EE2001 - Digital systems lab

Janakiraman Viraraghavan and Vinita Vasudevan

Verilog data, modules and Ports

Data Values:

- 0, 1, X (unknown), Z (tristate)
- a[3:0] $\Longrightarrow$ a[0] is the LSB and a[3] is the MSB

```
module Circuit(out1, out2, Inp
input Inp1, Inp2;
output out1, out2;
 HDL Modelling of functionality
endmodule
```

```
module A(a, b, sum, cout);
input[3:0] a, b;
output[3:0] sum;output cout;
 HDL Modelling of functionality
endmodule
```

Half Adder - Gate level and Data flow model

a	b	sum	cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

- Gates and, or, not, xor, nand, nor, xnor predefined
- Bit operators similar to C
- The inputs a and b are in the “sensitivity list”. If there is any change in a or b, the corresponding output is re-evaluated.

Gate level Model

```
//Half adder module

module ha(a, b, sum, cout);

input a, b;
output sum, cout;

xor x1(sum, a, b);
and a1(cout, a, b);

endmodule
```

Data flow model of half adder:

```
//Half adder module

module ha(a, b, sum, cout);

input a, b;
output sum, cout;

assign sum = a ^ b;
assign cout= a & b;

endmodule
```

Full Adder Truth Table

a	b	cin	sum	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\begin{aligned} sum(a, b, cin) &= \Sigma(1, 2, 4, 7) \\ &= a \oplus b \oplus cin \\ cout(a, b, cin) &= \Sigma(3, 5, 6, 7) \\ &= a\ b + b\ cin + a\ cin \end{aligned}$$

Hierarchical modelling

A module can contain other modules through module instantiation.

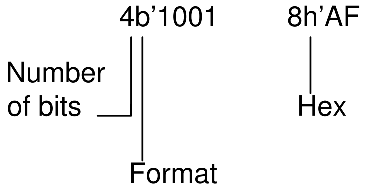
- Modules are connected together using nets
- Ports are attached to nets either by position or name

```
module FA(ain, bin, carryin, sum, cout);

input ain, bin, carryin;
output sum, cout;
wire s1, c1, c2;  These are nets connecting the half adders and OR gates

 ha ha1(.a(ain), .b(bin), .cout(c1), .sum(s1));// Bind to signals by name
ha ha1(ain, bin, s1, c1);  // By position
ha ha2( s1, carryin, sum, c2);
or or1( cout, c1, c2);
// assign cout = c1 | c2;
endmodule
```

Can interchange the order of the statements without affecting the output

Testbench

- The testbench is also a module
- Data type called "reg". Can assign values to variables that are declared as reg.
- Data format



- Declared as
  reg[3:0] a;
  a = 4'b1010;

```
//Half adder test bench

module test_ha;            //test bench module
wire s, c;                  //outputs from half adder
reg a, b;                  //inputs to the half adder module

ha DUT(a, b, s, c);          //instantiate the device under test (half adder)

 initial
 begin
 $monitor("At t=%t, a=%b, b=%b, s = %b, c=%b", $time, a, b, s, c);
 a = 1;  b = 1;         // t = 0
#10 a = 1;  b = 0;     //change values of a and b after 10 units of time
#10 a = 0;  b = 1;     // t = 20
#10 a = 0;  a = 0;

#10 $finish;
end
endmodule
```

Experiment 3: Multi-Bit data and Behavioural Test Bench

Objective: Build a multi-bit adder and verify the functionality across all input combinations.

- Build a full adder using
  a. Gate-level modelling
  b. Data-flow modelling

  You should have two different files descibing the circuit using various approaches and one file containing the testbench. Do not use half adders to build the full adder.
- Use the one-bit full adder to build an EIGHT bit adder and do a logic simulation to verify its functionality for all possible input combinations.

Experiment 3: Use of Multi-bit Multiplexers

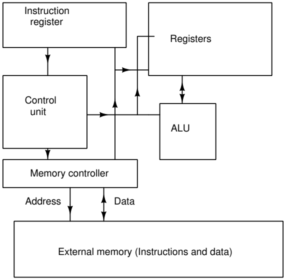
Objective: Build a small ALU that performs 4 operations based on a 2 bit instruction $I[1:0]$. You may use behavioural models for all the operations here. The operations are:

$I[1]$	$I[0]$	Operation
0	0	AND
0	1	OR
1	0	XOR
1	1	ADDITION

Input: $A[7:0], B[7:0], I[1:0]$ Output: $Y[7:0], COUT$

Functional Simulation: Behavioural Modelling

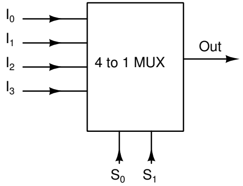
High level model specifying circuit function, without getting into actual implementation



Need to verify funcationality/execution behaviour of various instructions, before going to gate level or even data flow level modelling. Easier to write a C programe to verify functionality, using loops, case statements etc.

