Verilog Overview

CS/EE 3710
Fall 2010

Hardware Description Languages

- HDL
  - Designed to be an alternative to schematics for describing hardware systems
- Two main survivors
  - VHDL
    - Commissioned by DOD
    - Based on ADA syntax
  - Verilog
    - Designed by a company for their own use
    - Based on C syntax

Verilog Origins

- Developed as a proprietary HDL by Gateway Design Automation in 1984
- Acquired by Cadence in 1989
- Made an open standard in 1990
- Made an IEEE standard in 1995
- IEEE standard Revised in 2001

Verilog

- You can think of it as a programming language
  - BUT, that can get you into trouble!
- Better to think of it as a way to describe hardware
  - Begin the design process on paper
  - Plan the hardware you want
  - Use Verilog to describe that hardware

Quick Review

Module name (args...);
begin
  parameter ...; // define parameters
  input ...; // define inputs
  output ...; // define outputs
  wire ...; // internal wires
  reg ...; // internal regs, possibly output
  // the parts of the module body are
  // executed concurrently
  <continuous assignments>
  <always blocks>
endmodule

Quick Review (2001 syntax)

Module name (parameters, inputs, outputs);
begin
  wire ...; // internal wires
  reg ...; // internal regs
  // the parts of the module body are
  // executed concurrently
  <continuous assignments>
  <always blocks>
endmodule
### Quick Review
- Continuous assignments to wire vars
  - `assign variable = exp;`
  - Results in combinational logic
- Procedural assignment to reg vars
  - Always inside procedural blocks (always blocks in particular for synthesis)
  - blocking
    - `variable = exp;`
  - non-blocking
    - `variable <= exp;`
  - Can result in combinational or sequential logic

### Verilog Description Styles
- Verilog supports a variety of description styles
  - Structural
    - explicit structure of the circuit
    - e.g., each logic gate instantiated and connected to others
    - Hierarchical instantiations of other modules
  - Behavioral
    - program describes input/output behavior of circuit
    - many structural implementations could have same behavior
    - e.g., different implementation of one Boolean function

### Synthesis: Data Types
- Possible Values (wire and reg):
  - 0: logic 0, false
  - 1: logic 1, true
  - Z: High impedance
- Digital Hardware
  - The domain of Verilog
  - Either logic (gates)
  - Or storage (registers & latches)
- Verilog has two relevant data types
  - wire
  - reg

### Number Syntax
- Numbers with no qualifiers are considered decimal
  - 1 23 456 etc.
- Can also qualify with number of digits and number base
  - base can be b, B, h, H, d, D, o, O

\[
\begin{align*}
4'b1011 & \quad // 4\text{-}bit binary of value 1011 \\
234 & \quad // 3\text{-}digit decimal of value 234 \\
2'h5a & \quad // 2\text{-}digit (8\text{-}bit) hexadecimal of value 5A \\
3'o671 & \quad // 3\text{-}digit (9\text{-}bit) octal of value 671 \\
4b'1x0z & \quad // 4\text{-}bit binary. 2\text{nd MSB is unknown. LSB is Hi-Z.} \\
3.14 & \quad // Floating point \\
1.28e5 & \quad // Scientific notation
\end{align*}
\]

### Parameters
- Used to define constants
  - `parameter size = 16, foo = 8;`
  - `wire [size-1:0] bus; \quad // defines a 15:0 bus`
The assign statement creates combinational logic.

Assign LHS = expression;
LHS can only be wire type
expression can contain either wire or reg type mixed with operators
wire a, c; reg b; output out;
assign a = b & c;
assign out = ~(a & b); \ output as wire
wire [15:0] sum, a, b;
wire cin, cout;
assign {cout,sum} = a + b + cin;
wire[4:0]sum1;
assign (cout,sum) = a + b + cin;
assign (cout,sum) = a + b + (4'b0,cin);
assign sum1 = a + b;
assign sum = (a + b) >> 1; \ what is wrong?
The assign statement is sufficient to create all combinational logic.

