

Verilog for Testbenches

- ▶ A little Verilog...
- ▶ Big picture: Two main Hardware Description Languages (HDL) out there
 - ▶ VHDL
 - ▶ Designed by committee on request of the Department of Defense
 - ▶ Based on Ada
 - ▶ Verilog
 - ▶ Designed by a company for their own use
 - ▶ Based on C
- ▶ Both now have IEEE standards
- ▶ Both are in wide use

Overall Module Structure

```
module name (args...);  
  begin  
    parameter ...; // define parameters  
    input ...; // define input ports  
    output ...; // define output ports  
    wire ... ; // internal wires  
    reg ...; // internal regs, possibly output  
    // the parts of the module body are  
    // executed concurrently  
    <module/primitive instantiations>  
    <continuous assignments>  
    <procedural blocks (always/initial)>  
  endmodule
```

Assignments

- ▶ Continuous assignments to **wire** vars
 - ▶ `assign variable = exp;`
 - ▶ Results in combinational logic
- ▶ Procedural assignment to **reg** vars
 - ▶ Always inside procedural blocks
 - ▶ blocking
 - ▶ `variable = exp;`
 - ▶ non-blocking
 - ▶ `variable <= exp;`
 - ▶ Can result in combinational or sequential logic

Block Structures

- ▶ Two types:
 - ▶ `always // repeats until simulation is done`
`begin`
...
`end`
 - ▶ `initial // executed once at beginning of sim`
`begin`
...
`end`

Data Types

- ▶ Possible Values:
 - ▶ 0: logic 0, false
 - ▶ 1: logic 1, true
 - ▶ X: unknown logic value
 - ▶ Z: High impedance state
- ▶ Registers and Nets are the main data types
- ▶ Integer, time, and real are used in behavioral modeling, and in simulation

Registers

- ▶ Abstract model of a data storage element
- ▶ A reg holds its value from one assignment to the next
 - ▶ The value “sticks”
- ▶ Register type declarations
 - ▶ `reg a; // a scalar register`
 - ▶ `reg [3:0] b; // a 4-bit vector register`

Nets

- ▶ Nets (wires) model physical connections
- ▶ They don't hold their value
 - ▶ They must be driven by a "driver" (i.e. a gate output or a continuous assignment)
 - ▶ Their value is Z if not driven
- ▶ Wire declarations
 - ▶ `wire d;` \ a scalar wire
 - ▶ `wire [3:0] e;` \ a 4-bit vector wire
- ▶ There are lots of types of regs and wires, but these are the basics...

Memories

- ▶ Verilog memory models are arrays of regs
- ▶ Each element in the memory is addressed by a single array index
- ▶ Memory declarations:
 - ▶ \ a 256 word 8-bit memory
`reg [7:0] imem[0:255];`
 - ▶ \ a 1k word memory with 32-bit words
`reg [31:0] dmem[0:1023];`

Accessing Memories

reg [7:0] imem[0:255]; 256x8 memory

reg [7:0] foo; // 8-bit reg

Reg[2:0] bar; // 3-bit reg

foo = imem[15]; // get word 15 from mem

bar = foo[6:4]; // extract bits from foo

Other types

▶ Integers:

▶ integer i, j; \\ declare two scalar ints

▶ integer k[7:0]; \\ an array of 8 ints

▶ \$time - returns simulation time

▶ Useful inside \$display and \$monitor commands...

