

B38DF Computer Architecture and Embedded Systems

A quick intro to Verilog HDL

Alexander Belyaev

Heriot-Watt University

School of Engineering & Physical Sciences

Electrical, Electronic and Computer Engineering

E-mail: a.belyaev@hw.ac.uk

Office: EM2.29

Based on the slides accompanying *Digital Design and Computer Architecture* of Harris & Harris

Hardware Description Language (HDL)

- A Hardware Description Language (HDL): allows a designer to specify logic function only. Then a computer-aided design (CAD) tool produces or *synthesizes* optimized gates.
- Most commercial designs built using HDLs
- Two leading HDLs:
 - **Verilog**
 - developed in 1984 by Gateway Design Automation
 - became an IEEE standard in 1995
 - **VHDL**
 - Developed in 1981 by U.S. Department of Defense
 - Became an IEEE standard in 1987

HDL to Gates

- **Simulation**

- Input values are applied to the circuit
- Outputs checked for correctness
- Millions of dollars/pounds 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)

IMPORTANT:

When describing circuits using an HDL, it's critical to think of the **hardware** the code should produce.

Verilog Modules



Two types of Modules:

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

Behavioral Verilog Example

Verilog:

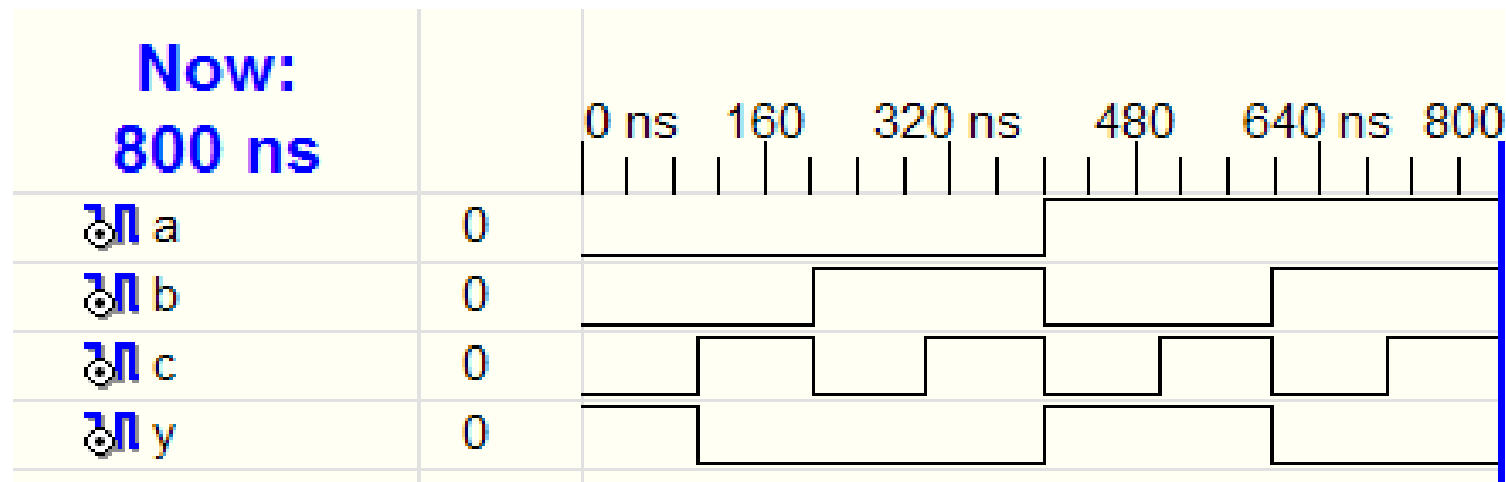
```
module example(input  a, b, c,  
               output y);  
    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b &  c;  
endmodule
```

Behavioral Verilog Simulation

Verilog code:

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

Timing diagram:

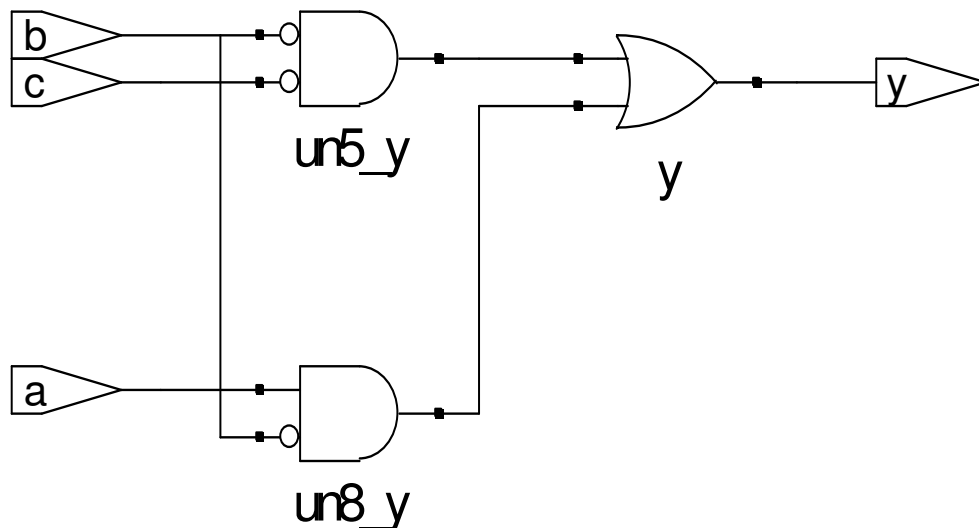


Behavioral Verilog Synthesis

Verilog:

