

## 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**

## Synthesis: Data Types

- ▶ Register declarations
  - ▶ **reg a;**    \\ a scalar register
  - ▶ **reg [3:0] b;**    \\ a 4-bit vector register
  - ▶ **output g;**    \\ an output can be a reg
  - reg g;**
  - ▶ **output reg g;**    \\ Verilog 2001 syntax
- ▶ Wire declarations
  - ▶ **wire d;**    \\ a scalar wire
  - ▶ **wire [3:0] e;**    \\ a 4-bit vector wire
  - ▶ **output f;**    \\ an output can be a wire

## 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
- ```
4'b1011 // 4-bit binary of value 1011
234     // 3-digit decimal of value 234
2'h5a   // 2-digit (8-bit) hexadecimal of value 5A
3'o671  // 3-digit (9-bit) octal of value 671
4b'1x0z // 4-bit binary. 2nd MSB is unknown. LSB is Hi-Z.
3.14    // Floating point
1.28e5  // Scientific notation
```

## Parameters

- ▶ Used to define constants
  - ▶ **parameter size = 16, foo = 8;**
  - ▶ **wire [size-1:0] bus; \\ defines a 15:0 bus**

## Synthesis: Assign Statement

- ▶ 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;`

## Synthesis: Basic Operators

- ▶ Bit-Wise Logical
  - ▶ `~` (not), `&` (and), `|` (or), `^` (xor), `^~` or `~^` (xnor)
- ▶ Simple Arithmetic Operators
  - ▶ Binary: `+`, `-`
  - ▶ Unary: `-`
  - ▶ Negative numbers stored as 2's complement
- ▶ Relational Operators
  - ▶ `<`, `>`, `<=`, `>=`, `==`, `!=`
- ▶ Logical Operators
  - ▶ `!` (not), `&&` (and), `||` (or)  
`assign a = (b > 'b0110) && (c <= 4'd5);`  
`assign a = (b > 'b0110) && !(c > 4'd5);`

## Synthesis: More Operators

### ▶ Concatenation

▶ `{a,b} {4{a==b}} {a,b,4'b1001},{4{a==b}}`

### ▶ Shift (logical shift)

▶ `<<` left shift

▶ `>>` right shift

`assign a = b >> 2; // shift right 2, division by 4`

`assign a = b << 1; // shift left 1, multiply by 2`

### ▶ Arithmetic

`assign a = b * c; // multiply b times c`

`assign a = b * 'd2; // multiply b times constant (=2)`

`assign a = b / 'b10; // divide by 2 (constant only)`

`assign a = b % 'h3; // b modulo 3 (constant only)`

## Synthesis: Operand Length

### ▶ When operands are of unequal bit length, the shorter operator is zero-filled in the most significant bit position

`wire [3:0] sum, a, b; wire cin, cout, d, e, f, g;`

`assign sum = f & a;`

`assign sum = f | a;`

`assign sum = {d, e, f, g} & a;`

`assign sum = {4{f}} | b;`

`assign sum = {4{f == g}} & (a + b);`

`assign sum[0] = g & a[2];`

`assign sum[2:0] = {3{g}} & a[3:1];`



## Synthesis: Operand Length

- ▶ Operator length is set to the longest member (both RHS & LHS are considered). Be careful.

```
wire [3:0] sum, a, b; wire cin, cout, d, e, f, g;  
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?
```

## Synthesis: Extra Operators

- ▶ Funky Conditional

- ▶ `cond_exp ? true_expr : false_expr`

```
wire [3:0] a,b,c; wire d, sel;
```

```
assign a = d ? b : c; // Mux with d as select
```

```
assign a = (b == c) ? (c + 'd1): 'o5; // good luck
```

- ▶ Reduction Logical

- ▶ Named for impact on your recreational time

- ▶ Unary operators that perform bit-wise operations on a single operand, reduce it to one bit

