

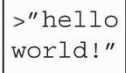


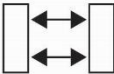
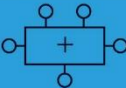

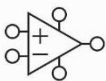


**Digital Design &
Computer Architecture**

Sarah Harris & David Harris

**Chapter 4:
Hardware Description
Languages**

Chapter 4 :: Topics

- Introduction
- Combinational Logic
- Delays
- Sequential Logic
- Combinational Logic w/ Always
- Blocking & Nonblocking Assignments
- Finite State Machines
- Parameterized Modules
- Testbenches

Application Software	
Operating Systems	
Architecture	
Micro-architecture	
Logic	
Digital Circuits	
Analog Circuits	
Devices	
Physics	

Chapter 4: Hardware Description Languages

Introduction

Introduction

- **Hardware description language (HDL):**
 - Specifies logic function only
 - Computer-aided design (CAD) tool produces or *synthesizes* the optimized gates
- Most commercial designs built using HDLs
- Two leading HDLs:
 - **SystemVerilog**
 - Developed in 1984 by Gateway Design Automation
 - IEEE standard (1364) in 1995
 - Extended in 2005 (IEEE STD 1800-2009)
 - **VHDL 2008**
 - Developed in 1981 by the Department of Defense
 - IEEE standard (1076) in 1987
 - Updated in 2008 (IEEE STD 1076-2008)

HDL to Gates

- **Simulation**

- Inputs applied to circuit
- Outputs checked for correctness
- Millions of dollars saved by debugging in simulation instead of hardware

- **Synthesis**

- Transforms HDL code into a *netlist* describing the hardware (i.e., a list of gates and the wires connecting them)

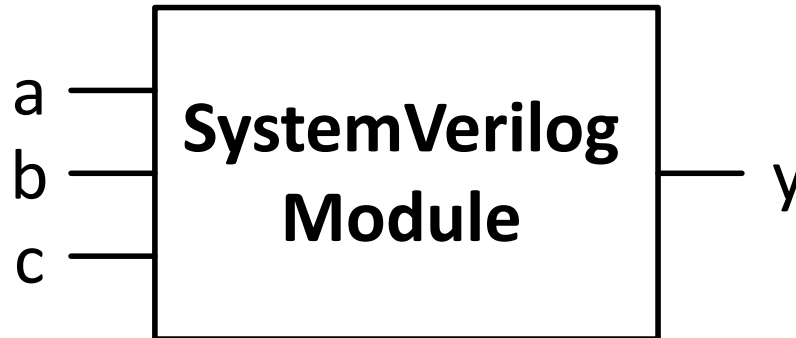
HDL: *Hardware* Description Language

IMPORTANT:

When using an HDL, think of the **hardware** the HDL should produce, then write the appropriate idiom that implies that hardware.

Beware of treating HDL like software and coding without thinking of the hardware.

SystemVerilog Modules



Two types of Modules:

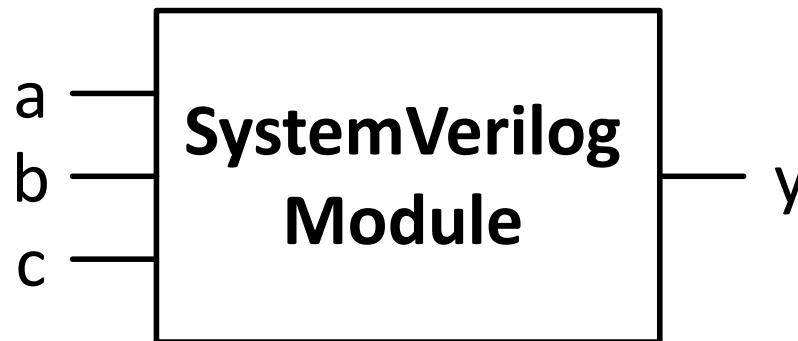
- **Behavioral:** describe what a module does
- **Structural:** describe how it is built from simpler modules

Module Declaration

SystemVerilog:

```
module example(input  logic a, b, c,  
               output logic y);  
    // module body goes here  
endmodule
```

- **module/endmodule:** required to begin/end module
- **example:** name of the module



Behavioral SystemVerilog

SystemVerilog:

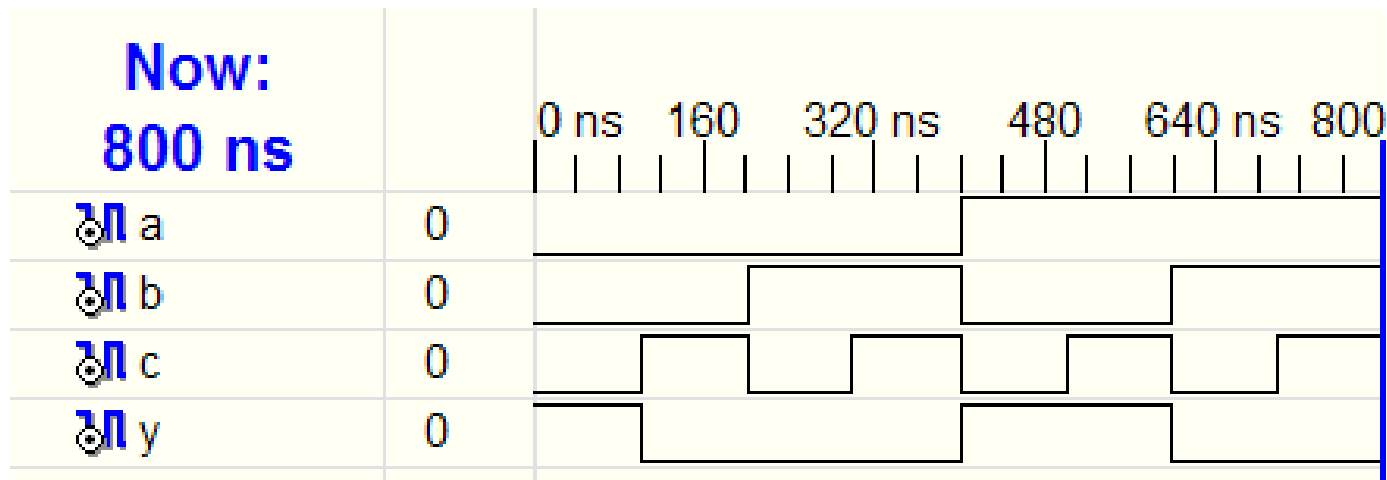
```
module example(input  logic a, b, c,
               output logic y);
    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;
endmodule
```

- **module/endmodule:** required to begin/end module
- **example:** name of the module
- **Operators:**
 - ~: NOT
 - &: AND
 - |: OR

HDL Simulation

SystemVerilog:

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

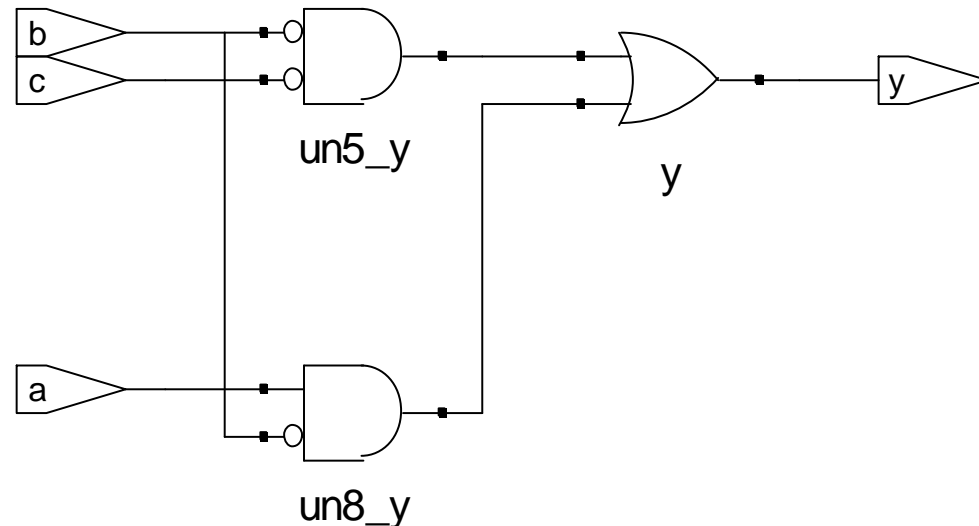


HDL Synthesis

SystemVerilog:

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

Synthesis:



SystemVerilog Syntax

- **Case sensitive**
 - **Example:** `reset` and `Reset` are not the same signal.
- **No names that start with numbers**
 - **Example:** `2mux` is an **invalid** name
- **Whitespace ignored**
- **Comments:**
 - `//` single line comment
 - `/*` multiline
comment `*/`

Structural SystemVerilog

```
module and3(input  logic a, b, c,  
            output logic y);  
    assign y = a & b & c;  
endmodule
```

```
module inv(input  logic a,  
           output logic y);  
    assign y = ~a;  
endmodule
```

```
module nand3(input  logic a, b, c  
            output logic y);  
    logic n1;                // internal signal  
  
    and3 andgate(a, b, c, n1); // instance of and3  
    inv  inverter(n1, y);      // instance of inv  
endmodule
```

Chapter 4: Hardware Description Languages

Combinational Logic

Bitwise Operators

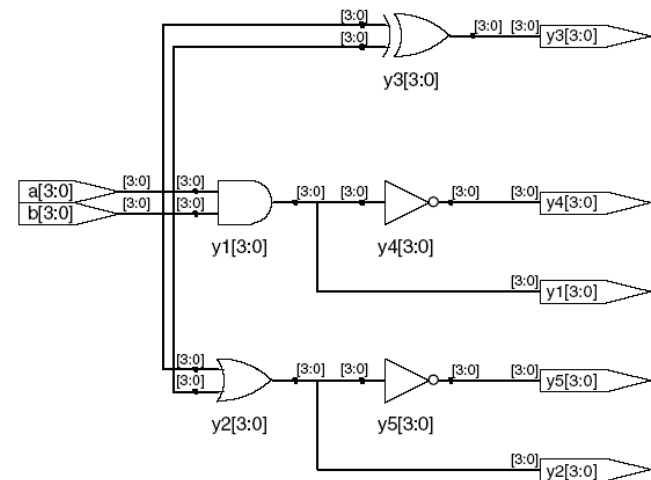
SystemVerilog:

```
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
```

// single line comment

/*...*/ multiline comment

Synthesis:

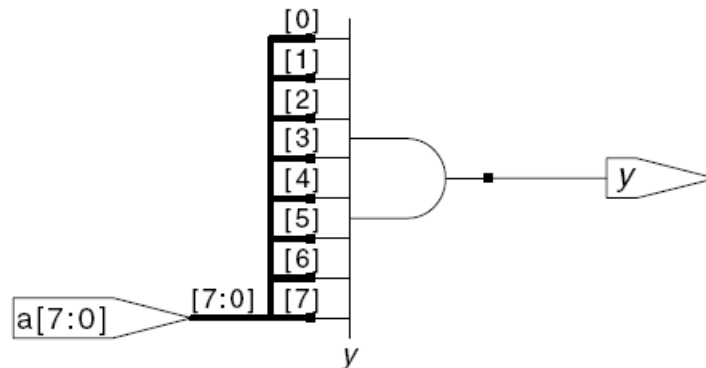


Reduction Operators

SystemVerilog:

```
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
```

Synthesis:

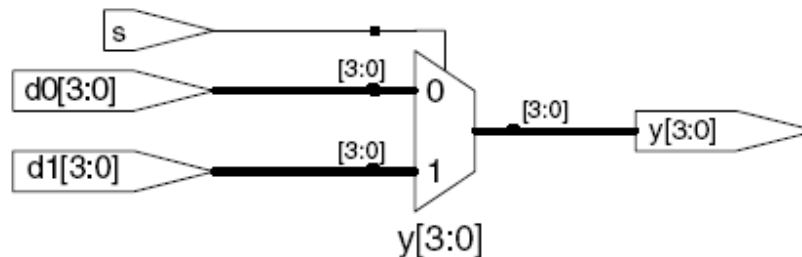


Conditional Assignment

SystemVerilog:

```
module mux2(input  logic [3:0] d0, d1,  
           input  logic      s,  
           output logic [3:0] y);  
    assign y = s ? d1 : d0;  
endmodule
```

Synthesis:



? : is also called a *ternary operator* because it operates on 3 inputs: s, d1, and d0.

Internal Variables

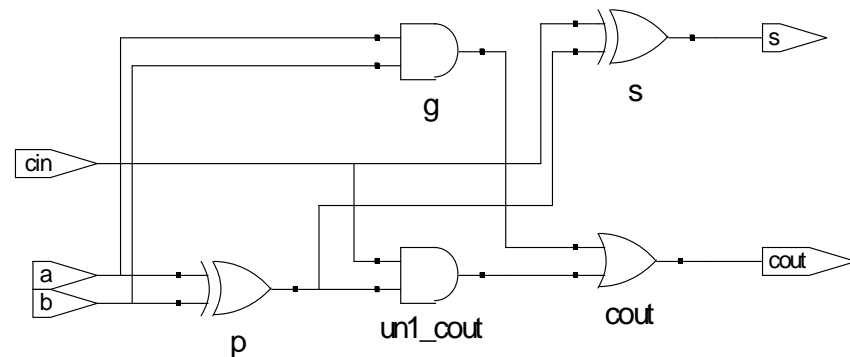
SystemVerilog:

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

    assign p = a ^ b;
    assign g = a & b;

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

Synthesis:



Precedence

Highest

~	NOT
*, /, %	mult, div, mod
+, -	add, sub
<<, >>	shift
<<<, >>>	arithmetic shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
, ~	OR, NOR
?:	ternary operator

Lowest

Numbers

Format: N'Bvalue

N = number of bits, **B** = base

N'B is optional but recommended (default is decimal)

Number	# Bits	Base	Decimal Equivalent	Stored
3'b101	3	binary	5	101
'b11	unsized	binary	3	00...0011
8'b11	8	binary	3	00000011
8'b1010_1011	8	binary	171	10101011
3'd6	3	decimal	6	110
6'o42	6	octal	34	100010
8'hAB	8	hexadecimal	171	10101011
42	Unsized	decimal	42	00...0101010

Bit Manipulations: Example 1

SystemVerilog:

```
assign y = {a[2:1], {3{b[0]}}, a[0], 6'b100_010};
```

If y is a 12-bit signal, the above statement produces:

```
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0
```

Underscores (`_`) are used for formatting only to make it easier to read. SystemVerilog ignores them.