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

Synthesis:



Verilog 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: similar to C/C++
 - // single line comment
 - /* multiline
comment */

Structural Modeling - Hierarchy

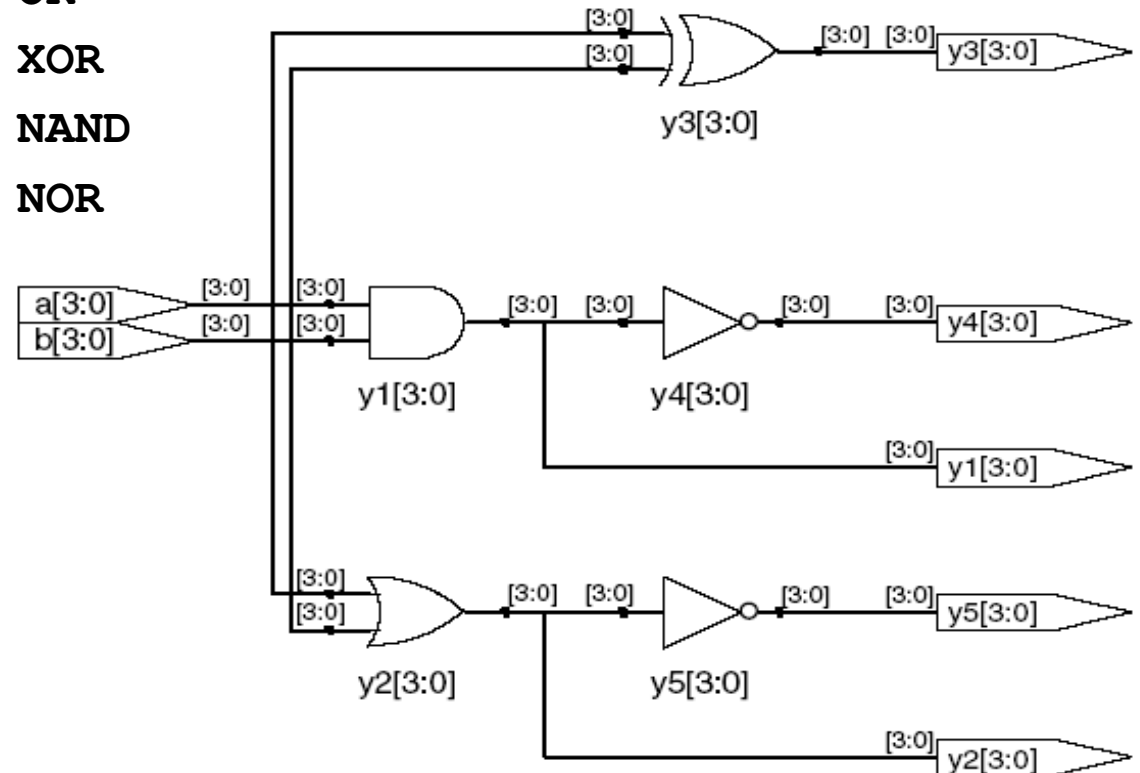
```
module and3(input  a, b, c,  
            output y);  
    assign y = a & b & c;  
endmodule
```