- ▶ `&, ~&, |, ~|, ^, ~^, ^~`

```
assign d = &a || ~^b ^ ^~c;
```

## Synthesis: Assign Statement

- ▶ The assign statement is sufficient to create all combinational logic
- ▶ What about this:

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

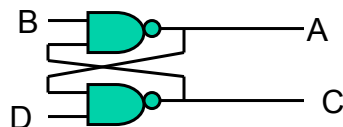
```
assign c = ~(d & a);
```

## Synthesis: Assign Statement

- ▶ The assign statement is sufficient to create all combinational logic
- ▶ What about this:

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

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



## Simple Behavioral Module

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

## Simple Behavioral Module

```
// Behavioral model of NAND gate
// Verilog 2001 syntax
module NAND (
    output out;
    input in1, in2);
    assign out = ~(in1 & in2);
endmodule
```

## Simple Behavioral Module

```
// Behavioral model of NAND gate
module NAND (
    output out;
    input in1, in2);
    // Uses Verilog builtin nand function
    // syntax is function id (args);
    nand i0(out, in1, in2);
endmodule
```

## Simple Structural Module

```
// Structural Module for NAND gate
module NAND (
    output out;
    input in1, 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
- ▶ They 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);`
    - `always @(*)`
    - `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
```

## Procedural NAND gate

```
// Procedural model of NAND gate
module NAND (out, in1, in2);
    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



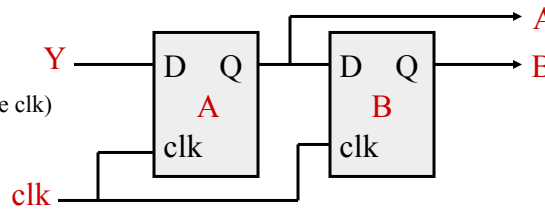
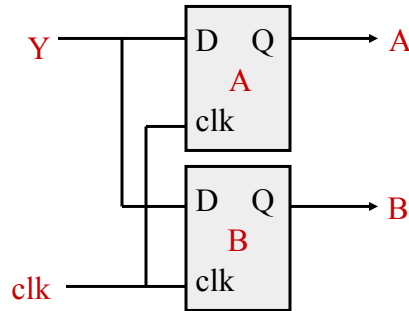
## Synthesized Circuits

```
always @(posedge clk)
begin
  A = Y;
  B = A;
end
```

```
always @(posedge clk)
begin
  B = A;
  A = Y;
end
```

```
always @(posedge clk)
begin
  A <= Y;
  B <= A;
end
```

```
always @(posedge clk)
begin
  B <= A;
  A <= Y
end
```



## 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

```
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
```

## Comparator Example

- ▶ 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;  
reg Cgt, Clt, Cne;  
always @(a or b)  
begin  
    Cgt <= (a > a);  
    Clt <= (a < b);  
    Cne <= (a != b);  
end  
endmodule
```

## Modeling a Flip Flop

- ▶ Use an `always` block to wait for clock edge

```
Module dff (  
    input clk, d;  
    output reg q);  
    always @(posedge clk)  
        d = q;  
endmodule
```

## Synthesis: Always Statement

- ▶ 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

- ▶ 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

```
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(.subtrahend(a), .subtractor(b), .difference(dif));

  assign alu = {4{(sel == 'b000)}} & sum
    | {4{(sel == 'b001)}} & dif;

  always @(posedge clk or posedge rst)
  if(rst) result <= 'h0;
  else result <= alu;

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);  
  parameter SIZE = 4;  
  input clk, rst;  
  input [SIZE-1:0] a,b;  
  input [2:0] sel;  
  output reg [SIZE-1:0] result;  
  wire [SIZE-1:0] sum, dif, alu;  
  
  adder #(.SIZE(SIZE)) u0(a,b,sum);  
  subber #(4) u1(.c(a), .d(b), .difference(dif));  
  
  assign alu = {SIZE{sel == 'b000}} & sum  
    | {SIZE{sel == '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

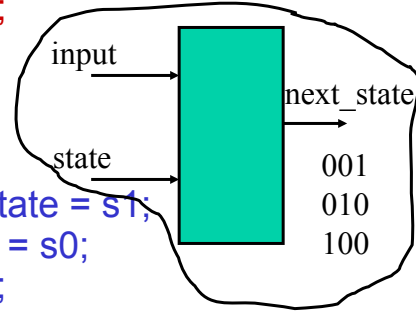
```
case ( <expression> )  
  <expression>: <statement>  
  <expression>, <expression>: <statement>  
  <expression>: <statement>  
  default: <statement>  
endcase
```

## Case Example