Bit Manipulations: Example 2

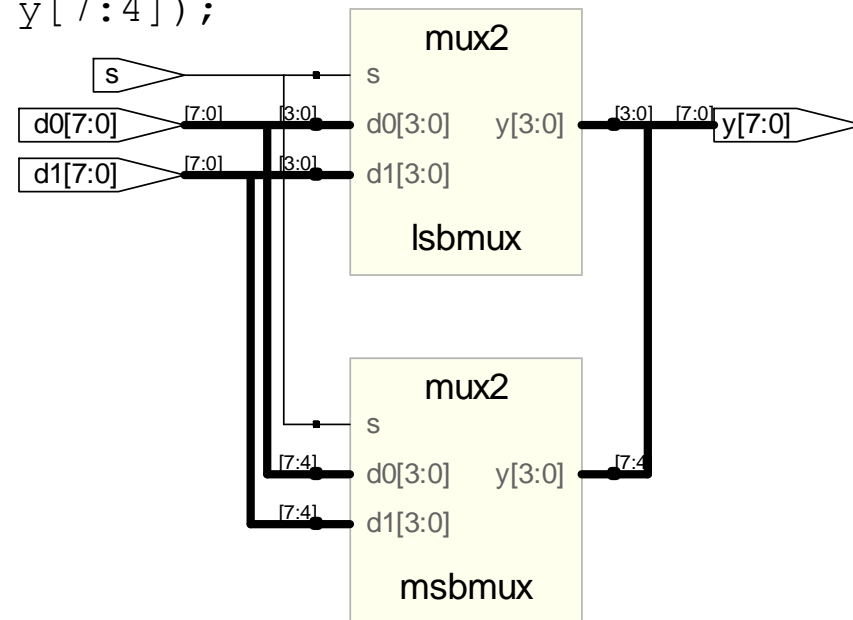
SystemVerilog:

```
module mux2_8(input  logic [7:0] d0, d1,  
             input  logic      s,  
             output logic [7:0] y);
```

```
    mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);  
    mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
```

```
endmodule
```

Synthesis:

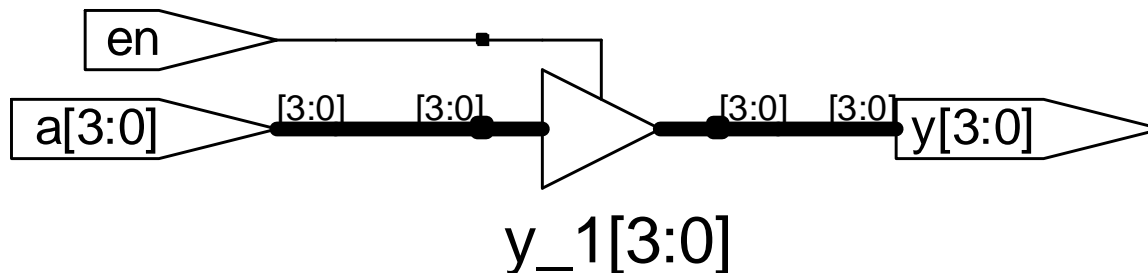


Z: Floating Output

SystemVerilog:

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

Synthesis:



Chapter 4: Hardware Description Languages

Delays

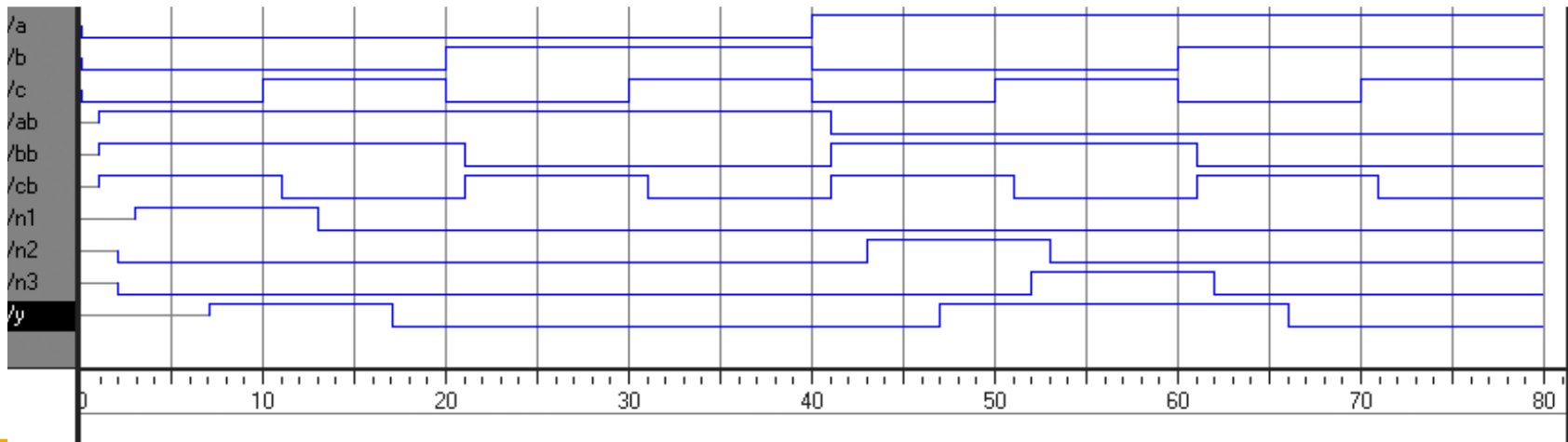
Delays

SystemVerilog:

```
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
```

Delays are for **simulation only!** They do not determine the delay of your hardware.