What about this:

```verilog
assign a = ~(b & c);
assign c = ~(d & a);
```

---

// Behavioral model of NAND gate
module NAND (out, in1, in2);
    output out;
    input in1, in2;
    assign out = ~(in1 & in2);
endmodule

---

// Structural Module for NAND gate
module NAND (input out, input in1, input in2);
    wire w1;
    // local wire
    // call existing modules by name
    // module-name ID (signal-list);
    AND2X1 u1(w1, in1, in2);
    INVX1 u2(out, w1);
endmodule
Simple Structural Module
// Structural Module for NAND gate
module NAND (output out; input in1, in2);
wire w1;
// call existing modules by name
// module-name ID (signal-list);
// can connect ports by name...
AND2X1 u1(.Q(w1), .A(in1), .B(in2));
INVX1 u2(.A(w1), .Q(out));
endmodule

Procedural Assignment
- Assigns values to register types
- They involve data storage
  - The register holds the value until the next procedural assignment to that variable
  - The occur only within procedural blocks
    - initial and always
    - initial is NOT supported for synthesis!
  - They are triggered when the flow of execution reaches them

Always Blocks
- When is an always block executed?
  - always
    - Starts at time 0
    - always @(a or b or c)
      - Whenever there is a change on a, b, or c
      - Used to describe combinational logic
    - always @(posedge foo)
      - Whenever foo goes from low to high
      - Used to describe sequential logic
    - always @(negedge bar)
      - Whenever bar goes from high to low

Synthesis: Always Statement
- The always statement creates...
  - always @sensitivity
    - LHS = expression;
    - @sensitivity controls when
      - LHS can only be reg type
      - expression can contain either wire or reg type mixed with operators
      - … Logic
        - reg c, b; wire a;
        - always @(a, b)
        - c = ~a & b;
      - … Storage
        - reg Q; wire clk;
        - always @(posedge clk)
        - Q <= D;

Procedural Control Statements
- Conditional Statement
  - if ( <expression> ) <statement>
  - if ( <expression> ) <statement> else <statement>
    - “else” is always associated with the closest previous if that lacks an else.
    - You can use begin-end blocks to make it more clear
  - if (index >0)
    - if (rega > regb)
      - result = rega;
    - else result = regb;

Multi-Way Decisions
- Standard if-else-if syntax
  - If ( <expression> )
    - <statement>
  - else if ( <expression> )
    - <statement>
  - else if ( <expression> )
    - <statement>
  - else <statement>
Procedural NAND gate

// Procedural model of NAND gate
module NAND (output out;
  reg out;
  input in1, in2);
  // always executes when in1 or in2
  // change value
  always @(in1 or in2)
      begin
        out = ~(in1 & in2);
      end
endmodule

Is out combinational?

Synthesis: NAND gate

input in1, in2;
reg n1, n2; // is this a flip-flop?
wire n3, n4;
always @(in1 or in2) n1 = ~(in1 & in2);
always @(*) n2 = ~(in1 & in2);
assign n3 = ~(in1 & in2);
nand u1(n4, in1, in2);

Notice always block for combinational logic
Full sensitivity list, but @(*) works
Can then use the always goodies
Is this a good coding style?

Procedural Assignments

Assigns values to reg types
- Only useable inside a procedural block, can synthesize to a register
- But, under the right conditions, can also result in combinational circuits

Blocking procedural assignment
- LHS = timing-control exp
- a = #10 1;
- Must be executed before any assignments that follow (timing control is simulated, not synthesized)
- Assignments proceed in order even if no timing is given

Non-Blocking procedural assignment
- LHS <= timing-control exp
- b <= 2;
- Evaluated simultaneously when block starts
- Assignment occurs at the end of the (optional) time-control

Procedural Synthesis

Synthesis ignores all that timing stuff
So, what does it mean to have blocking vs. non-blocking assignment for synthesis?

begin
  A=B;
  B=A;
end
begin
  A<=B;
  B<=A;
end

begin
  A=Y;
  B=A;
end
begin
  A<=Y;
  B<=A;
end

begin
  B=A;
  A=Y;
end
begin
  B<=A;
  A<=Y;
end

Synthesized Circuits
Synthesized Circuits

Always Statement

Assignments and Synthesis

Note that different circuit structures result from different types of procedural assignments

Therefore you can’t mix assignment types in the same always block

Non-blocking is often a better model for hardware

Real hardware is often concurrent...

