



(System)Verilog Tutorial

Aleksandar Milenković

The LaCASA Laboratory
Electrical and Computer Engineering Department
The University of Alabama in Huntsville

Email: milenka@ece.uah.edu

Web: <http://www.ece.uah.edu/~milenka>

<http://www.ece.uah.edu/~lacasa>



Outline

- Introduction
- Combinational Logic
- Sequential Logic
- Memories
- Testbenches





Introduction

- Verilog is a Hardware Description Language (HDL)
- HDLs are used for logic *simulation* and *synthesis*
 - Simulation: inputs are applied to a module and the outputs are checked to verify that the module operates correctly
 - Synthesis: the textual description of a module is transformed into logic gates
- Allow description of a digital system at
 - Behavioral Level – describes how the outputs are computed as functions of the inputs
 - Structural Level – describes how a module is composed of simpler modules of basic primitives (gates or transistors)
- Design styles: Top-Down vs. Bottom-Up





History

- Developed by Phil Moorby at Gateway Design Automation as a proprietary language for logic simulation in 1984
- Gateway is acquired by Cadence in 1989
- Made an open standard in 1990 under the control of Open Verilog International
- Became an IEEE standard in 1995 and
- Updated in 2001 [IEEE1364-01]
- In 2005, it was updated again with minor clarifications
- SystemVerilog [IEEE 1800-2009] was introduced
 - Streamlines many of the annoyances of Verilog and adds high-level programming language features that have proven useful in verification



Modules

- A block of hardware with inputs and outputs is called a module

```
module sillyfunction(input logic a, b, c,  
                    output logic y);  
  
    assign y = ~a & ~b & ~c |  
              a & ~b & ~c |  
              a & ~b & c;  
  
endmodule
```

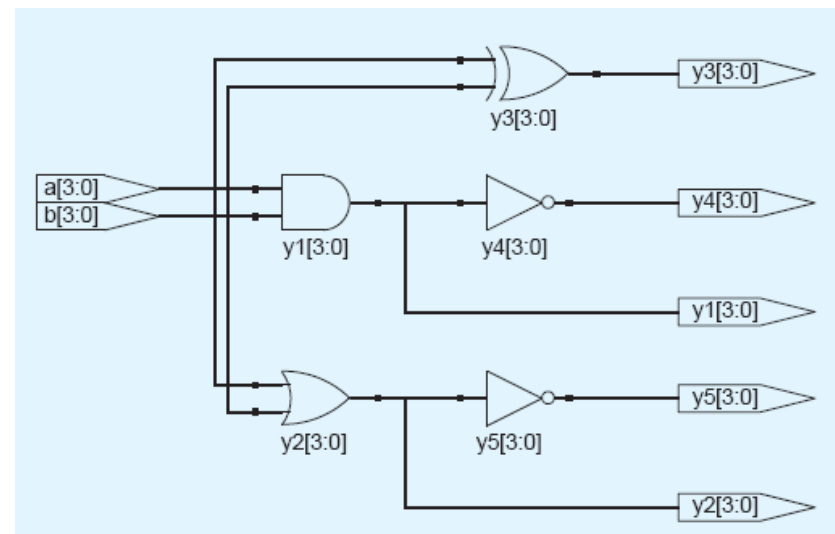
- A module begins with a listing of the inputs and outputs
- `assign` statement describes combinational logic
 - `~` indicates NOT, `&` indicates AND, and `|` indicates OR
 - logic signals such as the inputs and outputs are Boolean variables (0 or 1). They may also have floating (z) and undefined values (x).
- The logic type was introduced in SystemVerilog
 - It supersedes the `reg` type, which was a perennial source of confusion in Verilog
 - should be used everywhere except on nets with multiple drivers



Combinational Logic

- Operators: &, |, ~, ^
- Operands: a, b, y1-y5
- Expressions: e.g., (a & b)
- Statements: e.g., y5 = ~(a | b);
- Assign statement implies combinational logic
- Assign indicates a continuous assignment statement
 - Left-hand side of expression is updated any time the right-hand side changes (a or b)
- Assign statements are concurrent

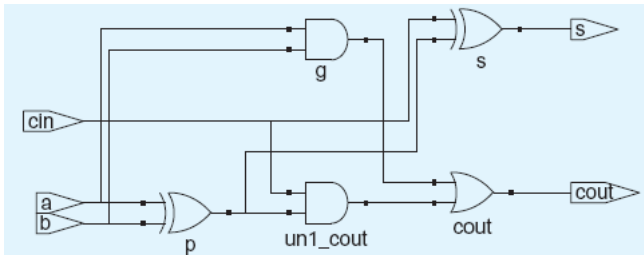
```
module gates(input logic [3:0] a, b,  
            output logic [3:0] y1, y2, y3, y4, y5);  
  /* Five different two-input logic  
   gates acting on 4 bit busses */  
  assign y1 = a & b; // AND  
  assign y2 = a | b; // OR  
  assign y3 = a ^ b; // XOR  
  assign y4 = ~(a & b); // NAND  
  assign y5 = ~(a | b); // NOR  
endmodule
```



Combinational Logic (cont'd)

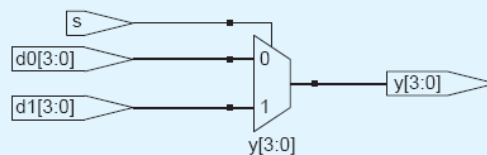
```
// full adder
module fa (input logic a, b, cin,
output logic s, cout);

    assign s = a ^ b ^ cin;
    assign cout = (a & b) | (cin & (a | b));
endmodule
```



```
/* 4-bit mux; selects one of two 4-bit
inputs d0 or d1 */
module mux2_4 (input logic [3:0] d0, d1,
input logic s, output logic [3:0] y);

    assign y = s ? d1 : d0;
endmodule
```



■ Comments

- Comments beginning with /* continue, possibly across multiple lines, to the next */
- Comments beginning with // continue to the end of the line

■ SystemVerilog is case-sensitive

- y1 and Y1 are different signals

■ Conditional assignment

- *conditional operator ?*: chooses, based on a first expression, between a second and third expression
- The first expression is called the *condition*. If the condition is 1, the operator chooses the second expression. If the condition is 0, the operator chooses the third expression
- Equivalent to this expression:
If s = 1, then y = d1.
If s = 0, then y = d0.