Simulation

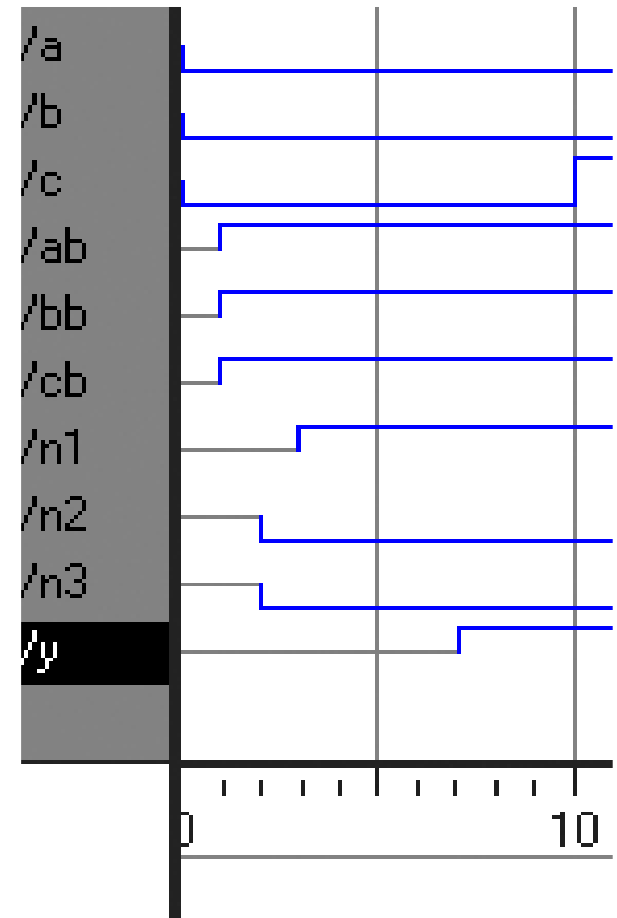


Delays

SystemVerilog:

```
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
```

Simulation

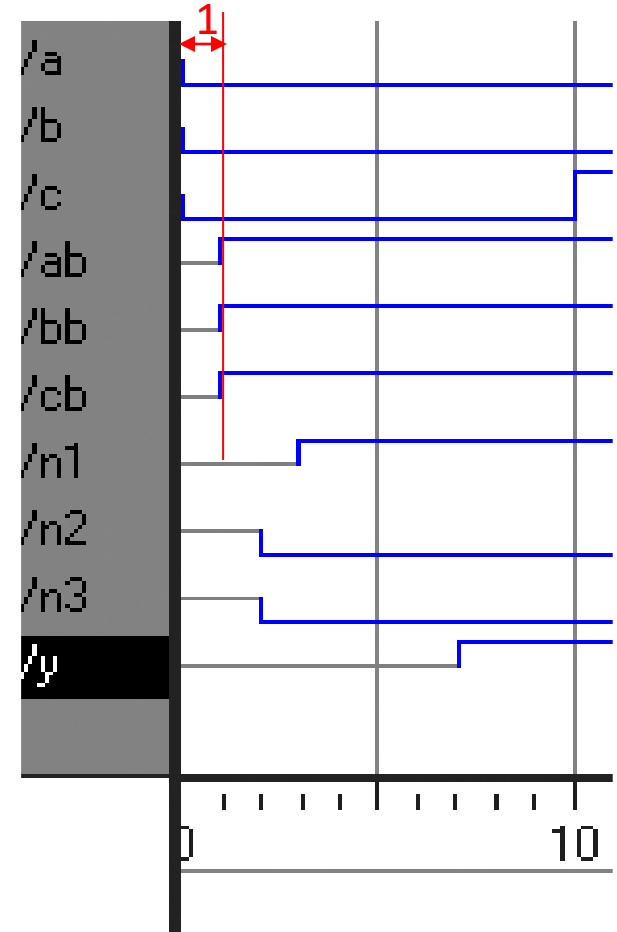


Delays

SystemVerilog:

```
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
```

Simulation

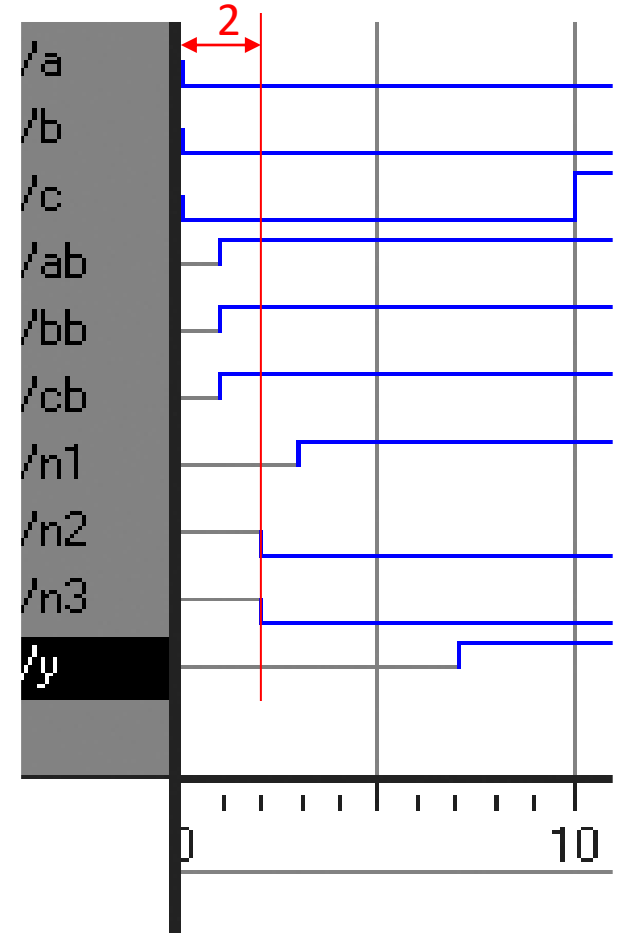


Delays

SystemVerilog:

```
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
```

Simulation

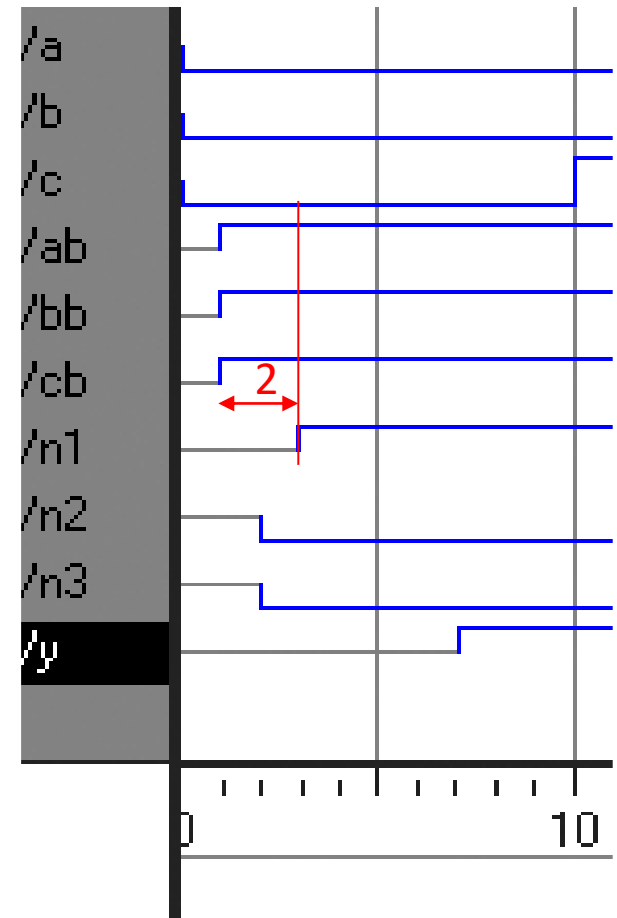


Delays

SystemVerilog:

```
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
```

Simulation

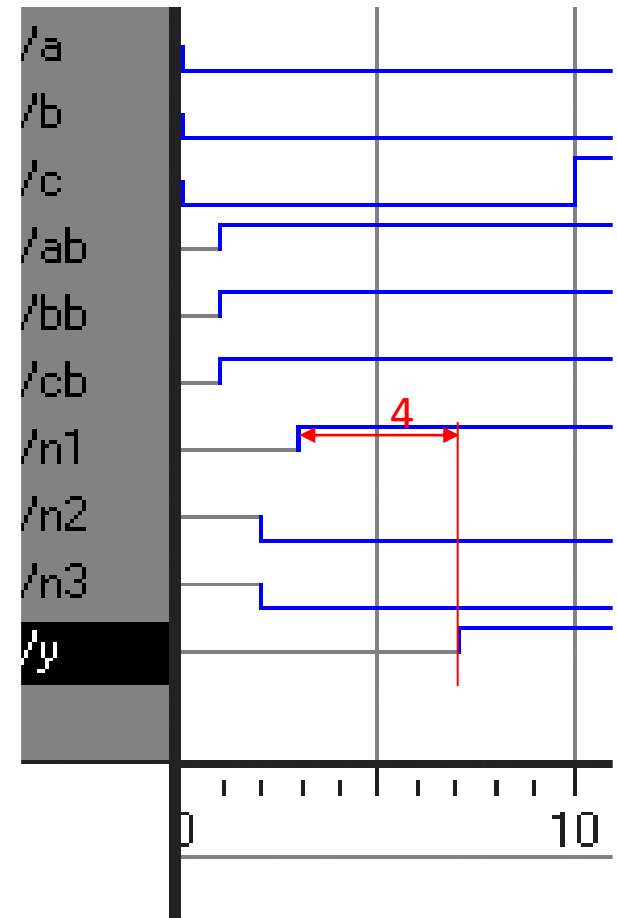


Delays

SystemVerilog:

```
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
```

Simulation



Chapter 4: Hardware Description Languages

Sequential Logic

Sequential Logic

- SystemVerilog uses **idioms** to describe latches, flip-flops and FSMs
- Other coding styles may simulate correctly but produce incorrect hardware

always Statement

General Structure:

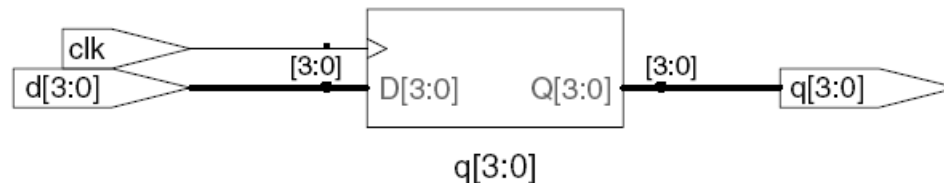
```
always @(sensitivity list)
    statement;
```

Whenever the event in `sensitivity list` occurs, `statement` is executed