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

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

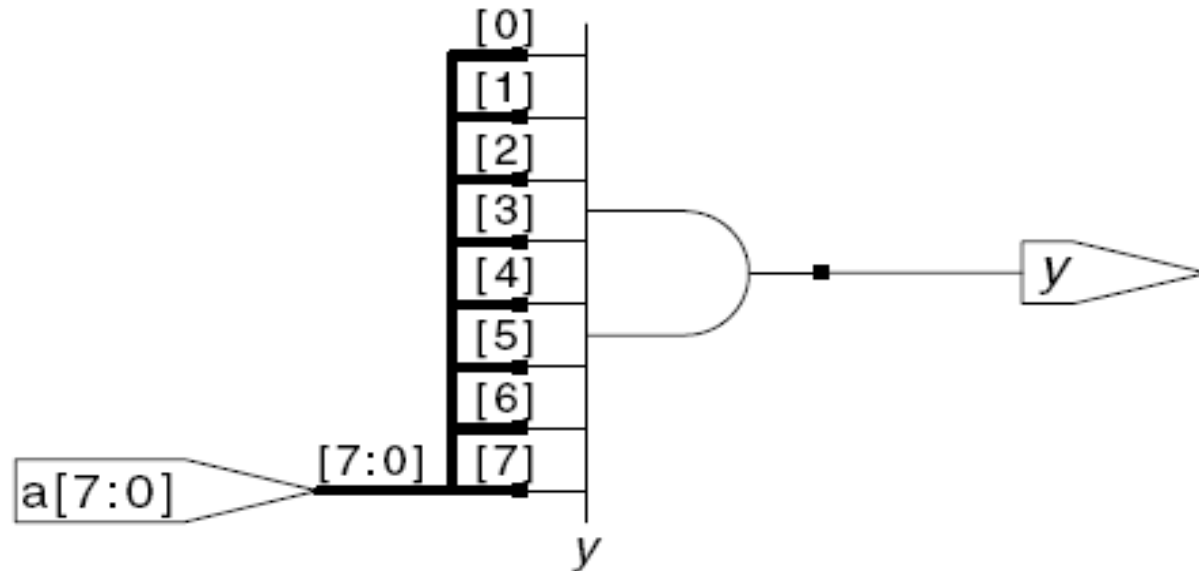
Bitwise Operators

```
module gates(input  [3:0]  a, b,  
             output [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
```



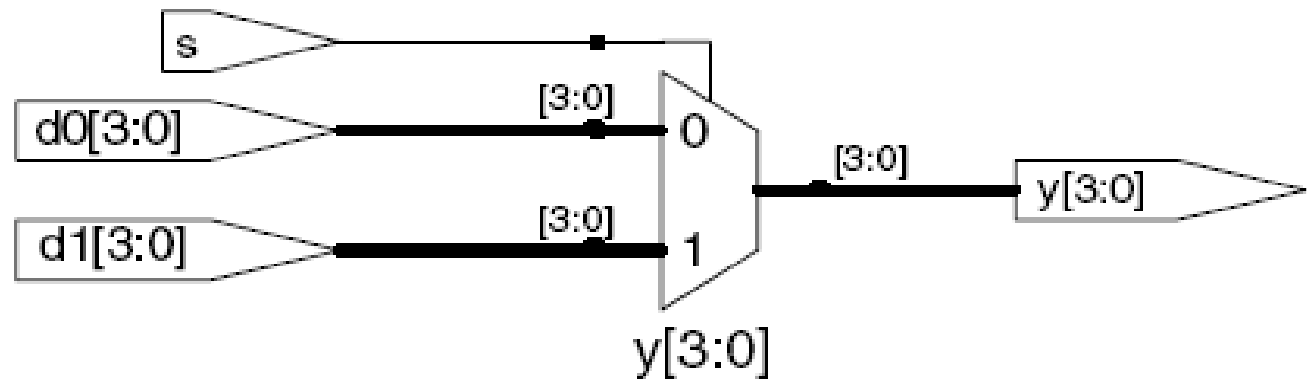
Reduction Operators

```
module and8(input  [7:0] a,  
            output          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
```



Conditional Assignment

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



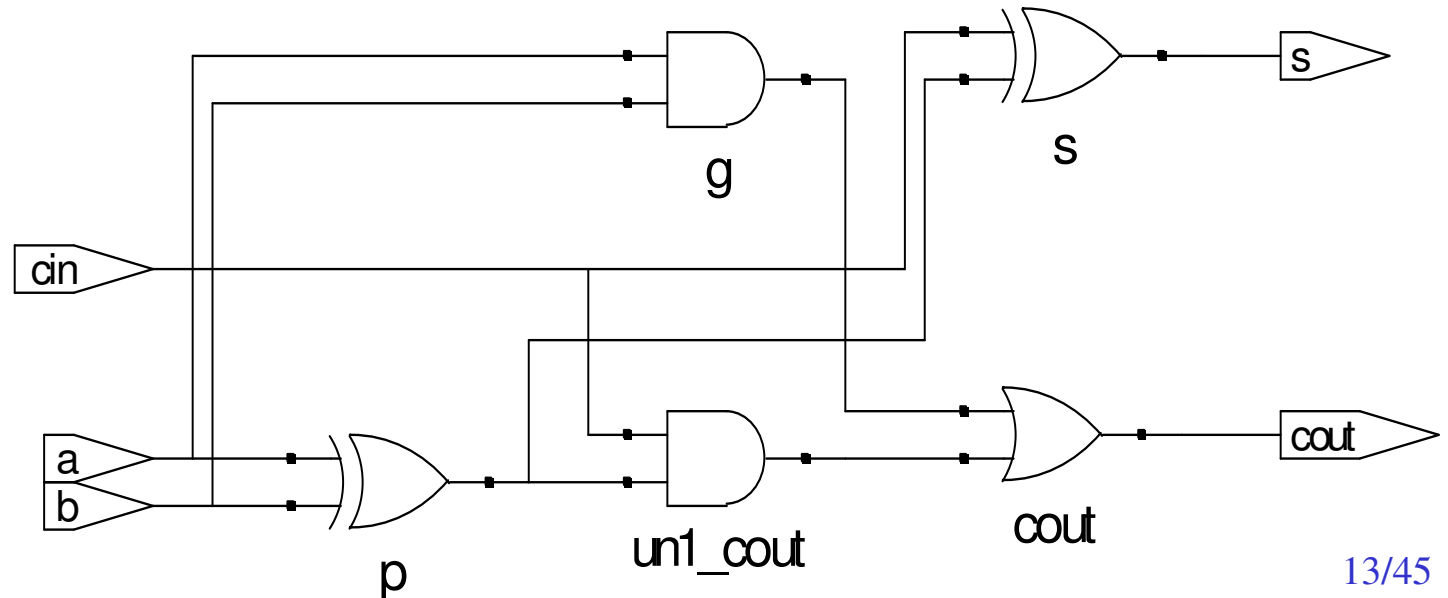
? : is also called a *ternary operator* because it operates on 3 inputs: **s**, **d1**, and **d0**