Comparator Example

Using continuous assignment
Concurrent execution of assignments

Using procedural assignment
Non-blocking assignment implies concurrent

Module comp (a, b, Cgt, Clt, Cne);
parameter n = 4;
input [n-1:0] a, b;
output Cgt, Clt, Cne;
assign Cgt = (a > b);
assign Clt = (a < b);
assign Cne = (a != b);
endmodule

Modeling a Flip Flop

Use an always block to wait for clock edge

This is a simple D Flip-Flop

reg Q;
always @(posedge clk) Q <= D;

@(posedge clk) is the sensitivity list
The Q <= D; is the block part
The block part is always "entered" whenever the sensitivity list becomes true (positive edge of clk)
The LHS of the <= must be of data type reg
The RHS of the <= may use reg or wire

Synthesis: Always Statement

Comparator Example

Module comp (a, b, Cgt, Clt, Cne);
parameter n = 4;
input [n-1:0] a, b;
output Cgt, Clt, Cne;
reg Cgt, Clt, Cne;
always @(a or b)
begin
    Cgt <= (a > a);
    Clt <= (a < b);
    Cne <= (a != b);
end
endmodule
Synthesis: Always Statement

- This is an asynchronous clear D Flip-Flop
  reg Q;
  always @(posedge clk, posedge rst)
    if (rst) Q <= 'b0; else Q <= D;
- Notice , instead of or
- Verilog 2001...
- Positive reset

Constants

- parameter used to define constants
  - parameter size = 16, foo = 8;
  - wire [size-1:0] bus; \ defines a 15:0 bus
  - externally modifiable
    - scope is local to module
  - localparam not externally modifiable
    - localparam width = size * foo;
  - `define macro definition
    - `define value 7'd53
    - assign a = (sel == `value) & b;
  - scope is from here on out

Example: Counter

module counter #(
    parameter width=8)
  (input clk, clr, load,
   input [width-1:0] in;
   output [width-1:0] count);
  reg [width-1:0] tmp;
  always @(
    posedge clk or negedge clr)
    begin
      if (!clr)
        tmp = 0;
      else if (load)
        tmp = in;
      else
        tmp = tmp + 1;
    end
  assign count = tmp;
endmodule

Synthesis: Modules

// Verilog 1995 syntax
module adder (e,f,g);
  parameter SIZE=2;
  input [SIZE-1:0] e, f;
  output [SIZE-1:0] g;
  assign g = e + f;
endmodule

// Verilog 2001 syntax
module subber #(
    parameter SIZE = 3)
  (input [SIZE-1:0] c,d,
   output [SIZE-1:0]difference);
  assign difference = c - d;
endmodule

Synthesis: Modules

module the_top (clk, rst, a, b, sel, result);
  input clk, rst;
  input [3:0] a,b;
  input [2:0] sel;
  output reg [3:0] result;
  wire[3:0] sum, dif, alu;
  adder u0(a,b,sum);
  subber u1(.c(a), .d(b), .difference(dif));
  assign alu = (4'(b000)) & sum |
  (4'(b001)) & dif;
  always @(
    posedge clk or posedge rst)
    if(rst) result <= 'h0;
    else result <= alu;
endmodule
Case Statements

- Multi-way decision on a single expression

```plaintext
case ( <expression> )
  <expression>: <statement>
  <expression>, <expression>: <statement>
  <expression>: <statement>
default: <statement>
endcase
```

Case Example

```plaintext
reg [1:0] sel;
reg [15:0] in0, in1, in2, in3, out;
case (sel)
  2'b00: out = in0;
  2'b01: out = in1;
  2'b10: out = in2;
  2'b11: out = in3;
endcase
```