Behavioural modelling also useful for components.
Example: Multiplexer

C program



```
if (!S0 && !S1)
{
Out = I0
}
else if (!S0 && S1)
{
Out = I1
}
else if (S0 && !S1)
{
Out = I2
}
else
{
Out = I3
}
```

$$Out = \bar{S}_1 \bar{S}_0 I_0 + \bar{S}_1 S_0 I_1 + S_1 \bar{S}_0 I_2 + S_1 S_0 I_3$$

Data flow modelling using Verilog

```
module mux4( I_0, I_1, I_2, I_3 , s , out );

input I_0, I_1, I_2, I_3 ;
input[1:0] s;
output out
wire t0, t1;

 assign t0 = (~s[1] & I_0) | (s[1] & I_2);
 assign t1 = (~s[1] & I_1) | (s[1] & I_3);
 assign out = (~s[0] & t0) | (s[0] & t1);

endmodule
```

```
module mux4( I_0, I_1, I_2, I_3 , s , out );

input I_0, I_1, I_2, I_3 ;
input[1:0] s;
output out;

 assign out = (s == 0)? I_0:
              (s == 1)? I_1:
              (s == 2)? I_2:
              (s == 3)? I_3: 1'bx;

endmodule
```

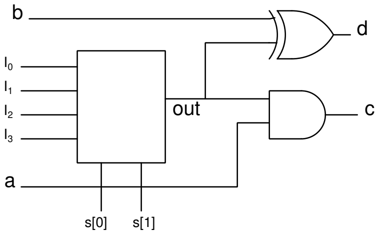
Behavioural Modelling

```
module mux4( $I_0, I_1, I_2, I_3$ , s , out );

input $I_0, I_1, I_2, I_3$ ;
input[1:0] s;
output out;
reg out;

 always@($I_0$ or $I_1$ or $I_2$ or $I_3$ or $s$)
begin
  if (s == 0)
    out = $I_0$;
  else if (s == 1)
    out = $I_1$;
  else if (s == 2)
    out = $I_2$;
  else if (s == 3)
    out = $I_3$;
end

endmodule
```

- Code is similar to C
- The arguments to the always block constitute the sensitivity list. Any change in these inputs will cause the always block to execute
- The statements within the always block are executed sequentially
- In order to assign values to it, "out" has to be declared as "reg" (similar to what we did in test benches).

Can mix models

```
module circuit( I0, I1, I2, I3, s, a, b, c, d);

input I0, I1, I2, I3 ;
input a, b;
input[1:0] s;
output c, d;
reg out;

 always@(I0 or I1 or I2 or I3 or s)
begin
case( s )
  0: out = I0;
  1: out = I1;
  2: out = I3;
  3: out = I3;
end

assign c = out & a;

xor x1(d, out, b);

endmodule
```



BACK UP SLIDES

Experiment 3: Behavioural modelling using Verilog

Objective:Model circuits using behavioural models.

- Try out the behavioural and data flow model of a 4-to-1 MUX.
- Model a 1-to-4 demultiplexer using a behavioural model.
- Repeat the experiment using a data-flow model model.
- A and B are 4-bit inputs and the output of the circuit is $MAX(A, B)$. You can use a mix of data flow and behavioural models if you wish.

In all three cases, write a test bench to test the circuit.

2s complement

- The decimal value of a n-bit binary number is $\sum_{i=0}^{n-1} a_i 2^i$.
- If a negative number is represented in the twos complement form, its decimal value is $-(2^n - \sum_{i=0}^{n-1} a_i 2^i)$.
- In practice it is obtained by first subtracting the number from $2^n - 1$ (each bit is complemented) and then adding 1 to the number.
- For a positive number $a_{n-1} = 0$ and for a negative number $a_{n-1} = 1$.
- Positive number the decimal value is $\sum_{i=0}^{n-2} a_i 2^i$. Negative number: $-(2^n - 2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i) = -2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$

2s complement

- In general, the value is $-a_{n-1}2^{n-1} + \sum_{i=0}^{n-2} a_i 2^i$
- if A is an n bit number $a_{n-1}a_{n-2}\cdots a_o$, we need $A - A = 0$. If the sum is also n bits, throw away the carry out from the MSB
- If the sum is $n+1$ bits? Do a sign extension. Verify that the decimal value of the number does not change with sign extension.
- How do you deal with fractions?