```
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

```
// 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;
  endcase
end
```



## Latch Inference

- ▶ Incompletely specified **if** and **case** statements cause the synthesizer to infer latches

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

- ▶ This infers a latch because it doesn't specify what to do when `cond = 0`
  - ▶ Fix by adding an **else**
  - ▶ In a case, fix by including **default**:

## Full vs. Parallel

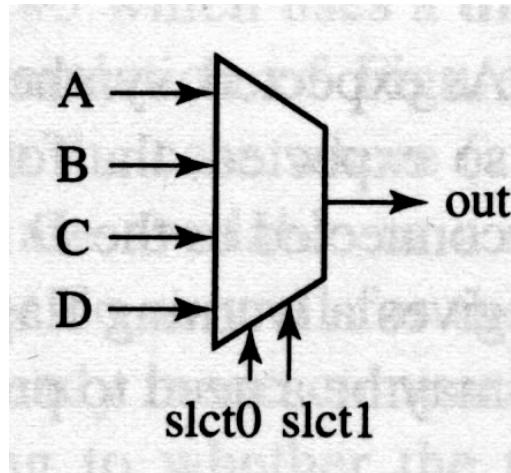
- ▶ **Case** statements check each case in sequence
- ▶ A **case** statement is **full** if all possible outcomes are accounted for
- ▶ A **case** statement is **parallel** if the stated alternatives are mutually exclusive
- ▶ These distinctions make a difference in how **cases** are translated to circuits...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

## Case full-par example

```
// 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
```

## Synthesis Result

- ▶ Note that full-par results in combinational logic

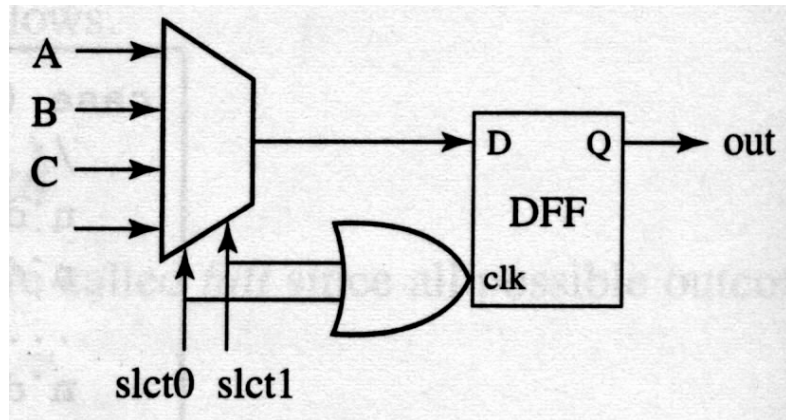


## Case notfull-par example

```
// a latch is synthesized because case is not full
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
```

## Synthesized Circuit

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



## Case full-notpar example

```
// because case is not parallel - priority encoding  
// but it is still full, so no latch...
```

```
// this uses a casez which treats ? as don't-care
```

```
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;
```