Another Case Example

```plaintext
// simple counter next-state logic
// one-hot state encoding...
parameter [2:0] s0=3’h1, s1=3’h2, s2=3’h4;
reg[2:0] state, next_state;
always @(input or state)
begin
  case (state)
    s0: if (input) next_state = s1;
        else next_state = s0;
    s1: next_state = s2;
    s2: next_state = s0;
  default:
    next_state = s0;
  endcase
end
```

Latch Inference

- Incompletely specified if and case statements cause the synthesizer to infer latches
- Fix by adding an else
- In a case, fix by including default:

```
begin
  if (cond) data_out <= data_in;
end
```

Full vs. Parallel

- Case statements check each case in sequence
- A case statement is full if all possible outcomes are accounted for
- A case statement is parallel if the stated alternatives are mutually exclusive
- These distinctions make a difference in how cases are translated to circuits...
- Similar to the if statements previously described

Full vs. Parallel example

```plaintext
// full and parallel = combinational logic
module full-par (slct, a, b, c, d, out);
input [1:0] slct;
input a, b, c, d;
output out;
reg out; // optimized away in this example
always @(slct or a or b or c or d)
  case (slct)
    2'b11 : out <= a;
    2'b10 : out <= b;
    2'b01 : out <= c;
    default : out <= d; // really 2'b10
  endcase
endmodule
```
Note that full-par results in combinational logic.

```
module notfull-par (slct, a, b, c, d, out);
  input [1:0] slct;
  input a, b, c, d;
  output out;
  reg out; // NOT optimized away in this example
  always @(slct or a or b or c)
  case (slct)
    2'b11 : out <= a;
    2'b10 : out <= b;
    2'b01 : out <= c;
  endcase
endmodule
```

Because it's not full, a latch is inferred...

```
module full-notpar (slct, a, b, c, out);
  ...
  always @(slct or a or b or c)
  casez (slct)
    2'b1? : out <= a;
    2'b?1 : out <= b;
    default : out <= c;
  endcase
endmodule
```

It's full, so it's combinational, but it's not parallel so it's a priority circuit instead of a "check all in parallel" circuit.

```
module notfull-notpar example
  ...
  always @(slct or a or b or c)
  casez (slct)
    2'b1? : out <= a;
    2'b?1 : out <= b;
  endcase
endmodule
```

Because it's not parallel, it's a priority encoding circuit.

```
module full-notpar example
  ...
  always @(slct or a or b or c)
  casez (slct)
    2'b1? : out <= a;
    2'b?1 : out <= b;
    default : out <= c;
  endcase
endmodule
```
Synthesized Circuit

- Not full and not parallel, infer a latch

![Diagram of a synthesized circuit with a D flip-flop (DFF) and selectors (slct0, slct1).]

FSM Description

- One simple way: break it up like a schematic
- A combinational block for next_state generation
- A combinational block for output generation
- A sequential block to store the current state

Mealy only

![Diagram of a Mealy FSM with inputs, next_state, state, and outputs.]

Modeling State Machines

// General view
module FSM (clk, in, out);
  input clk, in;
  output out;
  reg out;
  // state variables
  reg [1:0] state;
  // next state variable
  reg [1:0] next_state;
  always @posedge(clk) // state register
  state = next_state;
  always @(state or in); // next-state logic
  // compute next state and output logic
  // make sure every local variable has an
  // assignment in this block
endmodule

Next state Logic

State

outputs

Verilog Version

module moore (clk, clr, insig, outsig);
  input clk, clr, insig;
  output outsig;
  // define state encodings as parameters
  parameter [1:0] s0 = 2'b00, s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;
  // define reg vars for state register
  reg [1:0] state, next_state;
  //define state register (with synchronous active-high clear)
  always @(posedge clk)
  begin
    if (clr) state = s0;
    else state = next_state;
  end
  // define combinational logic for next_state
  always @(insig or state)
  begin
    case (state)
      s0: if (insig) next_state = s1;
      s1: if (insig) next_state = s2;
      s2: if (insig) next_state = s3;
      s3: if (insig) next_state = s1;
    endcase
  end
  // assign outsig as continuous assign
  assign outsig = (state == s1) || (state == s3);
endmodule