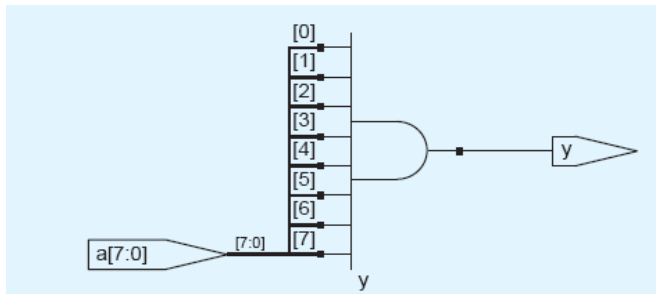


Combinational Logic (cont'd)

```
module and8 (input logic [7:0] a,
             output logic y);

    assign y = & a;
    // &a is much easier to write than
    // assign y = a[7] & a[6] & a[5] & a[4]
    // & a[3] & a[2] & a[1] & a[0];

endmodule
```

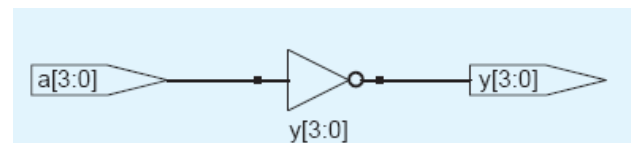


- Reduction operators imply a multiple-input gate acting on a single bus
 - $|$, \wedge , $\sim\&$, and $\sim|$ reduction operators are available for OR, XOR, NAND, and NOR

```
/* an array of inverters */
module invA4 (input logic [3:0] a,
             output logic [3:0] y);

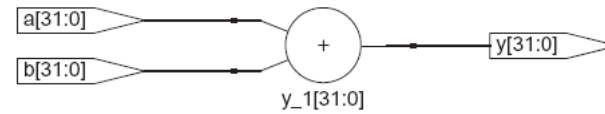
    assign y = ~a;

endmodule
```



Combinational Logic (cont'd)

```
/* 32-bit adder */  
module adder (a, b, y);  
  input logic [31:0] a;  
  input logic [31:0] b;  
  output logic [31:0] y;  
  
  assign y = a + b;  
endmodule
```



```
/* */  
module whatami(input logic [3:0] d0, d1, d2, d3,  
  input logic [1:0] s, output logic [3:0] y);  
  
  assign y = s[1] ? (s[0] ? d3 : d2)  
            : (s[0] ? d1 : d0);  
endmodule
```



Combinational Logic (cont'd)

```
module fulladder(input logic a, b, cin,
output logic s, cout);

    logic p, g;

    assign p = a ^ b;
    assign g = a & b;
    assign s = p ^ cin;
    assign cout = g | (p & cin);
endmodule
```

- Internal variables
 - Neither inputs or outputs
- Concurrency
 - Order of assign statements does not matter



Verilog Operators and Operator Precedence

TABLE A.1 SystemVerilog operator precedence

	Op	Meaning
H i g h e s t	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PLUS, MINUS
	<<, >>	Logical Left / Right Shift
	<<<, >>>	Arithmetic Left / Right Shift
	<, <=, >, >=	Relative Comparison
L o w e s t	==, !=	Equality Comparison
	&, ~&	AND, NAND
	^, ~^	XOR, XNOR
	~, ~	OR, NOR
	?:	Conditional

- Similar to precedence in other languages
 - assign cout = g | p & cin;
- Subtraction involves a two's complement and addition
- Multipliers and shifters use substantially more area (unless they involve easy constants)
- Division and modulus in hardware is so costly that it may not be synthesizable
- Equality comparisons (==) imply N 2-input XORs to determine equality of each bit and an N -input AND to combine all of the bits
- Relative comparison involves a subtraction

Verilog Numbers

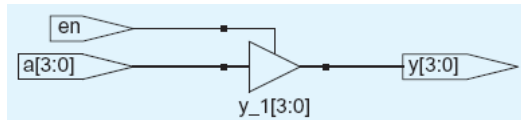
Numbers	Bits	Base	Val	Stored
3'b101	3	2	5	101
'b11	?	2	3	000...0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00...0101010
'1	?	n/a		11...111

- System Verilog numbers can specify their base and size (the number of bits used to represent them)
- The format for declaring constants is N'Bvalue
 - N is the size in bits
 - B is the base, and
 - Value gives the value
 - 9'h25 indicates a 9-bit hex number (37 decimal or 000100101 in binary)
- Base: 'b for binary (base 2), 'o for octal (base 8), 'd for decimal (base 10), and 'h for hexadecimal (base 16).
 - If the base is omitted, the base defaults to decimal
- Size: If the size is not given, the number is assumed to have as many bits as the expression in which it is being used.
 - Zeros are automatically padded on the front of the number to bring it up to full size
 - E.g., if w is a 6-bit bus, assign w = 'b11 gives w the value 000011. It is better practice to explicitly give the size. An exception is that '0 and '1 are SystemVerilog shorthands for filling a bus with all 0s and all 1s.



Z's and X's

```
module tristate(input logic [3:0] a,  
               input logic en, output tri [3:0] y);  
  
    assign y = en ? a : 4'bz;  
endmodule
```



- Z indicates a floating value
 - Useful for describing a tristate buffer; its output floats when the enable is 0
 - A bus can be driven by several tristate buffers, exactly one of which should be enabled
- y is declared as `tri` rather than `logic`
 - Logic signals can only have a single driver
 - Tristate busses can have multiple drivers, so they should be declared as a *net*
- Two types of nets: `tri` and `triereg`
 - One driver is active at a time, and the net takes that value;
 - If no drivers are enabled, the `tri` net floats (z), while the `triereg` net retains the previous value
- If no type is specified for an input or output, `tri` is assumed

Z's and X's (cont'd)

SystemVerilog AND gate truth table

&		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

- SystemVerilog signal values are 0, 1, z, and x
- x indicates an invalid logic level
 - E.g., if a bus is simultaneously driven to 0 and 1 by two enabled tristate buffers (or other gates), the result is x, indicating contention
 - At the start of simulation, state nodes such as flip-flop outputs are initialized to an unknown state (x)
- Constants starting with z or x are padded with leading zs or xs (instead of 0s) to reach their full length when necessary

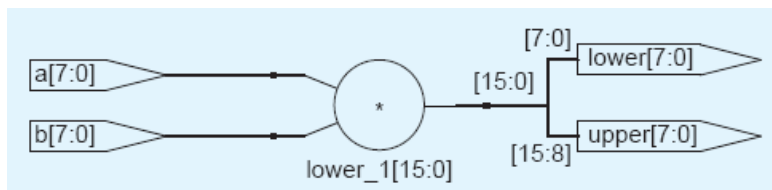


Bit swizzling

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

```
module mul(input logic [7:0] a, b,
           output logic [7:0] upper, lower);

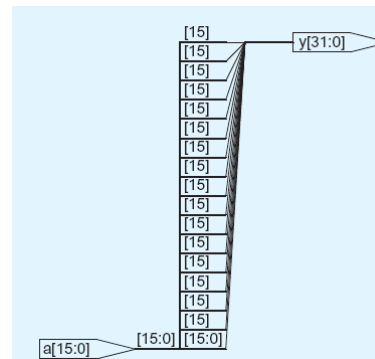
    assign {upper, lower} = a*b;
endmodule
```



```
module signextend(input logic [15:0] a,
                  output logic [31:0] y);

    assign y = {{16{a[15]}}, a[15:0]};
endmodule
```

- $c[2:1]$ – subset, 2-bit vector
- Concatenate operator: $\{ \}$
 - $\{3\{d[0]\}$ indicates three copies of $d[0]$
 - $3'b101$ is a 3-bit binary constant
 - y is a 9-bit vector
 - If y were wider than 9 bits, zeros would be placed in the most significant bit positions