(similar to C/C++, If Condition is true ? then X : otherwise Y)

Internal Variables

```
module fulladder(input  a, b, cin, output s, cout);  
    wire p, g;          // internal nodes  
  
    assign p = a ^ b;  
    assign g = a & b;  
  
    assign s = p ^ cin;  
    assign cout = g | (p & cin);  
endmodule
```

Often it is convenient to break a complex function into intermediate steps.



Precedence

Defines the order of operations

Highest

~	NOT
*, /, %	mult, div, mod
+, -	add, sub
<<, >>	shift
<<<, >>>	arithmetic shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
, ~	OR, XOR
?:	ternary operator

Lowest

Format: N'Bvalue

Numbers

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	Unsize	decimal	42	00...0101010

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. For example, if w is a 6-bit bus, assign w b11 gives w the value 000011.

Underscores in numbers are ignored and can be helpful in breaking long numbers into more readable chunks.

Bit Manipulations: Example 1

```
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. Verilog ignores  
  them. */
```

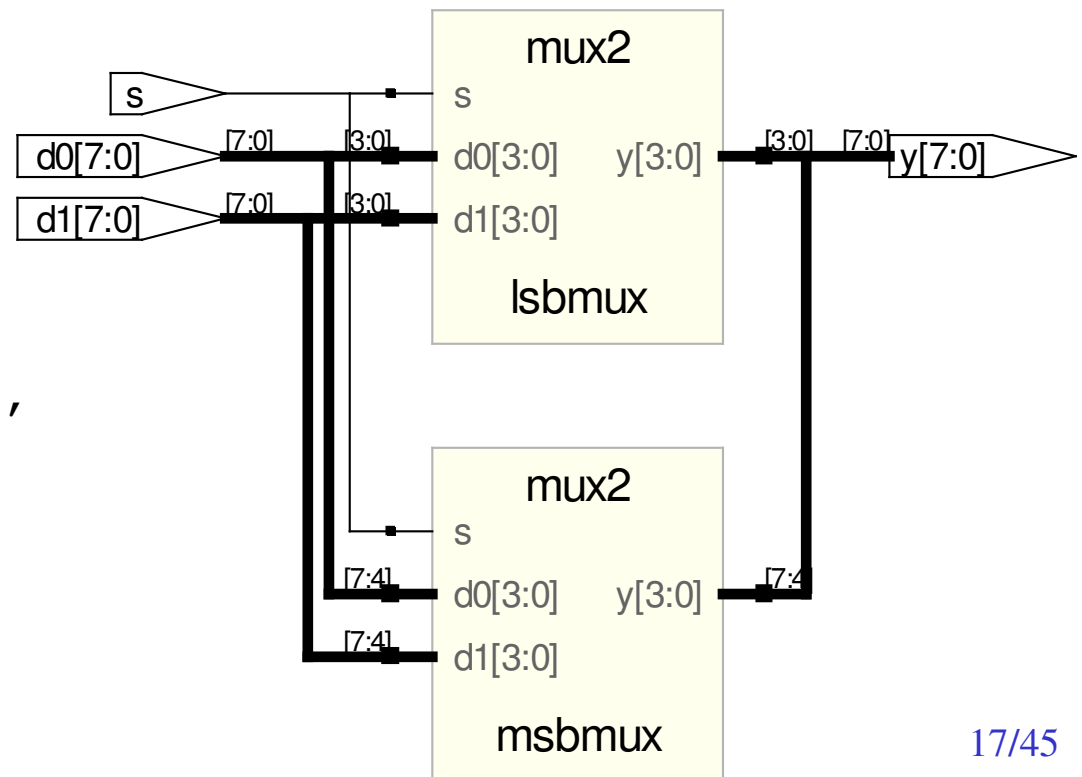

Bit Manipulations: Example 2

Verilog:

```
module mux2_8(input  [7:0] d0, d1,  
              input          s,  
              output [7:0] y);  
  
    mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]); // least significant bits  
    mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]); // most significant bits  
endmodule
```