D Flip-Flop

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

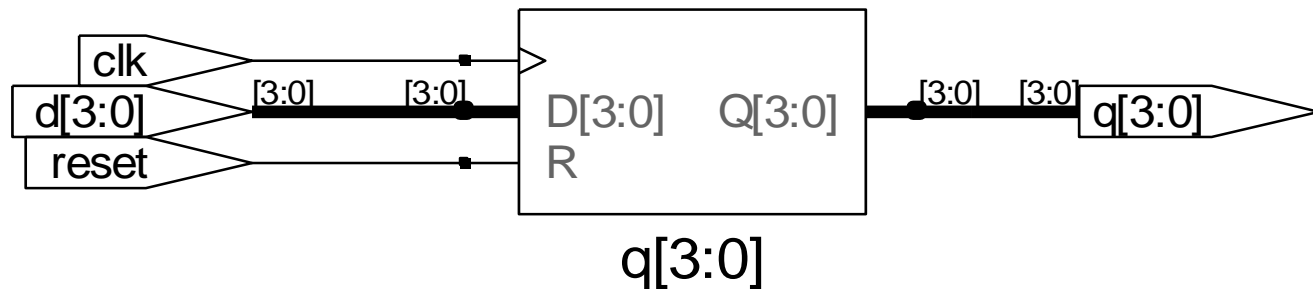
Synthesis:



Resettable D Flip-Flop

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

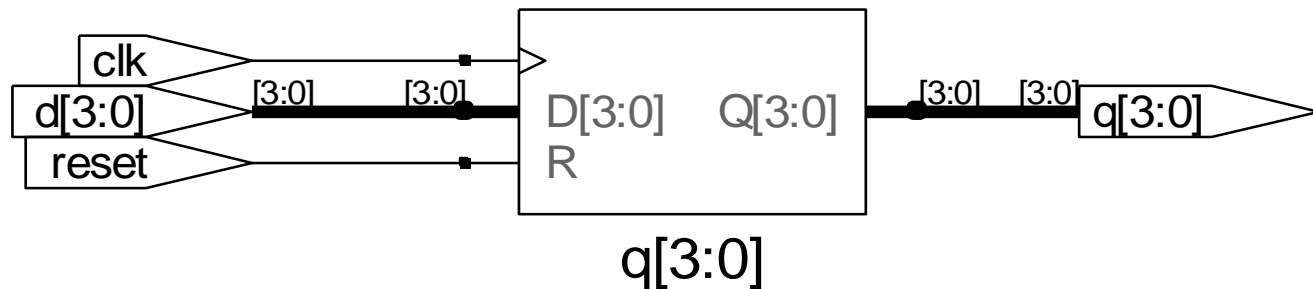
Synthesis:



Resettable D Flip-Flop

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

Synthesis:



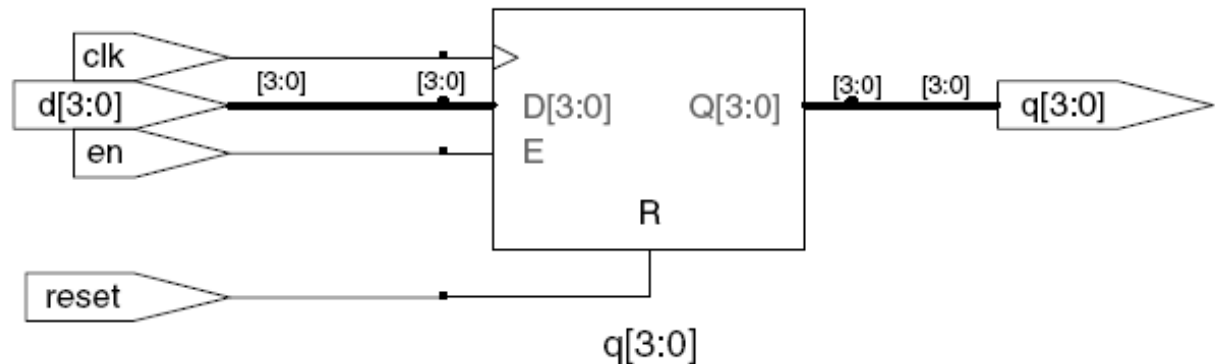
D Flip-Flop with Enable

```
module flopren(input  logic      clk,  
              input  logic      reset,  
              input  logic      en,  
              input  logic [3:0] d,  
              output logic [3:0] q);
```

```
    always_ff @(posedge clk, posedge reset)  
        if      (reset) q <= 4'b0;  
        else if (en)    q <= d;
```

```
endmodule
```

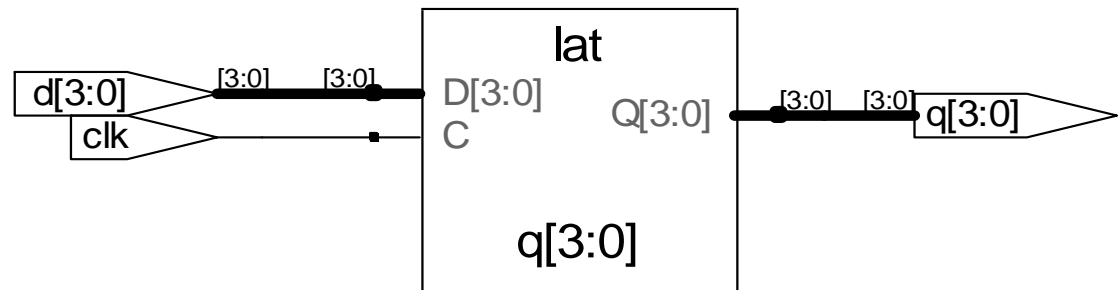
Synthesis:



Latch

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

Synthesis:



Warning: We don't use latches in this text. But you might write code that inadvertently implies a latch. Check synthesized hardware – if it has latches in it that you didn't intend to create, there's an **error**.