Number Representations

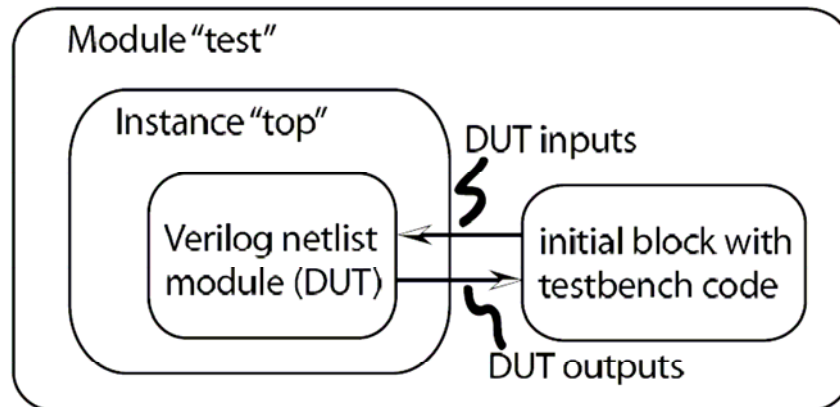
- ▶ Constant numbers can be decimal, hex, octal, or binary
- ▶ Two forms are available:
 - ▶ Simple decimal numbers: 45, 123, 49039...
 - ▶ `<size>'<base><number>`
 - ▶ base is d, h, o, or b
 - ▶ `4'b1001` // a 4-bit binary number
 - ▶ `8'h2fe4` // an 8-bit hex number

Relational Operators

- ▶ `A<B`, `A>B`, `A<=B`, `A>=B`, `A==B`, `A!=B`
 - ▶ The result is 0 if the relation is false, 1 if the relation is true, X if either of the operands has any X's in the number
- ▶ `A===B`, `A!==B`
 - ▶ These require an exact match of numbers, X's and Z's included
- ▶ `!`, `&&`, `||`
 - ▶ Logical not, and, or of expressions
- ▶ `{a, b[3:0]}` - example of concatenation

Testbench Template

- ▶ Testbench template generated by Cadence



Testbench Template

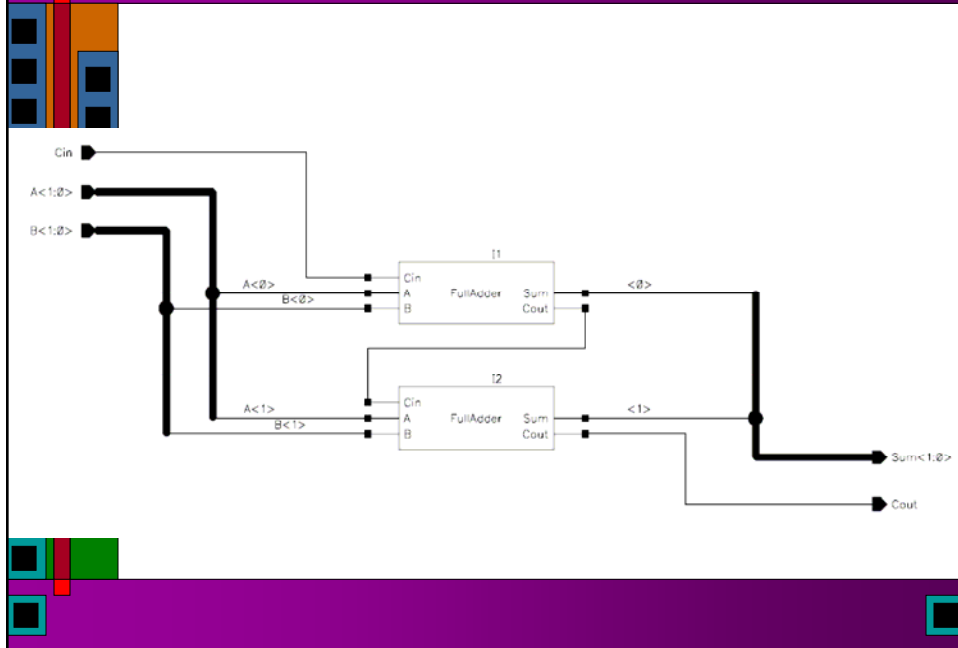
```
`timescale 1ns / 100ps
module test;

  wire Cout;
  reg Cin;
  wire [1:0] Sum;
  reg [1:0] A;
  reg [1:0] B;

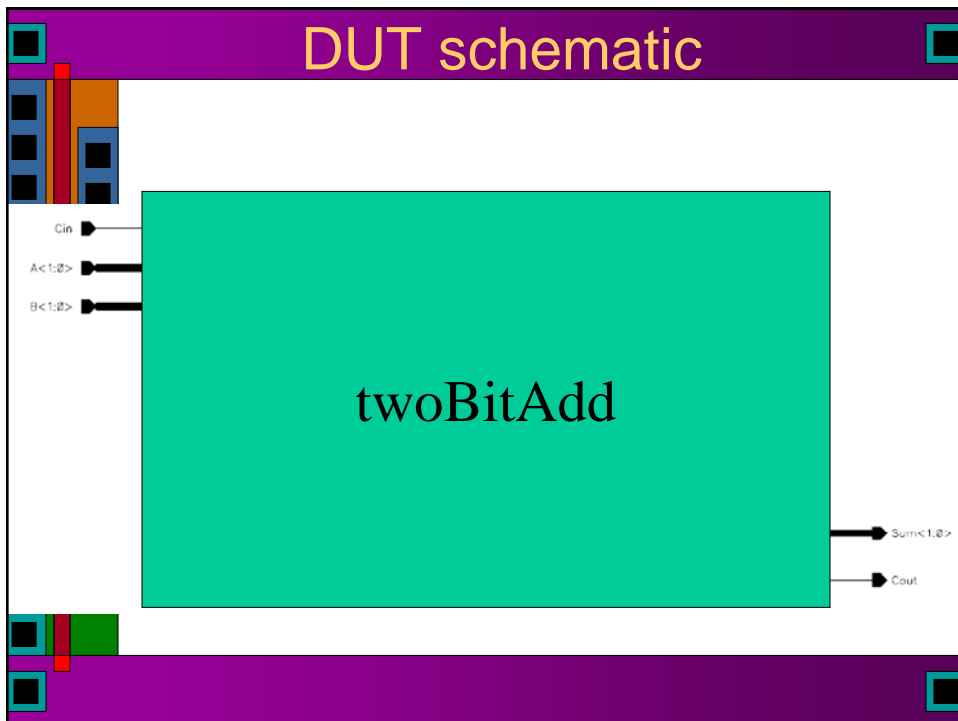
  twoBitAdd top(Cout, Sum, A, B, Cin);

  `include "testfixture.verilog"
endmodule
```