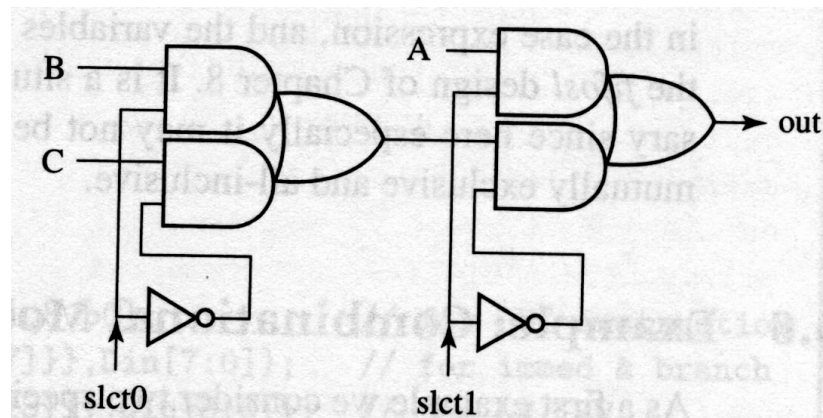
```
    default : out <= c;
```

```
  endcase
```

```
endmodule
```

## Synthesized Circuit

- ▶ 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

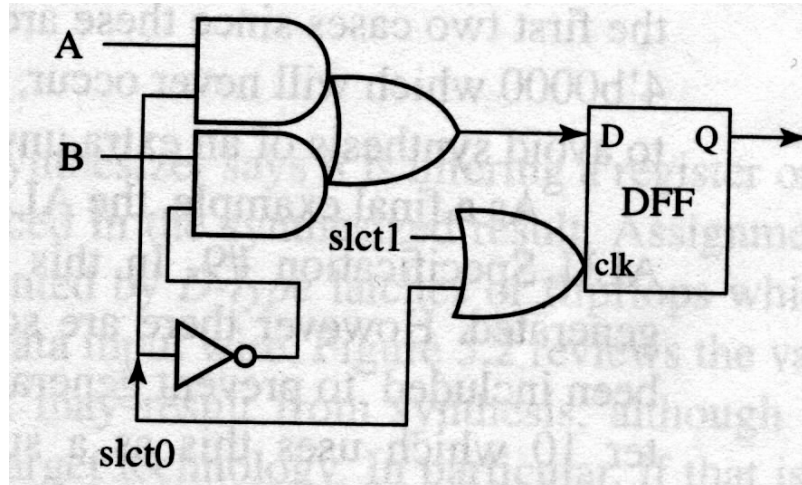


## Case notfull-notpar example

```
// because case is not parallel - priority encoding
// because case is not full - latch is inferred
// uses a casez which treats ? as don't-care
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;
    endcase
endmodule
```