Review

General Structure:

```
always @(sensitivity list)
    statement;
```

- **Flip-flop:** `always_ff`
- **Latch:** `always_latch` **(don't use)**

Chapter 4: Hardware Description Languages

Combinational Logic using `always`

if/else and case/casez

Statements that must be inside always statements:

- if/else
- case, casez

Combinational Logic using `always`

```
// combinational logic using an always statement
module gates(input logic [3:0] a, b,
             output logic [3:0] y1, y2, y3, y4, y5);
    always_comb // need begin/end because there is
    begin      // more than one statement in always
        y1 = a & b; // AND
        y2 = a | b; // OR
        y3 = a ^ b; // XOR
        y4 = ~(a & b); // NAND
        y5 = ~(a | b); // NOR
    end
endmodule
```

This hardware could be described with **assign statements using fewer lines** of code, so it's better to use **assign** statements in this case.

Combinational Logic using case

```
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_0011;
      default: segments = 7'b000_0000; // required
    endcase
endmodule
```

Combinational Logic using `case`

- `case` statement implies combinational logic **only if** all possible input combinations described
- Remember to use **`default`** statement

Combinational Logic using casez

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

```
  always_comb
```

```
    casez (a)
```

```
      4'b1???: y = 4'b1000; // ? = don't care
```

```
      4'b01??: y = 4'b0100;
```

```
      4'b001?: y = 4'b0010;
```

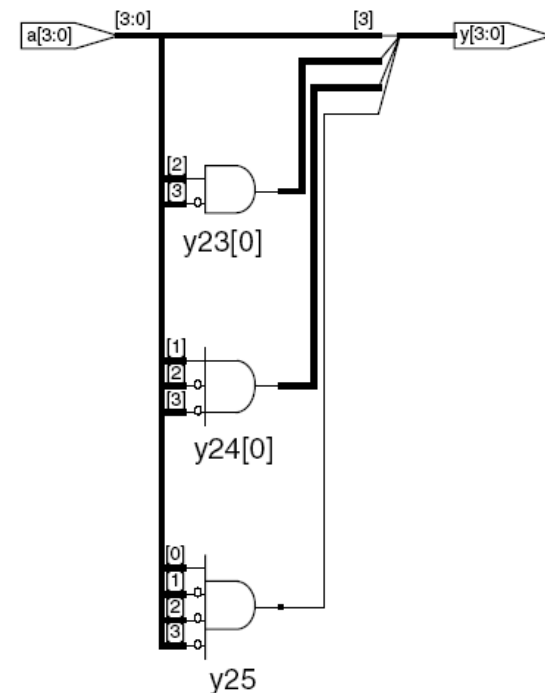
```
      4'b0001: y = 4'b0001;
```

```
      default: y = 4'b0000;
```

```
    endcase
```

```
endmodule
```

Synthesis:



Chapter 4: Hardware Description Languages

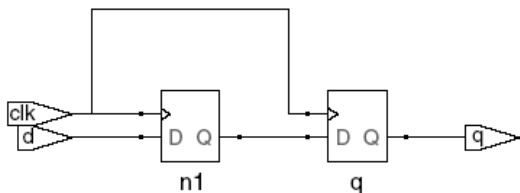
Blocking and Nonblocking Assignments

Blocking vs. Nonblocking Assignment

- `<=` is **nonblocking** assignment
 - Occurs simultaneously with others
- `=` is **blocking** assignment
 - Occurs in order it appears in file

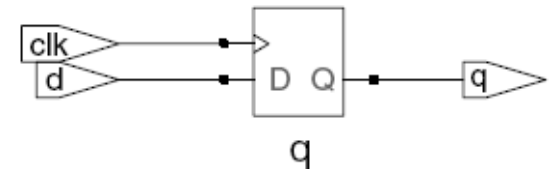
```
// Good synchronizer using
// nonblocking assignments
module syncgood(input logic clk,
                input logic d,
                output logic q);

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



```
// Bad synchronizer 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
```



Rules for Signal Assignment

- **Synchronous sequential logic:** use `always_ff @ (posedge clk)` and nonblocking assignments (`<=`)

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

- **Simple combinational logic:** use continuous assignments (`assign...`)