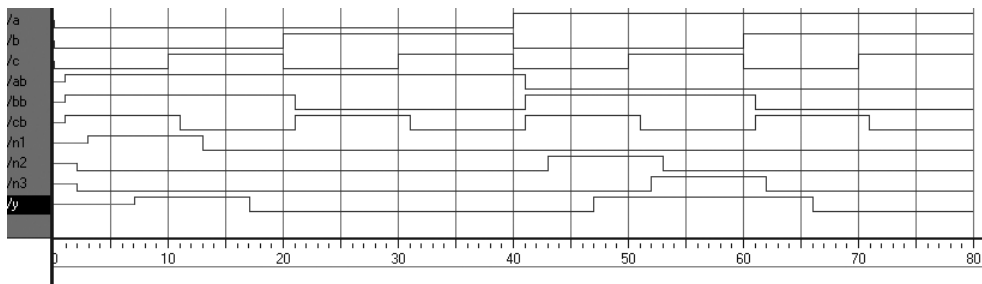


Delays

```
`timescale 1ns/1ps
module example(input logic a, b, c,
               output logic y);

    logic ab, bb, cb, n1, n2, n3;

    assign #1 {ab, bb, cb} = ~{a, b, c};
    assign #2 n1 = ab & bb & cb;
    assign #2 n2 = a & bb & cb;
    assign #2 n3 = a & bb & c;
    assign #4 y = n1 | n2 | n3;
endmodule
```

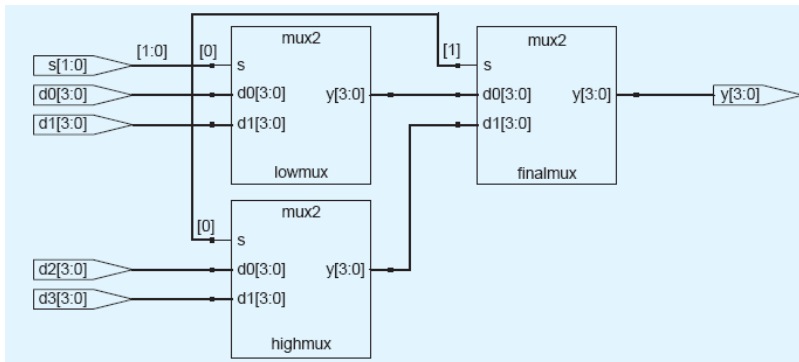


- Statements may be associated with delays specified in arbitrary units
- Helpful during simulation
 - Predict how fast a circuit will work (if you specify meaningful delays)
 - Debug the cause and effect
- Delays are ignored during synthesis
 - Delay of a gate produced by the synthesizer depends on its *tpd* and *tcd* specifications
- Timescale directive
 - ``timescale unit/step`
- `#` symbol is used to indicate the number of units of delay
 - It can be placed in assign statements, as well as nonblocking (`<=`) and blocking (`=`) assignments

Structural Model

- Describe a module in terms of how it is composed of simpler modules
- Mux4 out of Mux2s
- Try Dec4to16 using Dec2to4s?

```
module mux2_4 (input logic [3:0] d0, d1,  
              input logic s,  
              output logic [3:0] y);  
  
    assign y = s ? d1 : d0;  
  
endmodule
```



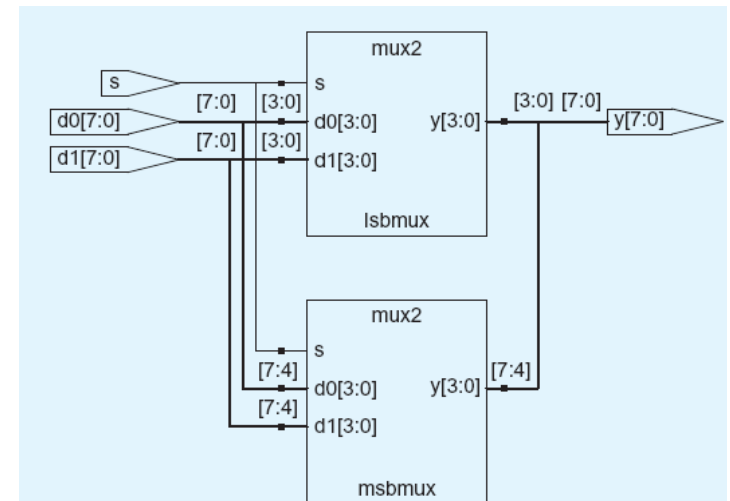
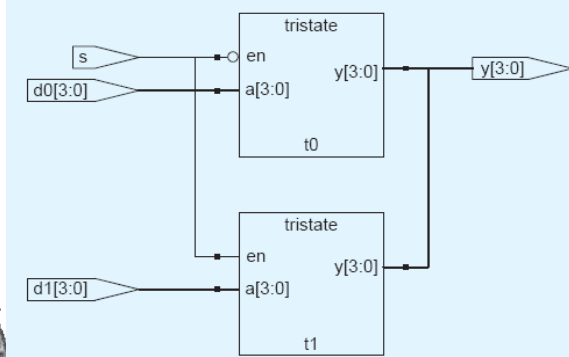
```
module mux4_4 (input logic [3:0] d0, d1, d2, d3,  
              input logic [1:0] s,  
              output logic [3:0] y);  
  
    logic [3:0] mol, moh;  
    mux2_4 lowmux(d0, d1, s[0], mol);  
    mux2_4 highmux(d2, d3, s[0], moh);  
    mux2_4 finalmux(mol, moh, s[1], y);  
  
endmodule
```

Structural Modeling

```
module tristate_4 (input logic [3:0] a,  
                 input logic en,  
                 output tri [3:0] y);  
  
    assign y = en ? a : 4'bz;  
  
endmodule
```

```
// mux2 using tristate  
module mux2_4 (input logic [3:0] d0, d1,  
              input logic s,  
              output tri [3:0] y);  
  
    tristate(d0, ~s, y);  
    tristate(d1, s, y);  
  
endmodule
```

```
// demonstrate selection of ranges on a bus  
module mux2_8 (input logic [7:0] d0, d1,  
              input logic s,  
              output logic [7:0] y);  
  
    mux2_4 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);  
    mux2_4 msbmux(d0[7:4], d1[7:4], s, y[7:4]);  
  
endmodule
```





Sequential Logic

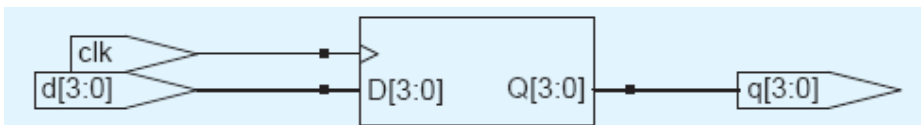
- HDL synthesizers recognize certain idioms and turn them into specific sequential circuits
- Other coding styles may simulate correctly, but synthesize into circuits with blatant or subtle errors
- Note: Use the proper idioms to describe registers and latches described here



Registers

4-bit register

```
module flop(input logic clk,  
            input logic [3:0] d,  
            output logic [3:0] q);  
  
    always_ff @(posedge clk)  
        q <= d;  
endmodule
```

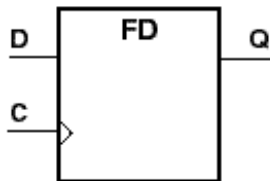


- always statement form:
 - always @(sensitivity list) statement;
- The statement is executed only when the event specified in the sensitivity list occurs
 - $q \leq d$ (pronounced “q gets d”) only on the positive edge of the clock and otherwise remembers the old state of q
- \leq is called a *nonblocking assignment*
 - A group of nonblocking assignments is evaluated concurrently
 - all of the expressions on the right-hand sides are evaluated before any of the left-hand sides are updated