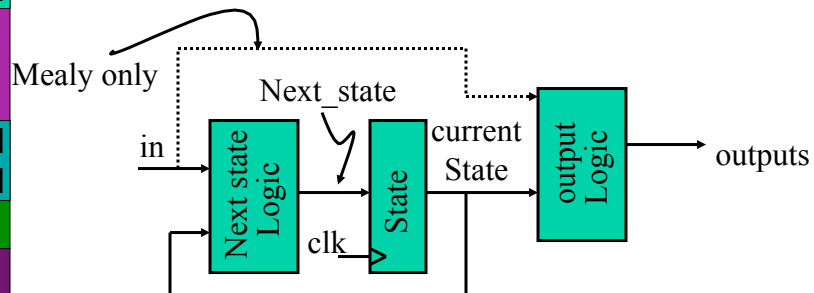
## Synthesized Circuit

- ▶ Not full and not parallel, infer a latch



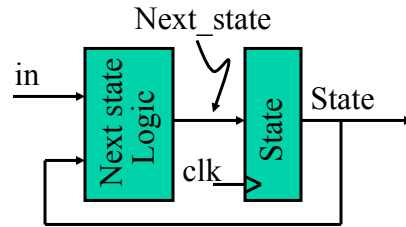
## FSM Description

- ▶ 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

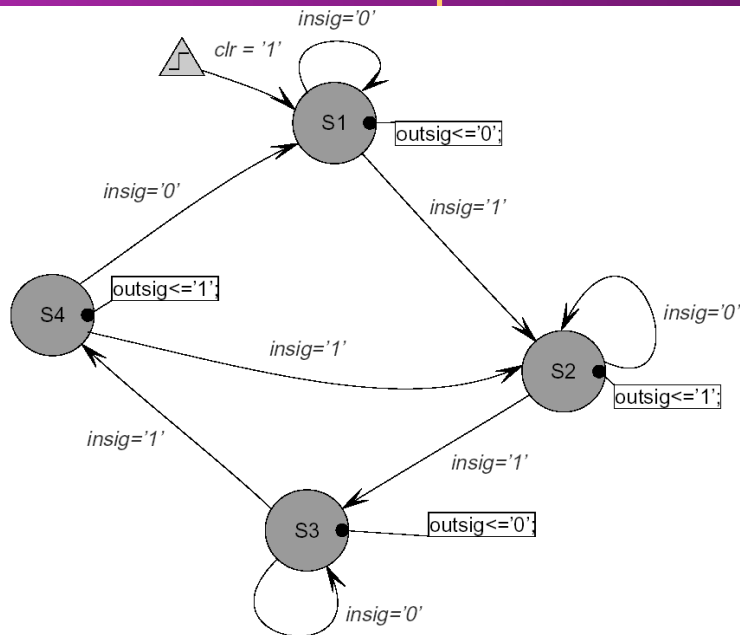


# Modeling State Machines

```
// 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
```



# FSM Description





## Verilog Version

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

## Verilog Version Continued...

```
// 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
```

## Verilog Version Continued...

```
// 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 #'s)
- ▶ 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

// don't do this...

```
always @(posedge a)  
    out = in1;
```

```
always @(posedge b)  
    out = in2;
```

## Combinational Always Blocks

▶ Be careful...

|                                                                             |                                                                                           |
|-----------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|
| <pre> always @(sel)   if (sel == 1)     out = in1;   else out = in2; </pre> | <pre> always @(sel or in1 or in2)   if (sel == 1)     out = in1;   else out = in2; </pre> |
|-----------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|

▶ Which one is a good mux?

## Sync vs. Async Register Reset

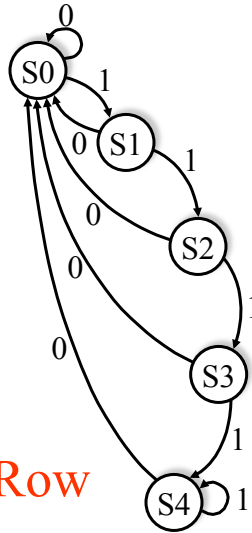
```

// 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;

```

# Finite State Machine



Four in a Row

# Textbook FSM

```
// Verilog HDL for "Ax", "see4" "behavioral"
// Four in a row detector - Allen Tanner
module see4 (clk, clr, insig, saw4);
input clk, clr, insig;
output saw4;

// define state encodings as parameters
parameter [2:0] s0 = 3'b000, s1 = 3'b001, s2 = 3'b010, s3 = 3'b011, s4 = 3'b100;

// define reg vars for state register and next_state logic
reg [2:0] state, next_state;

//define state register (with asynchronous active-low clear)
always @(posedge clk or negedge clr)
begin
    if (clr==0) 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 = s0;
        s2: if (insig) next_state = s3;
            else next_state = s0;
        s3: if (insig) next_state = s4;
            else next_state = s0;
        s4: if (insig) next_state = s4;
            else next_state = s0;
        default: next_state = s0;
    endcase
end

// now set the saw4. This could also be done in an always
// block... but in that case, saw4 would have to be
// defined as a reg.
assign saw4 = state == s4;
endmodule
```

## Textbook FSM