```
assign y = a & b;
```

- **More complicated combinational logic:** use `always_comb` and blocking assignments (`=`)
- Assign a signal in **only one** `always` statement or continuous assignment statement.

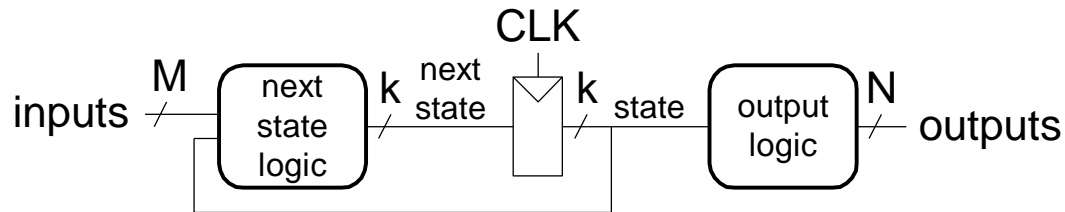
Chapter 4: Hardware Description Languages

Finite State Machines

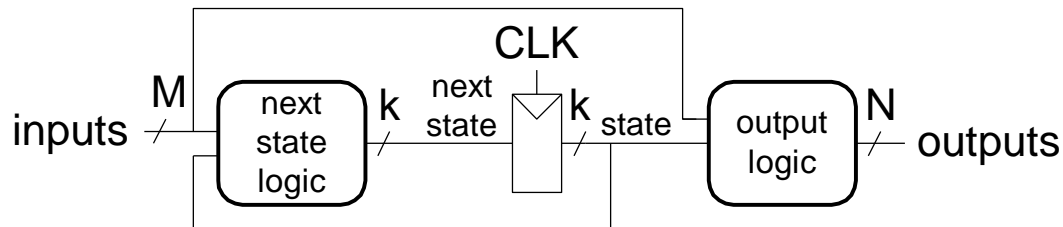
Finite State Machines

- **Three blocks:**
 - next state logic
 - state register
 - output logic

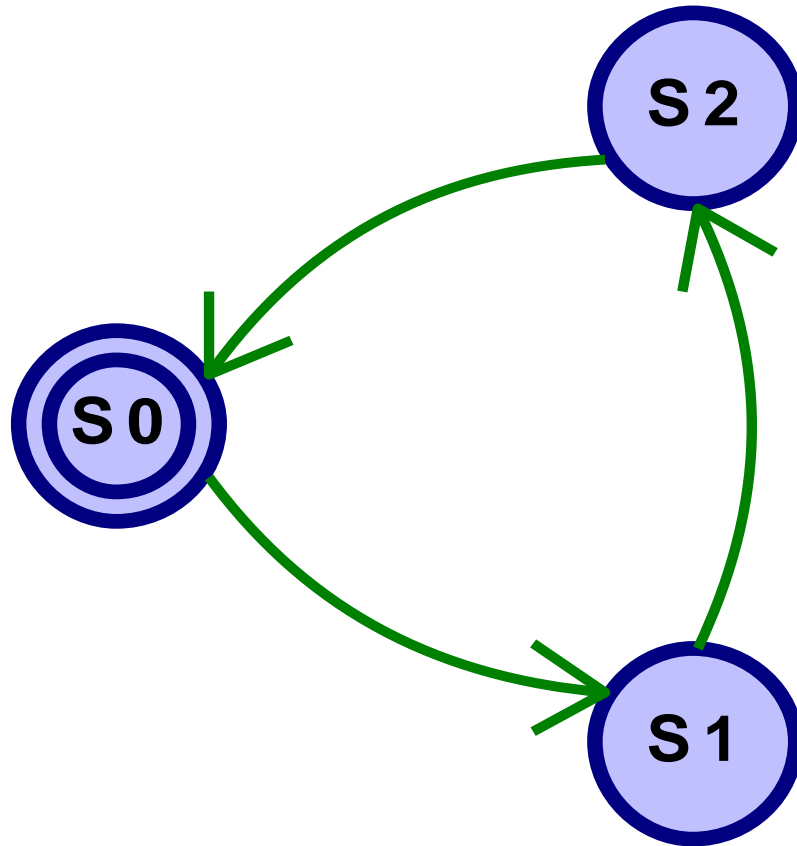
Moore FSM



Mealy FSM



FSM Example 1: Divide by 3



The double circle indicates the reset state

Divide by 3 FSM in SystemVerilog

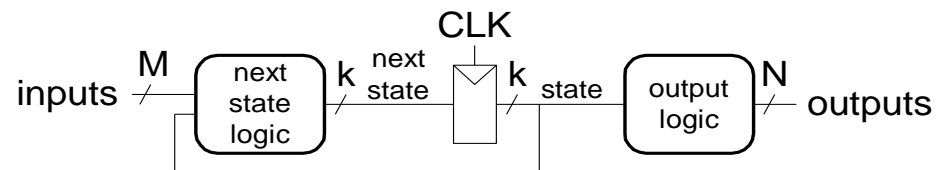
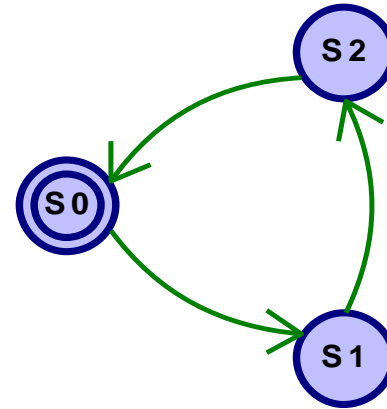
```
module divideby3FSM(input  logic clk,
                   input  logic reset,
                   output logic q);

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

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

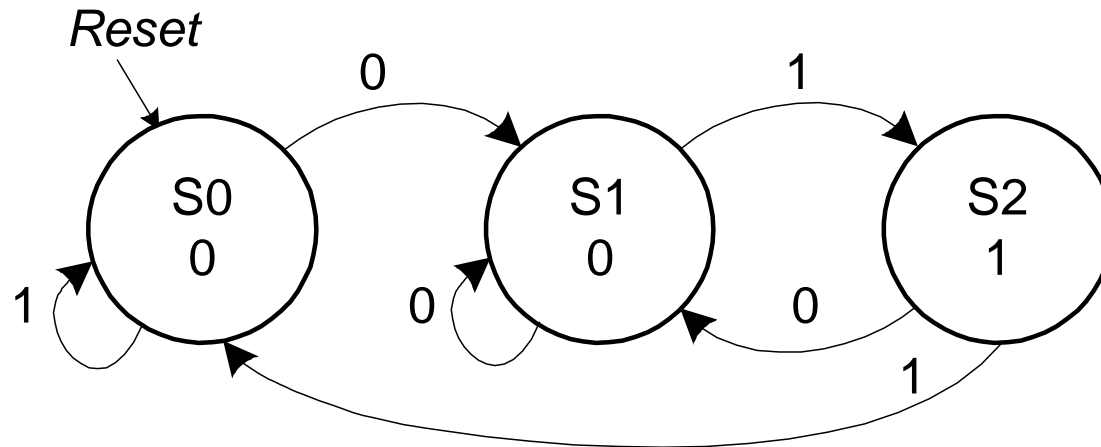
    // next state logic
    always_comb
        case (state)
            S0:      nextstate = S1;
            S1:      nextstate = S2;
            S2:      nextstate = S0;
            default: nextstate = S0;
        endcase

    // output logic
    assign q = (state == S0);
endmodule
```



FSM Example 2: Sequence Detector

Moore FSM



Sequence Detector FSM: Moore

```
module seqDetectMoore(input logic clk, reset, a,
                    output logic smile);

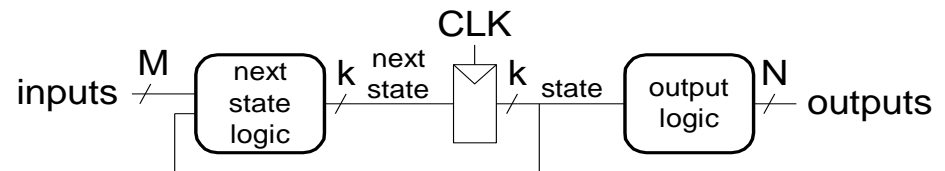
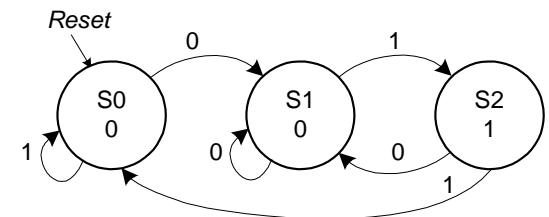
    typedef enum logic [1:0] {S0, S1, S2} statetype;
    statetype state, nextstate;

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

    // next state logic
    always_comb
        case (state)
            S0: if (a) nextstate = S0;
                else nextstate = S1;
            S1: if (a) nextstate = S2;
                else nextstate = S1;
            S2: if (a) nextstate = S0;
                else nextstate = S1;
            default: nextstate = S0;
        endcase