Flip-flops

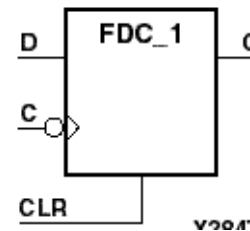
D-FF with Positive Clock

```
module flop (C, D, Q);  
  input C, D;  
  output Q;  
  reg Q;  
  
  always @(posedge C)  
  begin  
    Q = D;  
  end  
endmodule
```



D-FF with Negative-edge Clock and Async. Clear

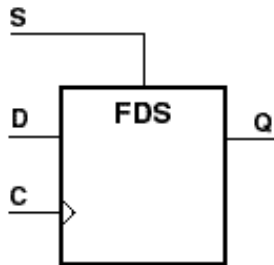
```
module flop (C, D, CLR, Q);  
  input C, D, CLR;  
  output Q;  
  reg Q;  
  
  always @(negedge C or posedge CLR)  
  begin  
    if (CLR)  
      Q = 1'b0;  
    else  
      Q = D;  
    end  
  end  
endmodule
```



Flip-flops

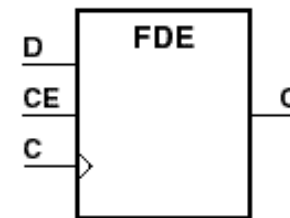
D-FF with Positive-Edge Clock and Synchronous Set

```
module flop (C, D, S, Q);  
  input C, D, S;  
  output Q;  
  reg Q;  
  
  always @(posedge C)  
  begin  
    if (S)  
      Q = 1'b1;  
    else  
      Q = D;  
  end  
endmodule
```



D-FF with Positive-Edge Clock and Clock Enable

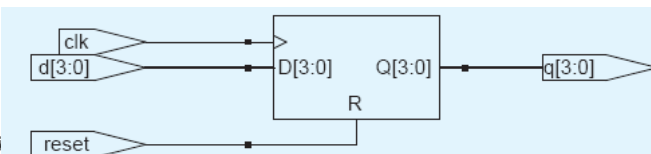
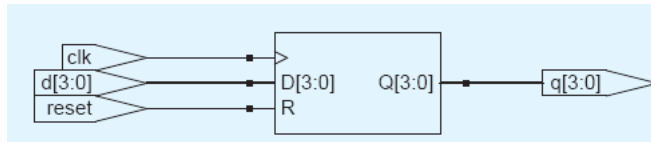
```
module flop (C, D, CE, Q);  
  input C, D, CE;  
  output Q;  
  reg Q;  
  
  always @(posedge C)  
  begin  
    if (CE)  
      Q = D;  
  end  
endmodule
```



Resettable Registers

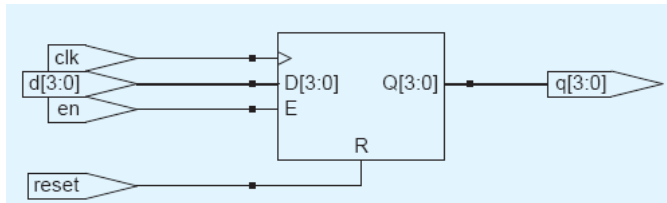
```
module flopr_s(input logic clk,  
  input logic reset,  
  input logic [3:0] d,  
  output logic [3:0] q);  
  
  // synchronous reset  
  always_ff @(posedge clk)  
    if (reset) q <= 4'b0;  
    else q <= d;  
endmodule  
  
module flopr_a(input logic clk,  
  input logic reset,  
  input logic [3:0] d,  
  output logic [3:0] q);  
  
  // asynchronous reset  
  always_ff @(posedge clk, posedge reset)  
    if (reset) q <= 4'b0;  
    else q <= d;  
endmodule
```

- When simulation begins or power is first applied to a circuit, the output of the flop is unknown (x in Verilog)
- => Use resettable registers so that on power up you can put your system in a known state
- Synchronous or asynchronous reset
 - Synchronous: occurs on the active edge of the clock
 - Asynchronous: occurs immediately
- Note:
 - Synchronous reset takes fewer transistors and reduces the risk of timing problems on the trailing edge of reset
 - However, if clock gating is used, care must be taken that all flipflops reset properly at startup



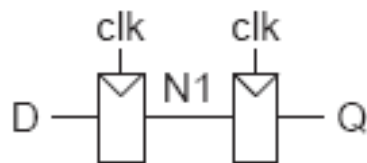
Enabled Resettable Registers

```
module flopenr(input logic clk,  
              input logic reset,  
              input logic en,  
              input logic [3:0] d,  
              output logic [3:0] q);  
  
    // synchronous reset  
    always_ff @(posedge clk)  
        if (reset) q <= 4'b0;  
        else if (en) q <= d;  
endmodule
```

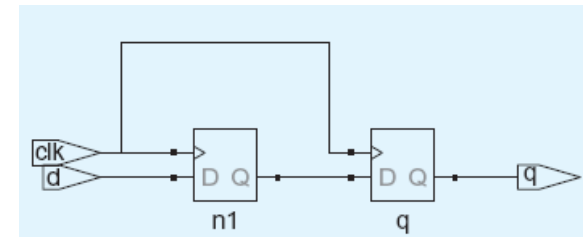


Synchronizer Circuit

```
module sync(input logic clk,  
            input logic d,  
            output logic q);  
  
    logic n1;  
  
    always_ff @(posedge clk)  
    begin  
        n1 <= d;  
        q <= n1;  
    end  
endmodule
```



- Multiple statements in always blocks are marked by a begin-end pair
- Nonblocking assignments

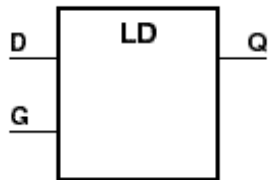


Latches

Latch with Positive Gate

```
module latch (G, D, Q);
  input G, D;
  output Q;
  reg Q;

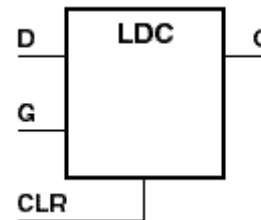
  always @(G or D)
    begin
      if (G)
        Q = D;
    end
endmodule
```



Latch with Positive Gate and Asynchronous Clear

```
module latch (G, D, CLR, Q);
  input G, D, CLR;
  output Q;
  reg Q;

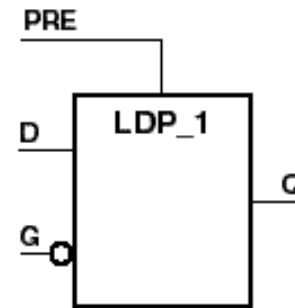
  always @(G or D or CLR)
    begin
      if (CLR)
        Q = 1'b0;
      else if (G)
        Q = D;
    end
endmodule
```