Synthesis:

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



Z: Floating Output

Verilog:

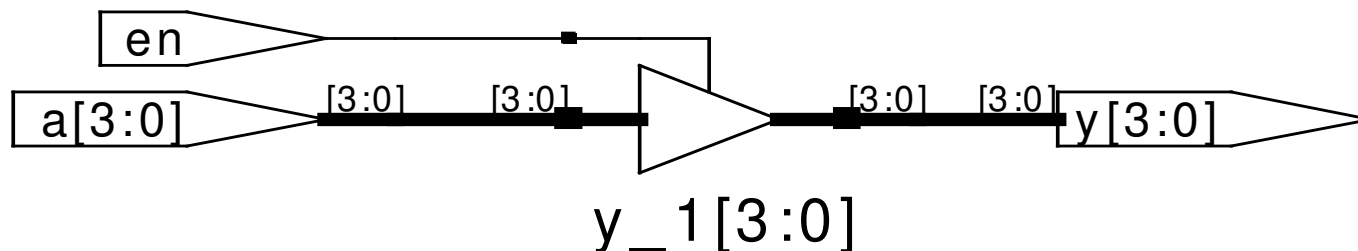
```
module tristate(input [3:0] a,  
                input      en,  
                output [3:0] y);  
    assign y = en ? a : 4'bz;  
endmodule
```

The symbol z indicates that a node is being driven neither HIGH nor LOW. The node is said to be *floating*, *high impedance*, or *high z*.

One common way to produce a floating node is to forget to connect a voltage to a circuit input, or to assume that an unconnected input is the same as an input with the value of 0. This mistake may cause the circuit to behave erratically as the floating input randomly changes from 0 to 1.

Synthesis:

If the buffer is enabled, the output is the same as the input. If the buffer is disabled, the output is assigned a floating value (z).



X: Invalid Logic Level

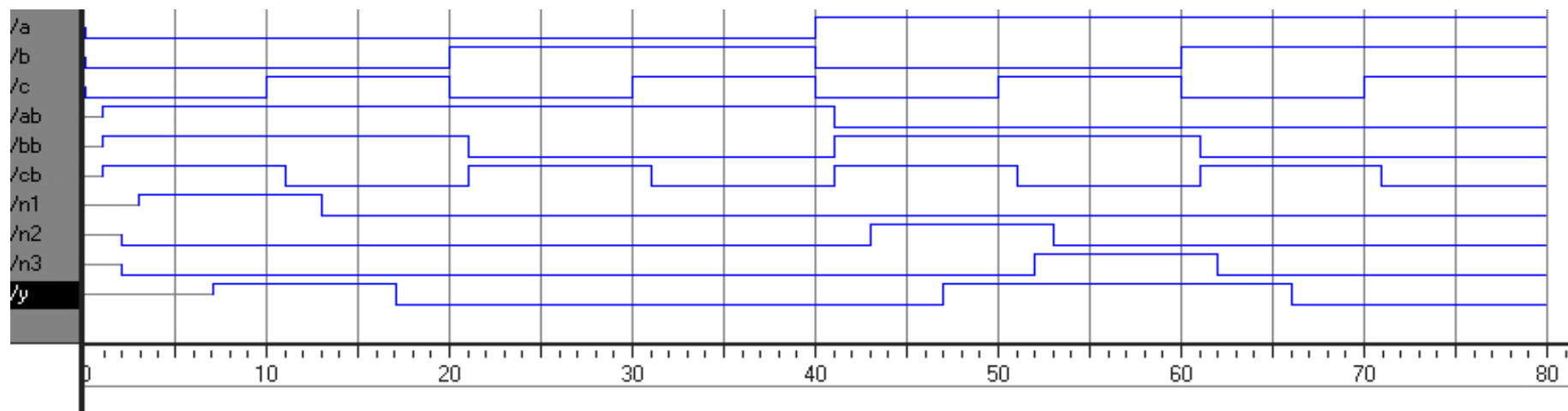
HDLs use x to indicate an invalid logic level. If a bus is simultaneously driven to 0 and 1 by two enabled tristate buffers (or other gates), the result is x, indicating contention.

&		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

Delays

```
'timescale 1ns/1ps
```

```
module example(input  a, b, c,  
                output y);  
    wire 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
```



Sequential Logic

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

IMPORTANT:

When describing circuits using an HDL, it's critical to think of the **hardware** the code should produce.