```
// Verilog HDL for "Ax", "see4" "behavioral"
// Four in a row detector - Allen Tanner

module see4 (clk, clr, insig, saw4);
input clk, clr, insig;
output saw4;

// define state encodings as parameters
parameter [2:0] s0 = 3'b000, s1 = 3'b001, s2 = 3'b010, s3 = 3'b011, s4 = 3'b100;

// define reg vars for state register and next_state logic
reg [2:0] state, next_state;

//define state register (with asynchronous active-low clear)
always @(posedge clk or negedge clr)
begin
    if (clr==0) 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 = s0;
        s2: if (insig) next_state = s3;
            else next_state = s0;
        s3: if (insig) next_state = s4;
            else next_state = s0;
        s4: if (insig) next_state = s4;
            else next_state = s0;
        default: next_state = s0;
    endcase
end

// now set the saw4. This could also be done in an always
// block... but in that case, saw4 would have to be
// defined as a reg.
assign saw4 = state == s4;
endmodule
```

Comments

Polarity?

Always use <= for FF

## Documented FSM

```
// Verilog HDL for "Ax", "see4" "behavioral"
// Four in a row detector - Allen Tanner

module see4 (clk, clr, insig, saw4);
input clk, clr, insig;
output saw4;

parameter [2:0] s0 = 3'b000; // initial state, saw at least 1 zero
parameter [2:0] s1 = 3'b001; // saw 1 one
parameter [2:0] s2 = 3'b010; // saw 2 ones
parameter [2:0] s3 = 3'b011; // saw 3 ones
parameter [2:0] s4 = 3'b100; // saw at least, 4 ones

reg [2:0] state, next_state;

always @(posedge clk or posedge clr) // state register
begin
    if (clr) state <= s0;
    else state <= next_state;
end

always @(insig or state) // next state logic
begin
    case (state)
        s0: if (insig) next_state = s1;
            else next_state = s0;
        s1: if (insig) next_state = s2;
            else next_state = s0;
        s2: if (insig) next_state = s3;
            else next_state = s0;
        s3: if (insig) next_state = s4;
            else next_state = s0;
        s4: if (insig) next_state = s4;
            else next_state = s0;
        default: next_state = s0;
    endcase
end

assign saw4 = state == s4;
endmodule
```

## 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

```
17 // Revision 0.01 - File Created
18 // Additional Comments:
19 //
20 ///
21 module mynand(A, B, Y);
22     input A;
23     input B;
24     output Y;
25
26     assign Y = ~(A & B);
27
28 endmodule
29
```

## NAND example

```
module mynand_tb_v;

    // Inputs
    reg A;
    reg B;

    // Outputs
    wire Y;

    // Instantiate the Unit Under Test (UUT)
    mynand uut (
        .A(A),
        .B(B),
        .Y(Y)
    );

    initial begin
        // Initialize Inputs
        A = 0;
        B = 0;

        // Wait 100 ns for global reset to finish
        #100;

        // Add stimulus here

    end
endmodule
```

## NAND Testbench

```
initial begin
    // Initialize Inputs
    A = 0;
    B = 0;

    // Wait 100 ns for global reset to finish
    #100;

    // Add stimulus here
    $display("AB = %b%b, Y = %b", A, B, Y);
    if (Y != 1) $display("ERROR - Y is %b, should be 1", Y);

    A = 1;
    #20 $display("AB = %b%b, Y = %b", A, B, Y);
    if (Y != 1) $display("ERROR - Y is %b, should be 1", Y);

    B = 1;
    A = 0;
    #20 $display("AB = %b%b, Y = %b", A, B, Y);
    if (Y != 1) $display("ERROR - Y is %b, should be 1", Y);

    A = 1;
    #20 $display("AB = %b%b, Y = %b", A, B, Y);
    if (Y != 0) $display("ERROR - Y is %b, should be 0", Y);

end
```



# NAND Tresetbench

```
initial begin
  // Initialize Inputs
  A = 0;
  B = 0;