4-bit Latch

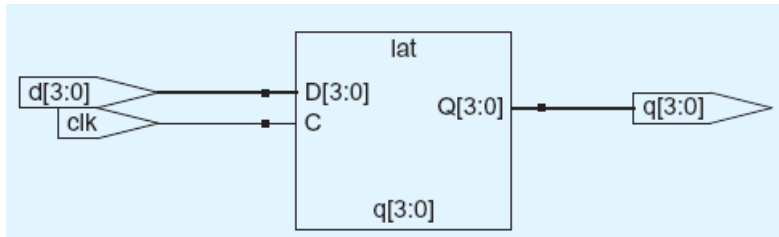
4-bit Latch with Inverted Gate and Asynchronous Preset

```
module latch (G, D, PRE, Q);  
  input G, PRE;  
  input [3:0] D;  
  output [3:0] Q;  
  reg [3:0] Q;  
  
  always @(G or D or PRE)  
  begin  
    if (PRE)  
      Q = 4'b1111;  
    else if (~G)  
      Q = D;  
    end  
endmodule
```



4-bit Latch

```
module latch(input logic clk,  
            input logic [3:0] d,  
            output logic [3:0] q);  
  
    always_latch  
        if (clk) q <= d;  
endmodule
```



- always_latch
 - New construct in SystemVerilog
 - Equivalent to always @(clk, d)

Counters

```

module counter(input logic clk,
               input logic reset,
               output logic [3:0] q);

    always_ff @(posedge clk)
        if (reset) q <= 4'b0;
        else q <= q+1;
endmodule

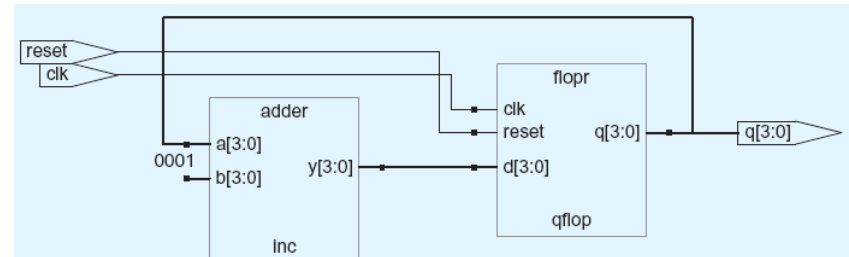
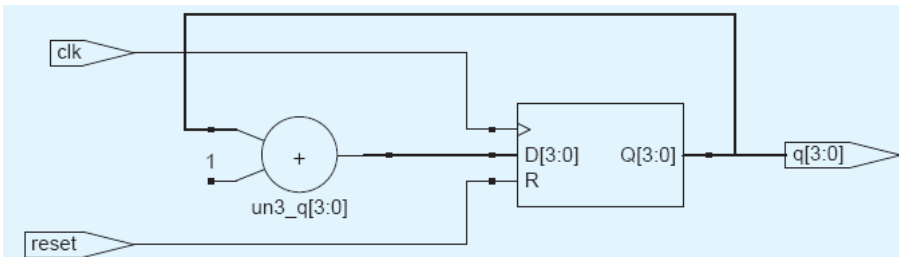
```

```

module counter(input logic clk,
               input logic reset,
               output logic [3:0] q);
    logic [3:0] nextq;

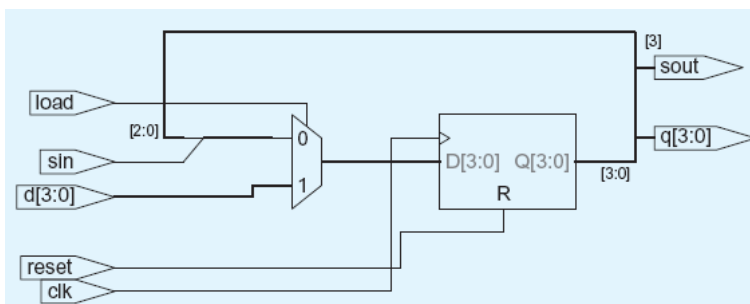
    flopr qflop(clk, reset, nextq, q);
    adder inc(q, 4'b0001, nextq);
endmodule

```



Shifters

```
module shiftreg(input logic clk,  
  input logic reset, load,  
  input logic sin,  
  input logic [3:0] d,  
  output logic [3:0] q,  
  output logic sout);  
  
  always_ff @(posedge clk)  
    if (reset) q <= 0;  
    else if (load) q <= d;  
    else q <= {q[2:0], sin};  
  
  assign sout = q[3];  
endmodule
```



Combinational Logic with Always Blocks

```
module inv(input logic [3:0] a,
           output logic [3:0] y);

    always_comb
        y = ~a;
endmodule
```

```
module fulladder(input logic a, b, cin,
                 output logic s, cout);
    logic p, g;

    always_comb
    begin
        p = a ^ b; // blocking
        g = a & b; // blocking
        s = p ^ cin;
        cout = g | (p & cin);
    end
endmodule
```

- `always_comb`
 - Model combinational logic
 - Reevaluates the statements inside the always block any time any of the signals on the right-hand side of `<=` or `=` inside the always statement change
 - Preferred way of describing combinational logic in SystemVerilog
 - Equivalent to `always @(*)`
- `=` in the always statement is called a *blocking assignment*
 - Preferred for combinational logic in SystemVerilog
- A group of blocking assignments are evaluated in the order they appear in the code



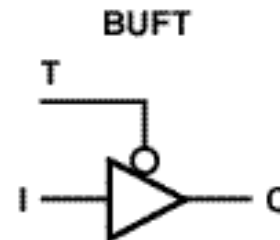
Tri-State

```
module three_st (T, I, O);
  input T, I;
  output O;
  reg O;

  always @(T or I)
  begin
    if (~T)
      O = I;
    else
      O = 1'bZ;
    end
  end
endmodule
```

```
module three_st (T, I, O);
  input T, I;
  output O;

  assign O = (~T) ? I: 1'bZ;
endmodule
```



Muxes

```
module mux2( input d0, d1, s,
             output y);

    reg y;

    always @(s or d0 or d1)
    begin : MUX
        case(s)
            1'b0 : y = d0;
            1'b1 : y = d1;
        endcase
    end
endmodule
```

```
module mux2( input d0, d1, s,
             output y);

    reg y;

    always @(*)
    begin : MUX
        case(s)
            1'b0 : y = d0;
            1'b1 : y = d1;
        endcase
    end
endmodule
```

Warning: All the signals on the left side of assignments in always blocks must be declared as reg. However, declaring a signal as reg does not mean the signal is actually a register.