Always Statement

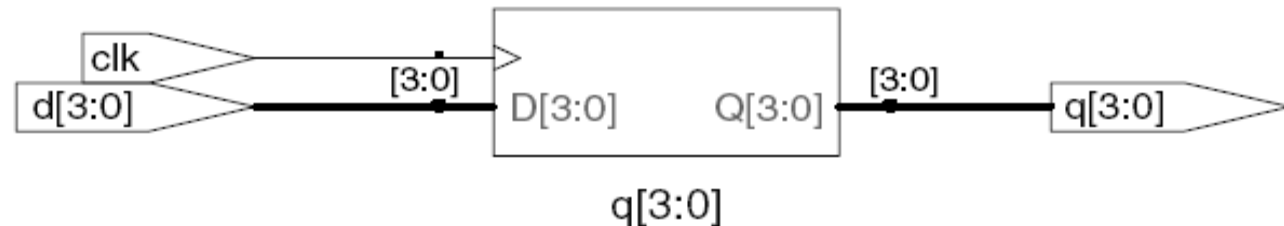
General Structure:

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

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

D Flip-Flop

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



Any signal assigned in an always statement must be declared reg. In this case q is declared as reg

Beware: A variable declared reg is not necessarily a registered output.

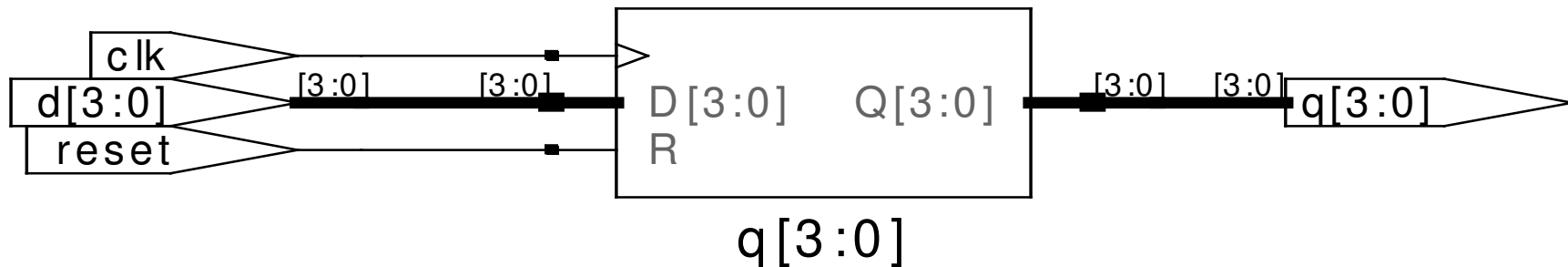
Resettable D Flip-Flop

```
module flopr(input          clk,  
             input          reset,  
             input  [3:0] d,  
             output reg [3:0] q);
```

```
// synchronous reset
```