  // Wait 100 ns for global reset to finish
  #100;

  // Add stimulus here
  $display("AB = %b%b, Y = %b", A, B, Y);
  if (Y != 1) $display("ERROR - Y is %b, should be 1", Y);

  A = 1;
  #20 $display("AB = %b%b, Y = %b", A, B, Y);
  if (Y != 1) $display("ERROR - Y is %b, should be 1", Y);

  B = 1;
  A = 0;
  #20 $display("AB = %b%b, Y = %b", A, B, Y);
  if (Y != 1) $display("ERROR - Y is %b, should be 1", Y);

  A = 1;
  #20 $display("AB = %b%b, Y = %b", A, B, Y);
  if (Y != 0) $display("ERROR - Y is %b, should be 0", Y);

end

if (Y != ~(A & B)) $display("ERROR - Y is %b, should be %b", Y, ~(A & B));
```

# NAND Tresetbench

```
integer i,j;
initial begin

  // Initialize Inputs
  A = 0;
  B = 0;

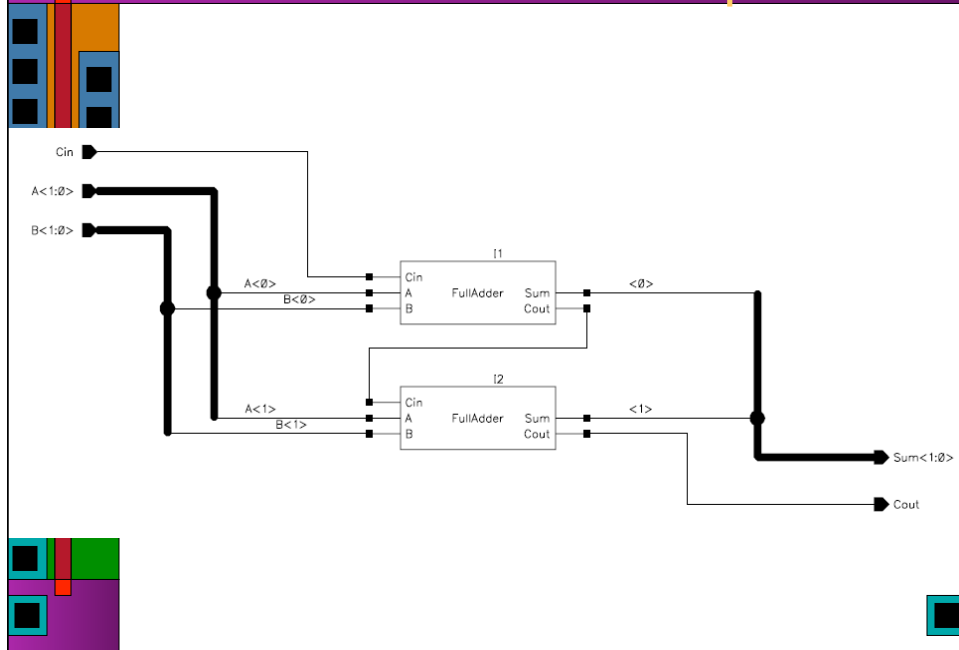
  // Wait 100 ns for global reset to finish
  #100;