Case Statements

```
module sevenseg(input logic [3:0] data,
                output logic [6:0] segments);

    always_comb
        case (data)
            // abc_defg
            0: segments = 7'b111_1110;
            1: segments = 7'b011_0000;
            2: segments = 7'b110_1101;
            3: segments = 7'b111_1001;
            4: segments = 7'b011_0011;
            5: segments = 7'b101_1011;
            6: segments = 7'b101_1111;
            7: segments = 7'b111_0000;
            8: segments = 7'b111_1111;
            9: segments = 7'b111_1011;
            default: segments = 7'b000_0000;
        endcase
    endmodule
```

```
module decoder3_8(input logic [2:0] a,
                 output logic [7:0] y);

    always_comb
        case (a)
            3'b000: y = 8'b00000001;
            3'b001: y = 8'b00000010;
            3'b010: y = 8'b00000100;
            3'b011: y = 8'b00001000;
            3'b100: y = 8'b00010000;
            3'b101: y = 8'b00100000;
            3'b110: y = 8'b01000000;
            3'b111: y = 8'b10000000;
        endcase
    endmodule
```





If-else, Casez

```
module priorityckt(input logic [3:0] a,
  output logic [3:0] y);

  always_comb
    if (a[3]) y = 4'b1000;
    else if (a[2]) y = 4'b0100;
    else if (a[1]) y = 4'b0010;
    else if (a[0]) y = 4'b0001;
    else y = 4'b0000;
endmodule
```

```
module priority_casez(input logic [3:0] a,
  output logic [3:0] y);

  always_comb
    casez(a)
      4'b1???: y = 4'b1000;
      4'b01??: y = 4'b0100;
      4'b001?: y = 4'b0010;
      4'b0001: y = 4'b0001;
      default: y = 4'b0000;
    endcase
endmodule
```



Blocking and Non-blocking Assignments

- Blocking assignments (use =)
 - A group of blocking assignments inside a begin-end block is evaluated sequentially
- Non-blocking assignments (use <=)
 - A group of non-blocking assignments are evaluated in parallel; all of the statements are evaluated before any of the left sides are updated.

Blocking vs. Nonblocking

- 1. Use `always_ff @(posedge clk)` and nonblocking assignments to model synchronous sequential logic
- 2. Use continuous assignments to model simple combinational logic
- Use `always_comb` and blocking assignments to model more complicated combinational logic where the `always` statement is helpful
- 4. Do not make assignments to the same signal in more than one `always` statement or continuous assignment statement. Exception: tristate busses.

```
always_ff @(posedge clk)
begin
    n1 <= d; // nonblocking
    q <= n1; // nonblocking
end
```

```
assign y = s ? d1 : d0;
```

```
always_comb
begin
    p = a ^ b; // blocking
    g = a & b; // blocking
    s = p ^ cin;
    cout = g | (p & cin);
end
```



Blocking vs. Nonblocking (cont'd)

```
// nonblocking assignments (not recommended)
module fulladder(input logic a, b, cin,
  output logic s, cout);
  logic p, g;

  always_comb
  begin
    p <= a ^ b; // nonblocking
    g <= a & b; // nonblocking
    s <= p ^ cin;
    cout <= g | (p & cin);
  end
endmodule
```

```
module fulladder(input logic a, b, cin,
  output logic s, cout);
  logic p, g;

  always_comb
  begin
    p = a ^ b; // blocking
    g = a & b; // blocking
    s = p ^ cin;
    cout = g | (p & cin);
  end
```

```
a=b=cin=0; // initial conditions
p=g=0;
a=1; // a changes => trigger always block
-----
/* all nonblocking assignments are
concurrently executed; all assume a=1, b=0,
cin=0 */
p<=1; // sch. change for next delta (1)
g<=0; // no changes
s<=0; // no changes
cout<=0; // no changes
-----
/* move time; p has changed, re-evaluate
always block again) (a=1, p=1)*/
.....
```

```
a=b=cin=0; // initial conditions
a=1; // a changes => trigger always block
/* Note that p and g get their new value
before s and cout are computed because of
the blocking assignments. This is
important because we want to compute s and
cout using the new values of p and g.*/
-----
p=1; // immediate assignment
g=0;
s=1;
cout=0;
```

Blocking vs. Nonblocking (cont'd)

```
module sync(input logic clk,  
            input logic d,  
            output logic q);  
  
    logic n1;  
  
    always_ff @(posedge clk)  
    begin  
        n1 <= d;  
        q <= n1;  
    end  
endmodule
```

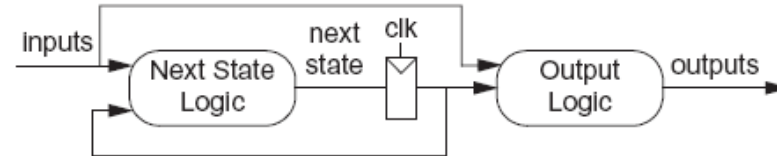
```
// Bad implementation using blocking assignments  
module syncbad(input logic clk,  
               input logic d,  
               output logic q);  
  
    logic n1;  
  
    always_ff @(posedge clk)  
    begin  
        n1 = d; // blocking  
        q = n1; // blocking  
    end  
endmodule
```

Blocking vs. Nonblocking (cont'd)

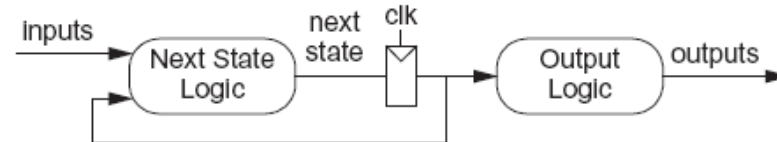
```
module shiftreg (input clk,
                 input sin,
                 output reg [3:0] q);
    always @(posedge clk)
    begin
        q[0] <= sin;
        q[1] <= q[0];
        q[2] <= q[1];
        q[3] <= q[2];
        // could be replaced by
        // q <= {q[2:0], sin}
    end
endmodule
```

```
// this is incorrect shift register
module shiftreg (input clk,
                 input sin,
                 output reg [3:0] q);
    always @(posedge clk)
    begin
        q[0] = sin;
        q[1] = q[0];
        q[2] = q[1];
        q[3] = q[2];
    end
endmodule
```

Finite State Machines



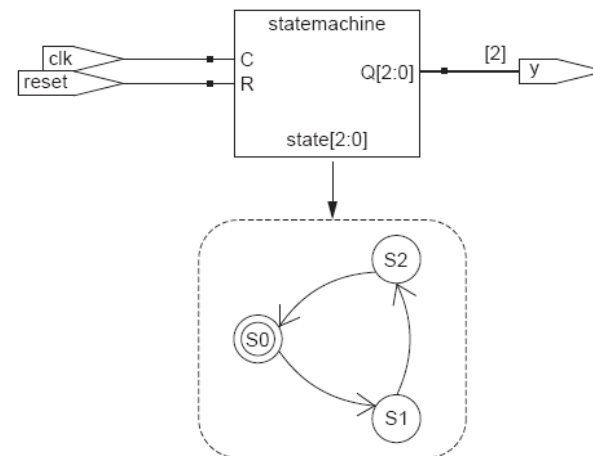
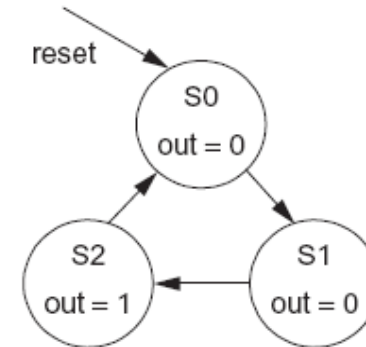
(a)



(b)