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

```
endmodule
```

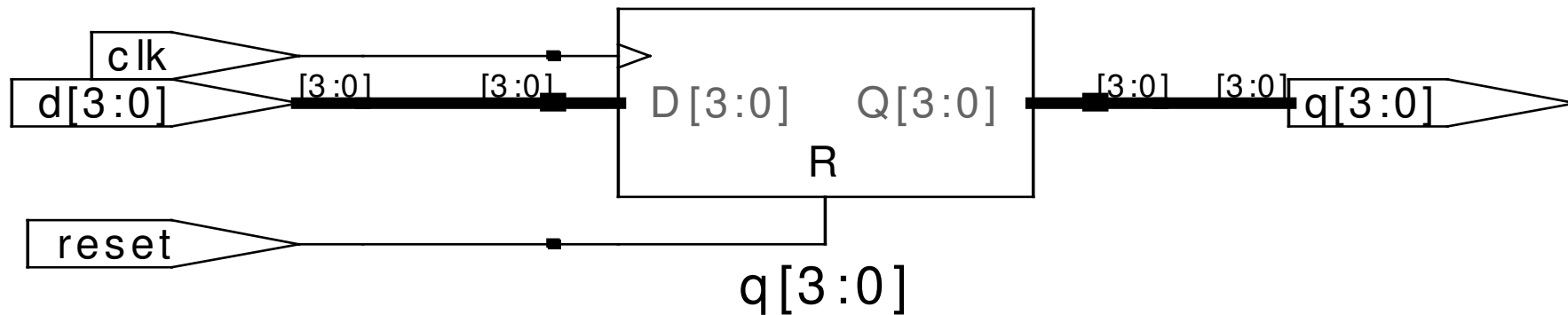


Resettable D Flip-Flop

```
module flopr(input          clk,
             input          reset,
             input  [3:0] d,
             output reg [3:0] q);

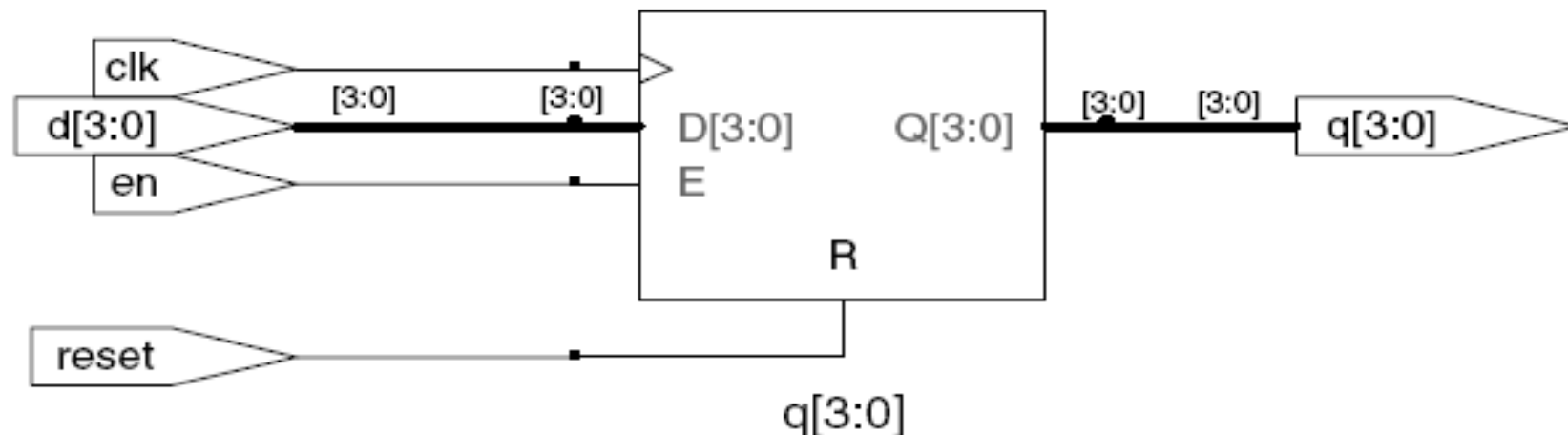
    // asynchronous reset
    always @ (posedge clk, posedge reset)
        if (reset) q <= 4'b0;
        else      q <= d;

endmodule
```



D Flip-Flop with Enable

```
module flopren(input          clk,  
               input          reset,  
               input          en,  
               input [3:0] d,  
               output reg [3:0] q);  
  
    // asynchronous reset and enable  
    always @ (posedge clk, posedge reset)  
        if      (reset) q <= 4'b0;  
        else if (en)    q <= d;  
  
endmodule
```



Other Behavioral Statements

- Statements that must be inside always statements:
 - **if / else**
 - **case, casez**
- **Reminder:** Variables assigned in an **always** statement must be declared as reg (even if they're not actually registered!)

Combinational Logic using always

```
// combinational logic using an always statement
module gates(input      [3:0] a, b,
              output reg [3:0] y1, y2, y3, y4, y5);
    always @(*)          // 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
```

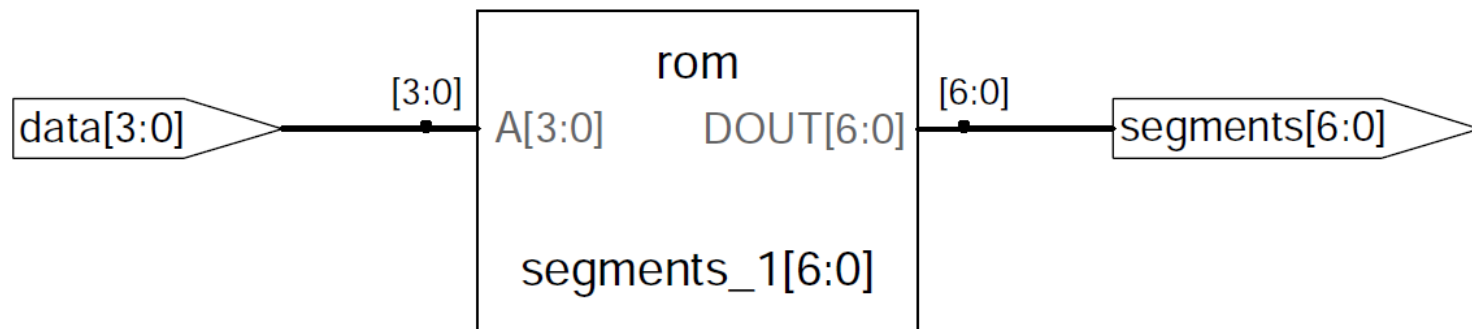
This hardware could be described with assign statements using fewer lines of code, so it's better to use assign statements in this case.

Combinational Logic using case

```
module sevenseg(input      [3:0] data,
                 output reg [6:0] segments);
    always @(*)
        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; // required
        endcase
endmodule
```

- In order for a **case** statement to imply combinational logic, all possible input combinations must be described by the HDL.
- Remember to use a **default** statement when necessary.

This synthesizes a *read-only memory* (*ROM*) containing the 7 outputs for each of the 16 possible inputs (addresses).