  // Add stimulus here

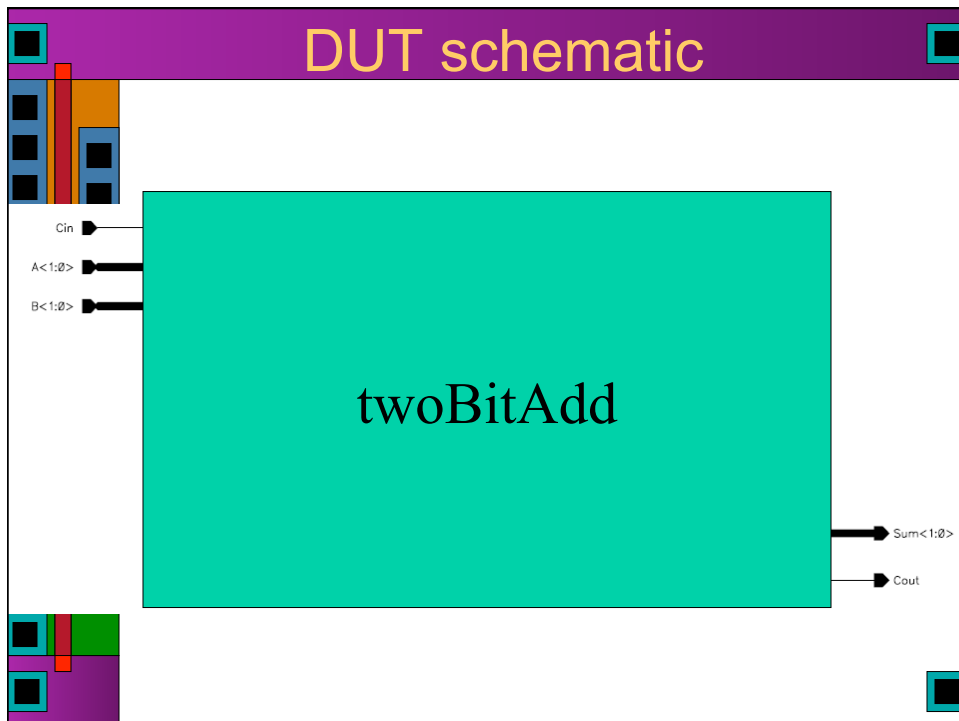
  for (i=0; i<2; i=i+1)
    for (j=0; j<2; j=j+1)
      begin
        A = i;
        B = j;
        #20 $display("AB = %b%b, Y = %b", A, B, Y);
        if (Y != ~(A & B))
          $display("ERROR - Y is %b, should be %b", Y, ~(A&B));
      end

end
```

## Two-bit adder example



## DUT schematic



## 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 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
```

## \$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...

## 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 the bits
      end
    end
  A = A+1; // shorthand notation for adding
end
$display("Simulation finished... ");
end
```

## Another adder example

```
module adder(a, b, y);
  input  [31:0]  a, b;
  output [31:0]  y;

  assign y = a + b;
endmodule

module testbench();
  reg  [31:0] testvectors[1000:0];
  reg      clk;
  reg  [10:0] vectornum, errors;
  reg  [31:0] a, b, expectedy;
  wire [31:0] y;

  // instantiate device under test
  adder dut(a, b, y);

  // read the test vector file and initialize test
  initial
  begin
    $readmemh("testvectors.tv", testvectors);
    vectornum = 0; errors = 0;
  end
end
```

## Another adder example

```
// generate a clock to sequence tests
always
begin
    clk = 0; #50; clk = 1; #50;
end

// on each clock step, apply next test
always @(posedge clk)
begin
    a = testvectors[vectornum*3];
    b = testvectors[vectornum*3 + 1];
    expectedy = testvectors[vectornum*3 + 2];
end

// then check for correct results
always @(negedge clk)
begin
    vectornum = vectornum + 1;
    if (y != expectedy) begin
        $display("Inputs were %h, %h", a, b);
        $display("Expected %h but actual %h", expectedy, y);
        errors = errors + 1;
    end
end
end
```

## Another adder example

```
// halt at the end of file
always @(vectornum)
begin
    if (vectornum == 100 || testvectors[vectornum*3] === 32'bx)
    begin
        $display("Completed %d tests with %d errors.",
            vectornum, errors);
        $finish;
    end
end
endmodule

testvectors.tv file:
00000000
00000000
00000000

00000001
00000000
00000001

ffffff
00000003
00000002

12345678
12345678
2468acf0
```