Moore Machine

```
module divideby3FSM(input logic clk,  
    input logic reset,  
    output logic y);  
  
    logic [1:0] state, nextstate;  
  
    // State Register  
    always_ff @(posedge clk)  
        if (reset) state <= 2'b00;  
        else state <= nextstate;  
  
    // Next State Logic  
    always_comb  
        case (state)  
            2'b00: nextstate = 2'b01;  
            2'b01: nextstate = 2'b10;  
            2'b10: nextstate = 2'b00;  
            default: nextstate = 2'b00;  
        endcase  
  
    // Output Logic  
    assign y = (state == 2'b00);  
endmodule
```



Moore Machine

```
module divideby3FSM(input logic clk,
                    input logic reset,
                    output logic y);

    typedef enum logic [1:0] {S0, S1, S2} statetype;

    statetype state, nextstate;

    // State Register
    always_ff @(posedge clk)
        if (reset) state <= S0;
        else state <= nextstate;

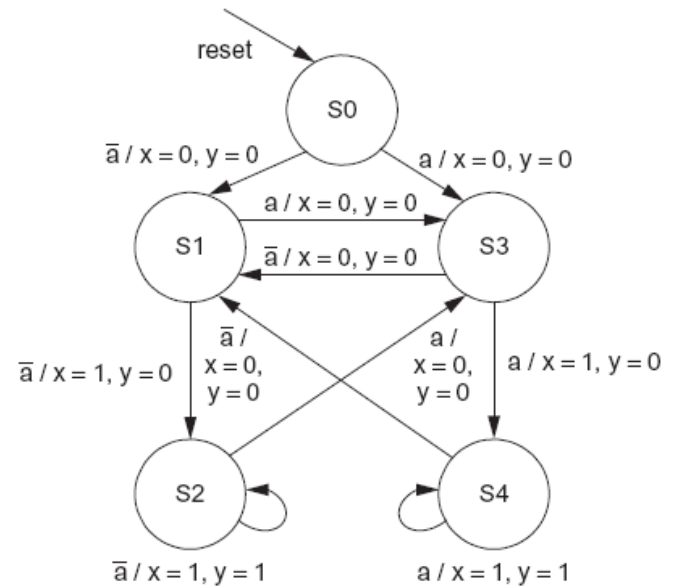
    // Next State Logic
    always_comb
        case (state)
            S0: nextstate = S1;
            S1: nextstate = S2;
            S2: nextstate = S0;
            default: nextstate = S0;
        endcase

    // Output Logic
    assign y = (state == S0);
endmodule
```

- typedef defines statetype to be a two-bit logic value with one of three possibilities: S0, S1, or S2
- state and nextstate are statetype signals
- Enumerated encodings default to numerical order: S0 = 00, S1 = 01, and S2 = 10
- The encodings can be explicitly set by the user
 - The following snippet encodes the states as 3-bit one-hot values:
 - typedef enum logic [2:0] {S0 = 3'b001, S1 = 3'b010, S2 = 3'b100} statetype;

Mealy Machine

- Identify 3 portions
 - *State register*
 - *Next state logic*
 - *Output logic*
- State register
 - Resets asynchronously to the initial state
 - Otherwise advances to the next state
- Next logic
 - Computes the next state as a function of the current state and inputs
- Output logic
 - Computes the output as a function of the current state and the inputs



Mealy Machine (cont'd)

```
module historyFSM(input logic clk,
  input logic reset,
  input logic a,
  output logic x, y);

  typedef enum logic [2:0]
    {S0, S1, S2, S3, S4} statetype;

  statetype state, nextstate;

  always_ff @(posedge clk)
    if (reset) state <= S0;
    else state <= nextstate;

  always_comb
    case (state)
      S0: if (a) nextstate = S3;
          else nextstate = S1;
      S1: if (a) nextstate = S3;
          else nextstate = S2;
      S2: if (a) nextstate = S3;
          else nextstate = S2;
      S3: if (a) nextstate = S4;
          else nextstate = S1;
      S4: if (a) nextstate = S4;
          else nextstate = S1;
      default: nextstate = S0;
    endcase

endmodule
```

```
//output logic

assign x = (state[1] & ~a) |
           (state[2] & a);

assign y = (state[1] & state[0] & ~a)
           | (state[2] & state[0] & a);

endmodule
```

Type Idiosyncracies

- Standard Verilog primarily uses two types: reg and wire
 - reg signal might or might not be associated with a register
 - Was a great source of confusion for those learning the language
- In standard Verilog
 - if a signal appears on the left-hand side of <= or = in an always block, it must be declared as reg
 - Otherwise, it should be declared as wire
 - Input and output ports default to the wire type unless their type is explicitly specified as reg

```
module flop(input clk,  
            input [3:0] d,  
            output reg [3:0] q);  
  
    always @(posedge clk)  
        q <= d;  
endmodule
```

- q is set in always block =>
- declared as reg type



Type Idiosyncracies (cont'd)

- SystemVerilog introduced the logic type
 - logic is a synonym for reg
- Logic can be used outside always blocks where a wire traditionally would be required
 - Nearly all SystemVerilog signals can be logic
- The exception are signals with multiple drivers (e.g., a tristate bus)
 - Must be declared as a net
 - Allows SystemVerilog to generate an error message rather than an x value when a logic signal is accidentally connected to multiple drivers
- The most common type of net is called a wire or tri
 - Synonymous, but wire is conventionally used when a single driver is present and tri is used when multiple drivers are present
 - wire is obsolete in SystemVerilog => use logic
- Types and net resolution (see table)

Net Type	No Driver	Conflicting Drivers
tri	z	x
triand	z	0 if any are 0
trior	z	1 if any are 1
trireg	previous value	x
tri0	0	x
tri1	1	x



Parameterized Modules

2-way mux, default width 8

```
module mux2
  #(parameter width = 8)
  (input logic [width-1:0] d0, d1,
   input logic s,
   output logic [width-1:0] y);

  assign y = s ? d1 : d0;
endmodule
```

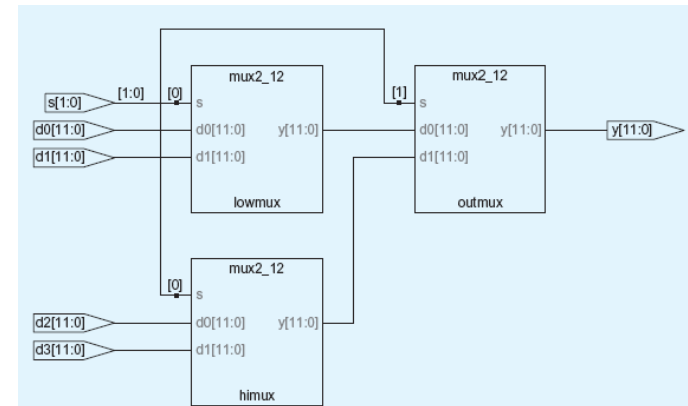
```
// use default widths
module mux4_8(input logic [7:0] d0, d1, d2, d3,
  input logic [1:0] s,
  output logic [7:0] y);
  logic [7:0] low, hi;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[0], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

```
// 12-bit mux; override default using #()
module mux4_12(input logic [11:0] d0, d1, d2, d3,
  input logic [1:0] s,
  output logic [11:0] y);

  logic [11:0] low, hi;

  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[0], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```





Parameterized Modules

```
module decoder #(parameter N = 3)
  (input logic [N-1:0] a,
   output logic [2**N-1:0] y);

  always_comb
  begin
    y = 0;
    y[a] = 1;
  end
endmodule
```