Verilog Version

module moore (clk, clr, insig, outsig);
  input clk, clr, insig;
  output outsig;
  // define state encodings as parameters
  parameter [1:0] s0 = 2'b00, s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;
  // define reg vars for state register
  reg [1:0] state, next_state;
  //define state register (with synchronous active-high clear)
  always @(posedge clk)
  begin
    if (clr) state = s0;
    else state = next_state;
  end
// define combinational logic for next_state
always @(insig or state)
begin
    case (state)
        s0: if (insig) next_state = s1;
            else next_state = s0;
        s1: if (insig) next_state = s2;
            else next_state = s1;
        s2: if (insig) next_state = s3;
            else next_state = s2;
        s3: if (insig) next_state = s1;
            else next_state = s0;
    endcase
end

// now set the outsig. This could also be done in an always
// block... but in that case, outsig would have to be
// defined as a reg.
assign outsig = ((state == s1) || (state == s3));
endmodule

Unsupported for Synthesis
- Delay (ISE will ignore #')
- initial blocks (use explicit resets)
- initial values for defined variables (use explicit resets)
- repeat
- wait
- fork
- event
- deassign
- force
- release

More Unsupported Stuff
- You cannot assign the same reg variable in more than one procedural block

Sync vs. Async Register Reset
- synchronous reset (active-high reset)
  always @(posedge clk)
  if (reset) state = s0;
  else state = s1;

- async reset (active-low reset)
  always @(posedge clk or negedge reset)
  if (reset == 0) state = s0;
  else state = s1;

Combinational Always Blocks
- Be careful…
  always @(sel)  always @(sel or in1 or in2)
  if (sel == 1)  if (sel == 1)
      out = in1;  out = in1;
  else out = in2;  else out = in2;

