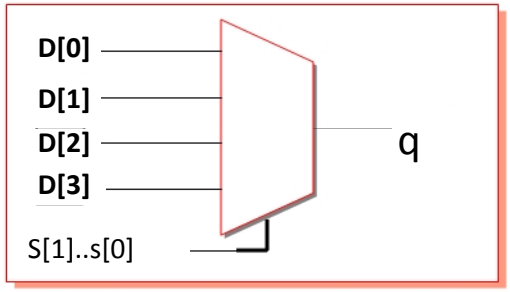
$display & $monitor string format

Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format
%f or %F	Display real number in decimal format
%g or %G	Display scientific or decimal, whichever is shorter

Veriety of Coding Style in Verilog

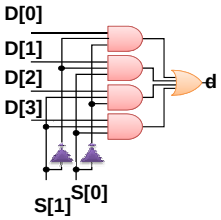
4x1 Mux Example

4x1 Multiplexer



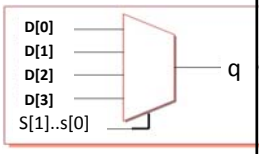
4x1 Multiplexer : structural

```
module mux1( S, d, q );
        input[1:0] S;
        input[3:0] d;
        output     q;

wire q, q1, q2, q3, q4, NS0, NS1;
wire[1:0] S;
wire[3:0] d;

not    n1( NS0, S[0] );
not    n2( NS1, S[1] );
and    a1( q1, NS0, NS1, d[0] );
and    a2( q2, S[0], NS1, d[1] );
and    a3( q3, NS0, S[1], d[2] );
and    a4( q4, S[0], S[1], d[3] );
or     o1( q, q1, q2, q3, q4 );
endmodule
```



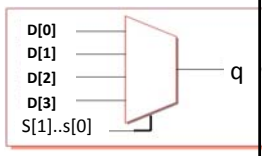
4x1 Multiplexer : Data Flow

```
module mux1( S, d, q );
        input[1:0] S;
        input[3:0] d;
        output     q;

        wire    q;
        wire[1:0] S;
        wire[3:0] d;

        assign q = d[S];
endmodule
```



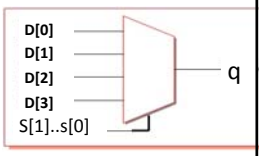
4x1 Multiplexer : Data Flow

```
module mux1( S, d, q );
        input[1:0] S;
        input[3:0] d;
        output     q;

        wire     q;
        wire[1:0] S;
        wire[3:0] d;
        assign  q = (S == 0 )? d[0] : ( S == 1 )? d[1]
                  : ( S == 2 )? d[2] : d[3];
endmodule
```



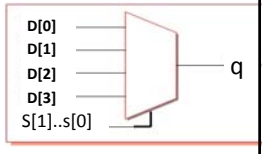
4x1 Multiplexer: Behav-Data Flow

```
module mux1( S, d, q );
        input[1:0] S;
        input[3:0] d;
        output     q;

        reg q;
        wire[1:0] S;
        wire[3:0] d;

always @(d or S)
 begin
    q = ( ~S[0] & ~S[1] & d[0] ) |
        (   S[0] & ~S[1] & d[1] ) |
        ( ~S[0] &   S[1] & d[2] ) |
        (   S[0] &   S[1] & d[3] ) ;
end
endmodule
```



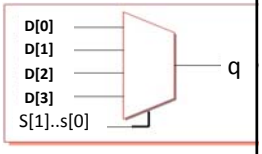
4x1 Multiplexer

```
module mux1( S, d, q );
        input[1:0] S;
        input[3:0] d;
        output     q;

        reg q;
        wire[1:0] S;
        wire[3:0] d;
always @(d or S)
        q = d[select];
endmodule
```



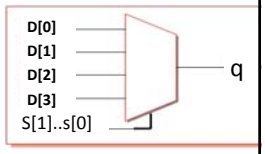
4x1 Multiplexer

```
module mux1( S, d, q );
        input[1:0] S;
        input[3:0] d;
        output     q;

        reg q;
        wire[1:0] S;
        wire[3:0] d;

always @(d or S)
        begin
        if( S == 0) q = d[0];
        if( S == 1) q = d[1];
        if( S == 2) q = d[2];
        if( S == 3) q = d[3];
endmodule
```



4x1 Multiplexer

```
module mux1( S, d, q );
        input[1:0] S;
        input[3:0] d;
        output     q;

        reg q;
        wire[1:0] S;
        wire[3:0] d;

always @(d or S)
 begin case ( S )
        0 : q = d[0];
        1 : q = d[1];
        2 : q = d[2];
        3 : q = d[3];
endcase
endendmodule
```



2x1 Mux using UDP

```
primitive Mux (y, a, b, sel); // combinational UDP
        output y;
        input a, b, sel;
        table // a b sel : y
        0 ? 0 : 0;
        1 ? 0 : 1;
        ? 0 1 : 0;
        ? 1 1 : 1;
endtable
endprimitive
```