Generate Statement

```
module andN
  #(parameter width = 8)
  (input logic [width-1:0] a,
   output logic y);

  genvar i;
  logic [width-1:1] x;
  generate
    for (i=1; i<width; i=i+1) begin:forloop
      if (i == 1)
        assign x[1] = a[0] & a[1];
      else
        assign x[i] = a[i] & x[i-1];
      end
    endgenerate
    assign y = x[width-1];
endmodule
```

- Generate statements produce a variable amount of hardware depending on the value of a parameter
- Generate supports for loops and if statements to determine how many of what types of hardware to produce
- Example: N -input AND function from a cascade of 2-input ANDs

Memories

```
// separate read and write busses
module ram #(parameter N = 6, M = 32)
  (input logic clk,
   input logic we,
   input logic [N-1:0] adr,
   input logic [M-1:0] din,
   output logic [M-1:0] dout);

  logic [M-1:0] mem[2**N-1:0];
  always @(posedge clk)
    if (we) mem[adr] <= din;

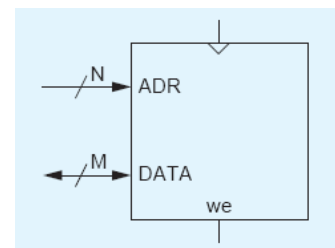
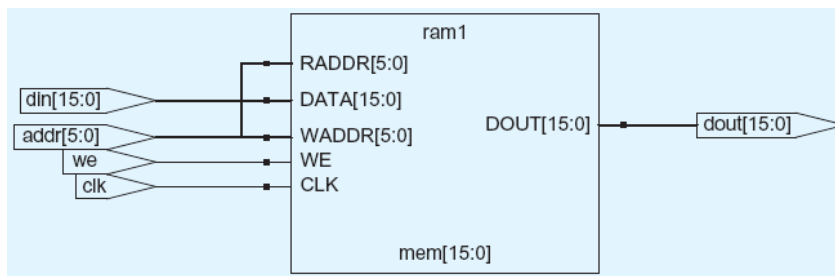
  assign dout = mem[adr];
endmodule
```

```
// bidirectional bus
module ram #(parameter N = 6, M = 32)
  (input logic clk,
   input logic we,
   input logic [N-1:0] adr,
   inout tri [M-1:0] data);

  logic [M-1:0] mem[2**N-1:0];

  always @(posedge clk)
    if (we) mem[adr] <= data;

  assign data = we ? 'z : mem[adr];
endmodule
```



Multi-ported Register Files, ROMs

```
module ram3port #(parameter N = 6, M = 32)
    (input logic clk,
     input logic we3,
     input logic [N-1:0] a1, a2, a3,
     output logic [M-1:0] d1, d2,
     input logic [M-1:0] d3);

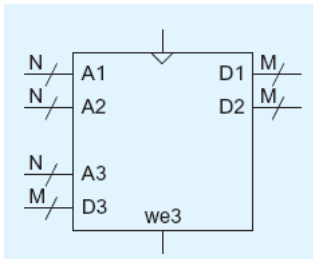
    logic [M-1:0] mem[2**N-1:0];

    always @(posedge clk)
        if (we3) mem[a3] <= d3;

    assign d1 = mem[a1];
    assign d2 = mem[a2];
endmodule
```

```
module rom(input logic [1:0] adr,
           output logic [2:0] dout);

    always_comb
        case(adr)
            2'b00: dout = 3'b011;
            2'b01: dout = 3'b110;
            2'b10: dout = 3'b100;
            2'b11: dout = 3'b010;
        endcase
endmodule
```



Testbench

- Testbench is an HDL module used to test another module, called the *device under test* (DUT)
- Contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced
 - Input and desired output patterns are called *test vectors*.
- Initial statement executes the statements in its body at the start of simulation
- Initial statements should only be used in testbenches for simulation, not in modules intended to be synthesized into actual hardware

```
module testbench1();
  logic a, b, c;
  logic y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);
  // apply inputs one at a time
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
    a = 1; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
  end
endmodule
```

Self-Checking Testbench

- Checking for correct outputs by hand is tedious and error-prone
- Determining the correct outputs is much easier when the design is fresh in your mind
- => A much better approach is to write a self-checking testbench
- SystemVerilog assert statement checks if a specified condition is true
 - If it is not, it executes the else statement.
 - `$error system task` in the else statement prints an error message describing the assertion failure
- In SystemVerilog, comparison using `==` or `!=` spuriously indicates equality if one of the operands is x or z.
- The `===` and `!==` operators must be used instead for testbenches because they work correctly with x and z

```
module testbench2();
    logic a, b, c;
    logic y;
    // instantiate device under test
    sillyfunction dut(a, b, c, y);
    // apply inputs one at a time
    // checking results
    initial begin
        a = 0; b = 0; c = 0; #10;
        assert (y === 1) else $error("000 failed.");
        c = 1; #10;
        assert (y === 0) else $error("001 failed.");
        b = 1; c = 0; #10;
        assert (y === 0) else $error("010 failed.");
        c = 1; #10;
        assert (y === 0) else $error("011 failed.");
        a = 1; b = 0; c = 0; #10;
        assert (y === 1) else $error("100 failed.");
        c = 1; #10;
        assert (y === 1) else $error("101 failed.");
        b = 1; c = 0; #10;
        assert (y === 0) else $error("110 failed.");
        c = 1; #10;
        assert (y === 0) else $error("111 failed.");
    end
endmodule
```



Testbench with Test Vector File

```
module testbench3();
  logic clk, reset;
  logic a, b, c, yexpected;
  logic y;
  logic [31:0] vectornum, errors;
  logic [3:0] testvectors[10000:0];

  // instantiate device under test
  sillyfunction dut(a, b, c, y);
  // generate clock
  always
  begin
    clk = 1; #5; clk = 0; #5;
  end
  // at start of test, load vectors
  // and pulse reset
  initial
  begin
    $readmemb("example.tv", testvectors);
    vectornum = 0; errors = 0;
    reset = 1; #27; reset = 0;
  end
end
```

example.tv

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```



Testbench with Test Vector File (cont'd)

```
// apply test vectors on rising edge of clk
always @(posedge clk)
begin
    // wait for 1 time unit before assignment
    #1; {a, b, c, yexpected} =
    testvectors[vectornum];
end

// check results on falling edge of clk
always @(negedge clk)
if (~reset) begin // skip during reset
    if (y !== yexpected) begin
        $display("Error: inputs = %b", {a, b, c});
        $display(" outputs = %b (%b expected)",
            y, yexpected);
        errors = errors + 1;
    end
    vectornum = vectornum + 1;
    if (testvectors[vectornum] === 'bx) begin
        $display("%d tests completed with %d
            errors", vectornum, errors);
        $finish;
    end
end
end
endmodule
```

- \$readmemb reads a file of binary numbers into an array
- \$readmemh reads a file of hexadecimal numbers into an array
- #1; waits one time unit after the rising edge of the clock (to avoid any confusion of clock and data changing simultaneously)
- \$display is a system task to print in the simulator window
- \$finish terminates the simulation
- Note how we terminate simulation after 8 test vectors