- Which one is a good mux?
Finite State Machine

Four in a Row

Textbook FSM

```verilog
module FSM2(C, EN, x1, x2, x3, ERROR);
input [2:0] C;
input EN, x1, x2, x3;
output ERROR;

// Delimit state boundaries by parameter, // always use <= for FF
parameter [2:0] z1 = 0B000, z2 = 0B001, z3 = 0B100, // Always use <= for FF
z4 = 0B110;
parameter [2:0] z5 = 0B111;
parameter [2:0] z6 = 0B111;

// Delimit state changes by parameters, // always use <= for FF
parameter [2:0] z0 = 0B000, z1 = 0B001, z2 = 0B100, // always use <= for FF
z3 = 0B101, z4 = 0B110;
parameter [2:0] z5 = 0B111;
parameter [2:0] z6 = 0B111;

// Delimit state changes by parameters, // always use <= for FF
parameter [2:0] s0 = 0B000, s1 = 0B001, s2 = 0B100, // always use <= for FF
s3 = 0B101, s4 = 0B110;
parameter [2:0] s5 = 0B111;
parameter [2:0] s6 = 0B111;

// Delimit state changes by parameters, // always use <= for FF
parameter [2:0] x0 = 0B000, x1 = 0B001, x2 = 0B100, // always use <= for FF
x3 = 0B101, x4 = 0B110;
parameter [2:0] x5 = 0B111;
parameter [2:0] x6 = 0B111;

always_ff @(C, EN)
begin
  if (EN) // always use <= for FF
    ERROR = 0;
    case (C)
      z0: begin
        always_ff @(x0)
          begin
            case (x0)
              x0: begin
                ERROR = 0;
                always_ff @(x0)
                  begin
                    case (x0)
                      x0: begin
                        ERROR = 0;
                        always_ff @(x0)
                          begin
                            case (x0)
                              x0: begin
                                ERROR = 0;
                                always_ff @(x0)
                                  begin
                                    case (x0)
                                      x0: begin
                                        ERROR = 0;
                                        always_ff @(x0)
                                          begin
                                            case (x0)
                                              x0: begin
                                                ERROR = 0;
                                                always_ff @(x0)
                                                  begin
                                                    case (x0)
                                                      x0: begin
                                                        ERROR = 0;
                                                        always_ff @(x0)
                                                          begin
                                                            case (x0)
                                                              x0: begin
                                                                ERROR = 0;
                                                                always_ff @(x0)
                                                                  begin
                                                                    case (x0)
                                                                      x0: begin
                                                                        ERROR = 0;
                                                                        always_ff @(x0)
                                                                          begin
                                                                            case (x0)
                                                                              x0: begin
                                                                                ERROR = 0;
                                                                                always_ff @(x0)
                                                                                  begin
                                                                                      case (x0)
                                                                                          x0: begin
                                                                                           ERROR = 0;
                                                                                           always_ff @(x0)
                                                                                             begin
                                                                                               case (x0)
                                                                                                   x0: begin
                                                                                                    ERROR = 0;
                                                                                                    always_ff @(x0)
                                                                                                      begin
                                                                                                        case (x0)
                                                                                                            x0: begin
                                                                                                             ERROR = 0;
                                                                                                             always_ff @(x0)
                                                                                                               begin
                                                                                                                 case (x0)
                                                                                                                    x0: begin
                                                                                                                     ERROR = 0;
                                                                                                                     always_ff @(x0)
                                                                                                                       begin
                                                                                                                          case (x0)
                                                                                                                              x0: begin
                                                                                                                               ERROR = 0;
                                                                                                                               always_ff @(x0)
                                                                                                                                  begin
                                                                                                                                     case (x0)
                                                                                                                                         x0: begin
                                                                                                                                          ERROR = 0;
                                                                                                                                          always_ff @(x0)
                                                                                                                                            begin
                                                                                                                                                case (x0)
                                                                                                                                                    x0: begin
                                                                                                                                                     ERROR = 0;
                                                                                                                                                     always_ff @(x0)
                                                                                                                                                        begin
                                                                                                                                                           case (x0)
                                                                                                                                                               x0: begin
                                                                                                                                                                 ERROR = 0;
                                                                                                                                                                 always_ff @(x0)
                                                                                                                                                                    begin
                                                                                                                                                                                   case (x0)
                                                                                                                                                                                       x0: begin
                                                                                                                                                                                        ERROR = 0;
                                                                                                                                                                                        always_ff @(x0)
                                                                                                                                                                                           begin
                                                                                                                                                                                               case (x0)
                                                                                                                                                                                                  x0: begin
                                                                                                                                                                                                     ERROR = 0;
                                                                                                                                                                                                     always_ff @(x0)
                                                                                                                                                                                                        begin
                                                                                                                                                                                                                case (x0)
                                                                                                                                                                                                                    x0: begin
                                                                                                                                                                                                                     ERROR = 0;
                                                                                                                                                                                                                     always_ff @(x0)
                                                                                                                                                                                                                        begin
                                                                                                                                                                                                                                   case (x0)
                                                                                                                                                                                                                                     x0: begin
                                                                                                                                                                                                                                         ERROR = 0;
                                                                                                                                                                                                                                         always_ff @(x0)
                                                                                                                                                                                                                                            begin
                                                                                                                                                                                                                                               case (x0)
                                                                                                                                                                                                                                                 x0: begin
                                                                                                                                                                                                                                                   ERROR = 0;
                                                                                                                                                                                                                                                   always_ff @(x0)
                                                                                                                                                                                                                                                      begin
                                                                                                                                                                                                                                                                  case (x0)
                                                                                                                                                                                                                                                                                  x0: begin
                                                                                                                                                                                                                                                                                    ERROR = 0;
                                                                                                                                                                                                                                                                                    always_ff @(x0)
                                                                                                                                                                                                                                                                                        begin
                                                                                                                                                                                                                                                                                                                                                    case (x0)
                                                                                                                                                                                                                                                                                                                                                        x0: begin
                                                                                                                                                                                                                                                                                                                                                                                   ERROR = 0;
                                                                                                                                                                                                                                                                                                                                                                                  always_ff @(x0)
                                                                                                                                                                                                                                                                                                                                                                                     begin
                                                                armac
                                                                                                  end
                                                                                              end
                                                                                        end
                                            end
                                         end
                                      end
                                    end
                                  end
                                end
                              end
                            end
                         end
                       end
                    end
                  end
                end
              end
            end
          end
        end
      end
    end
endcase
end
```