DUT schematic



DUT schematic



testfixture.verilog

- ▶ Again, template generated by Cadence

```
// Verilog stimulus file.  
// Please do not create a module in this file.  
  
// Default verilog stimulus.  
  
initial  
begin  
  
    A[1:0] = 2'b00;  
  
    B[1:0] = 2'b00;  
  
    Cin = 1'b0;  
end
```

Testbench code

- ▶ So, all your test code will be inside an initial block!
 - ▶ Or, you can create new procedural blocks that will be executed concurrently
 - ▶ Remember the structure of the module
 - ▶ If you want new temp variables you need to define those outside the procedural blocks
 - ▶ Remember that DUT inputs and outputs have been defined in the template
 - ▶ DUT inputs are reg type
 - ▶ DUT outputs are wire type

Basic Testbench

```
initial
begin
  a[1:0] = 2'b00;
  b[1:0] = 2'b00;
  cin = 1'b0;
  $display("Starting...");
  #20
  $display("A = %b, B = %b, c = %b, Sum = %b, Cout = %b", a, b, cin, sum, cout);
  if (sum != 00) $display("ERROR: Sum should be 00, is %b", sum);
  if (cout != 0) $display("ERROR: cout should be 0, is %b", cout);
  a = 2'b01;
  #20
  $display("A = %b, B = %b, c = %b, Sum = %b, Cout = %b", a, b, cin, sum, cout);
  if (sum != 00) $display("ERROR: Sum should be 01, is %b", sum);
  if (cout != 0) $display("ERROR: cout should be 0, is %b", cout);
  b = 2'b01;
  #20
  $display("A = %b, B = %b, c = %b, Sum = %b, Cout = %b", a, b, cin, sum, cout);
  if (sum != 10) $display("ERROR: Sum should be 10, is %b", sum);
  if (cout != 0) $display("ERROR: cout should be 0, is %b", cout);
  $display("...Done");
  $finish;
end
```

\$display, \$monitor

- ▶ `$display(format-string, args);`
 - ▶ like a printf
 - ▶ `$fdisplay` goes to a file...
 - ▶ `$fopen` and `$fclose` deal with files
- ▶ `$monitor(format-string, args);`
 - ▶ Wakes up and prints whenever args change
 - ▶ Might want to include `$time` so you know when it happened...
 - ▶ `$fmonitor` is also available...