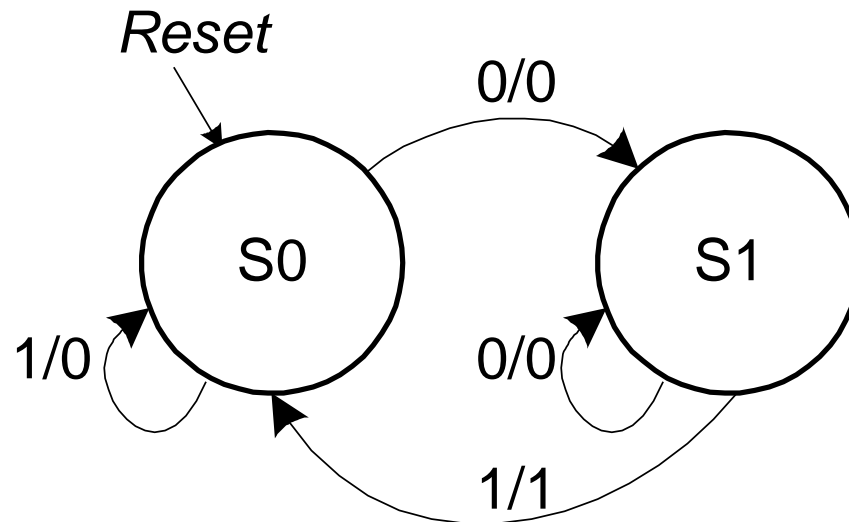
    // output logic
    assign smile = (state == S2);
endmodule
```

Moore FSM



FSM Example 3: Sequence Detector

Mealy FSM



Sequence Detector FSM: Mealy

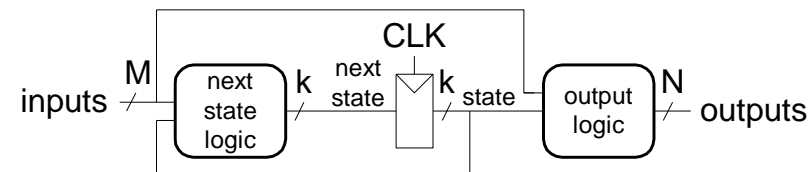
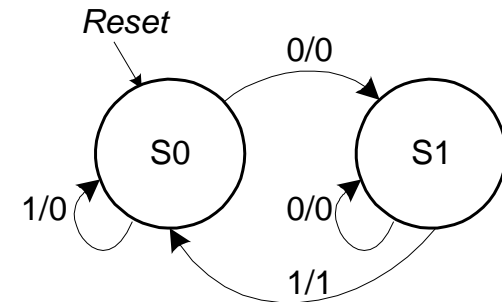
```
module seqDetectMealy(input logic clk, reset, a,
                    output logic smile);

    typedef enum logic {S0, S1} statetype;
    statetype state, nextstate;

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

    // next state and output logic
    always_comb begin
        smile = 1'b0;
        case (state)
            S0:    if (a)    nextstate = S0;
                   else    nextstate = S1;
            S1:    if (a) begin
                       nextstate = S0;
                       smile = 1'b1;
                   end
                   else    nextstate = S1;
            default:    nextstate = S0;
        endcase
    end
endmodule
```

Mealy FSM



Chapter 4: Hardware Description Languages

Parameterized Modules

Parameterized Module

2:1 mux:

```
module mux2
    #(parameter width = 8) // name and default value
    (input logic [width-1:0] d0, d1,
     input logic s,
     output logic [width-1:0] y);
    assign y = s ? d1 : d0;
endmodule
```

Instance with 8-bit bus width (uses default):

```
mux2 myMux(d0, d1, s, out);
```

Instance with 12-bit bus width:

```
mux2 #(12) lowmux(d0, d1, s, out);
```

Chapter 4: Hardware Description Languages

Testbenches

Testbenches

- HDL that **tests another module**: *device under test* (dut)
- **Not synthesizable**
- **Types:**
 - Simple
 - Self-checking
 - Self-checking with testvectors

Testbenches

- Write SystemVerilog code to implement the following function in hardware. Name the module `sillyfunction`.

$$y = \overline{bc} + a\overline{b}$$

Testbench 1: Simple Testbench

```
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
```

Testbench 2: Self-Checking Testbench

```
module testbench2();
  logic a, b, c;
  logic y;
  sillyfunction dut(a, b, c, y); // instantiate dut
  initial begin // apply inputs, check results one at a time
    a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display("000 failed.");
    c = 1; #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("010 failed.");
    c = 1; #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1; #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("110 failed.");
    c = 1; #10;
    if (y !== 0) $display("111 failed.");
  end
endmodule
```

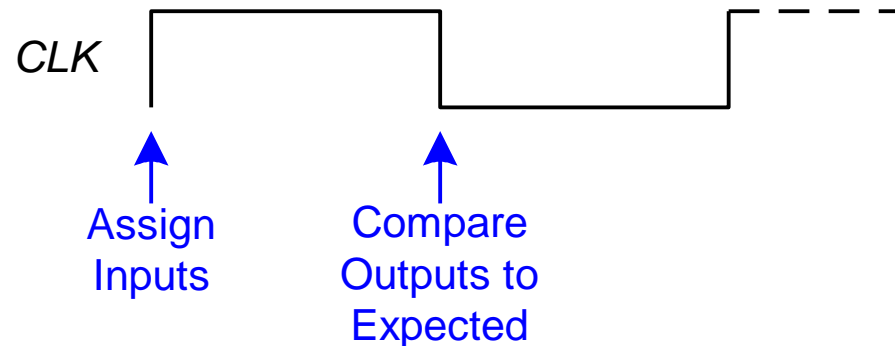
Testbench 3: Testbench w/ Testvectors

- Testvector file: inputs and expected outputs
- Testbench:
 1. Generate clock for assigning inputs, reading outputs
 2. Read testvectors file into array
 3. Assign inputs, expected outputs
 4. Compare outputs with expected outputs and report errors

Testbench 3: Testbench w/ Testvectors

- **Testbench clock:**

- Assign **inputs** (on rising edge).
- Compare **outputs** with expected outputs (on falling edge).



- Testbench clock **also used as clock** for synchronous sequential circuits

Testbench 3: Testvectors File

- **File:** `example.txt`
- **contains vectors of `abc_yexpected`**

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

1. Generate Clock

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

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

    // generate clock
    always        // no sensitivity list, so it always executes
    begin
        clk = 1; #5; clk = 0; #5;
    end
```

2. Read Testvectors into Array

```
// at start of test, load vectors and pulse reset
```

```
initial
begin
    $readmemb("example.tv", testvectors);
    vectornum = 0; errors = 0;
    reset = 1; #22; reset = 0;
end
```

```
// Note: $readmemb reads testvector files written in
// hexadecimal
```

3. Assign Inputs and Expected Outputs

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

4. Compare with Expected Outputs

```
// 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
  end
```

```
// Note: to print in hexadecimal, use %h. For example,
//      $display("Error: inputs = %h", {a, b, c});
```

This bit width needs to be the same as vector size!

```
// increment array index and read next testvector
vectornum = vectornum + 1;
if (testvectors[vectornum] === 4'bx) begin
  $display("%d tests completed with %d errors",
          vectornum, errors);
  $stop;
end
end
endmodule
```

```
// === and !== can compare values that are 1, 0, x, or z.
```

About these Notes

Digital Design and Computer Architecture Lecture Notes

© 2021 Sarah Harris and David Harris

These notes may be used and modified for educational and/or non-commercial purposes so long as the source is attributed.