NAND example

```verilog
module AND2(input A, B, output Y);
input A, B;
output Y;
always @ (A or B)
begin
  if (A) // always use <= for FF
    Y = 1;
  else
    Y = 0;
end
```

Verilog Testbenches

- Verilog code that applies inputs and checks outputs of your circuit
- Framework is generated by ISE
- You fill in the details of the test
- Your circuit is instantiated and named UUT
- Unit Under Test
- You apply inputs, and check outputs
NAND example

```verilog
module nand Example;
  // Inputs
  reg A;
  reg B;
  // Outputs
  wire Y;

  // Instantiate the Unit Under Test (UUT)
  assign Y = (A & B);

  initial begin
    // Wait 100 ns for global reset to finish
    #100
    // Add stimulus here
    A = 1'b1;
    B = 1'b0;
    // Add stimulus here
    A = 1'b0;
    B = 1'b1;
    // Add stimulus here
    A = 1'b1;
    B = 1'b1;
    // Add stimulus here
    A = 1'b0;
    B = 1'b0;
  end
endmodule
```

NAND Testbench

```verilog
initial begin
  // Initialize Inputs
  A = 1'b1;
  B = 1'b0;
  // Wait 100 ns for global reset to finish
  #100
  // Add stimulus here
  if (Y == 1'b1) begin
    $display("ERROR: Y is 1'b1 should be 0'b1", Y);
  end
  A = 1'b1;
  B = 1'b0;
  // Add stimulus here
  if (Y == 1'b1) begin
    $display("ERROR: Y is 1'b1 should be 0'b1", Y);
  end
  A = 1'b0;
  B = 1'b1;
  // Add stimulus here
  if (Y == 1'b1) begin
    $display("ERROR: Y is 1'b1 should be 0'b1", Y);
  end
  A = 1'b0;
  B = 1'b0;
  // Add stimulus here
  if (Y == 1'b1) begin
    $display("ERROR: Y is 1'b1 should be 0'b1", Y);
  end
end
```

Two-bit adder example

```verilog
initial begin
  // Initialize Inputs
  A = 0;
  B = 0;
  // Wait 100 ns for global reset to finish
  #100
  // Add stimulus here
  if (Y == 1'b1) begin
    $display("ERROR: Y is 1'b1 should be 0'b1", Y);
  end
end
```

DUT schematic

```
```

```
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 UUT inputs and outputs have been defined in the template

UUT inputs are reg type

UUT 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 != 00) $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
```

Another Nifty Testbench

```
```

Another adder example

```
```
Another adder example

// generate a clock to sequence tests
always begin
  clk = 1; $stop; clk = 0; $stop;
end

// on each clock edge, apply next test
always @posedge clk
begin
  a = testvector[vectorsum[3]];
  b = testvector[vectorsum[3] + 1];
  expected = testvector[vectorsum[3] + 2];
end

// then check for correct results
always @posedge clk
begin
  vectorsum = vectorsum + 1;
  if (y != expected) begin
    $display("Inputs were 90, 90, a, b, 890000. Expected: 890000, but actual: %d", actual);"y, errors + 1);
    $display("Expected %d but actual %d", expected, actual);
    errors = errors + 1;
  end
end

Another adder example

// halt at the end of file
always @(vectorsum)
begin
  if (vectorsum == 100) begin
    $display("Completed 100 tests with 0 errors.", vectorsum, errors);
    exit(0);
  end
end

module testvector;
  reg [3:0] vectorsum;

  always #1 begin
    vectorsum = vectorsum + 1;
  end

  reg [31:0] vectordata[100];

  initial begin
    vectordata = 890000; // Example initial data
  end

  initial begin
    vectorsum = 0;
  end

endmodule

initial begin
  testvector
end

initial begin
  vectorsum = 100;
end

digitalout�述 m 0 [3:0] vectordata;

Simulation dump:
00000000
00000001
00000002
00000003
00000004
00000005
00000006
00000007
12345678
12345679
2450abc