Conditional, For

- ▶ If (<expr>) <statement> else <statement>
 - ▶ else is optional and binds with closest previous if that lacks an else
 - ▶ if (index > 0)
 - if (rega > regb)
result = rega;
 - else
result = regb;
- ▶ For is like C
 - ▶ for (initial; condition; step)
 - ▶ for (k=0; k<10; k=k+1)
statement;

No k++ syntax
in Verilog...

for

```
parameter MAX_STATES 32;
integer state[0:MAX_STATES-1];
integer i;

initial
begin
for(i=0; i<32 ; i=i+2)
state[i] = 0;
for(i=1; i<32; i=i+2)
state[i] = 1;
end
```

while

▶ A while loop executes until its condition is false



```
count = 0;
while (count < 128)
  begin
    $display("count = %d", count);
    count = count + 1;
  end
```

repeat

▶ repeat for a fixed number of iterations

```
parameter cycles = 128;
integer count;
initial
  begin
    count = 0;
    repeat(cycles)
      begin
        $display("count = %d", count);
        count = count+1;
      end
  end
end
```

Nifty Testbench

```
reg [1:0] ainarray [0:4]; // define memory arrays to hold input and result
reg [1:0] binarray [0:4];
reg [2:0] resultsarray [0:4];
integer i;
initial begin
  $readmemb("ain.txt", ainarray); // read values into arrays from files
  $readmemb("bin.txt", binarray);
  $readmemb("results.txt", resultsarray);
  a[1:0] = 2'b00; // initialize inputs
  b[1:0] = 2'b00;
  cin = 1'b0;
  $display("Starting...");
  #10 $display("A = %b, B = %b, c = %b, Sum = %b, Cout = %b", a, b, cin, sum, cout);
  for (i=0; i<=4; i=i+1) // loop through all values in the memories
  begin
    a = ainarray[i]; // set the inputs from the memory arrays
    b = binarray[i];
    #10 $display("A = %b, B = %b, c = %b, Sum = %b, Cout = %b", a, b, cin, sum, cout);
    if ((cout,sum) != resultsarray[i])
      $display("Error: Sum should be %b, is %b instead", resultsarray[i],sum); // check results array
  end
  $display("...Done");
  $finish;
end
```

Another Nifty Testbench

```
integer i,j,k;
initial
begin
  A[1:0] = 2'b00;
  B[1:0] = 2'b00;
  Cin = 1'b0;
  $display("Starting simulation...");
  for(i=0;i<=3;i=i+1)
  begin for(j=0;j<=3;j=j+1)
    begin for(k=0;k<=1;k=k+1)
      begin
        #20 $display("A=%b B=%b Cin=%b, Cout-Sum=%b%b", A, B, Cin, Cout, S);
        if ({Cout,S} != A + B + Cin)
          $display("ERROR: CoutSum should equal %b, is %b", (A + B + Cin), {Cin,S});
          Cin=~Cin; // invert Cin
        end
        B[1:0] = B[1:0] + 2'b01; // add 1 to the B input
      end
      A = A+1; // shorthand notation for adding
    end
  end
  $display("Simulation finished... ");
end
```

Another Example

```
initial // executed only once
begin
    a = 2'b01; // initialize a and b
    b = 2'b00;
end
always // execute repeatedly
begin // until simulation completes
    #50 a = ~a; // reg a inverts every 50 units
end
always // execute repeatedly
begin // until simulation completes
    #100 b = ~b // reg b inverts every 100 units
end
```

What's wrong with this code?

Another Example

```
initial // executed only once
begin
    a = 2'b01; // initialize a and b
    b = 2'b00;
    #200 $finish; // make sure the simulation
                // finishes!
end
always // execute repeatedly
begin // until simulation completes
    #50 a = ~a; // reg a inverts every 50 units
end
always // execute repeatedly
begin // until simulation completes
    #100 b = ~b // reg b inverts every 100 units
end
```