Combinational Logic using casez

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

```
  always @(*)
```

```
    casez (a)
```

```
      // ? = don't care
```

```
      4'b1???: y = 4'b1000;
```

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

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

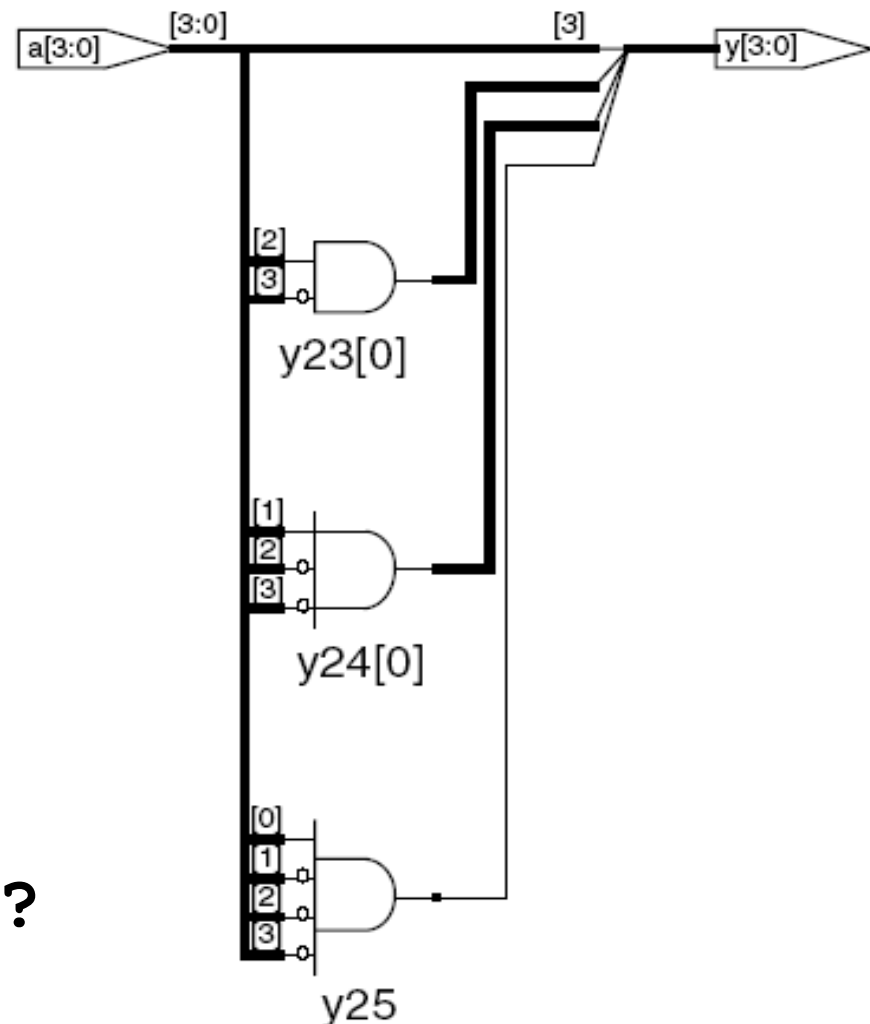
```
      4'b0001: y = 4'b0001;
```

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

```
    endcase
```

```
endmodule
```

don't cares are indicated with ?
in the casez statement



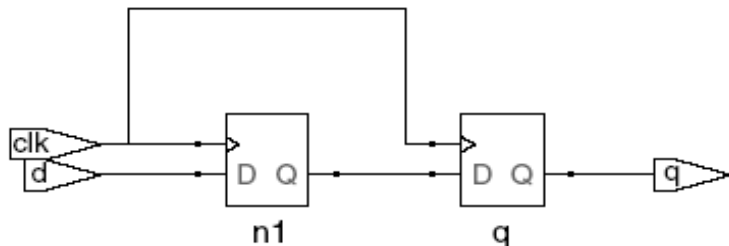
Blocking vs. Nonblocking Assignments

`<=` is a “nonblocking assignment”
Occurs simultaneously with others

`=` is a “blocking assignment”
Occurs in the order it appears in the file

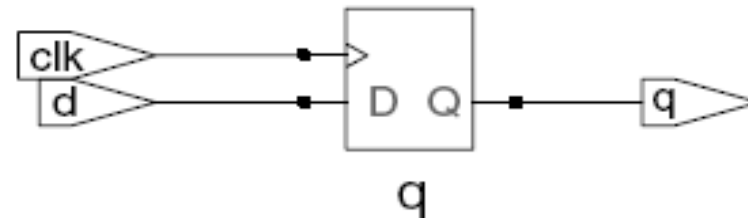
```
// Good synchronizer using
// nonblocking assignments
module syncgood(input    clk,
                 input    d,
                 output reg q);

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



```
// Bad synchronizer using
// blocking assignments
module syncbad(input    clk,
                 input    d,
                 output reg q);

    reg n1;
    always @(posedge clk)
        begin
            n1 = d;  // blocking
            q  = n1; // blocking
        end
endmodule
```

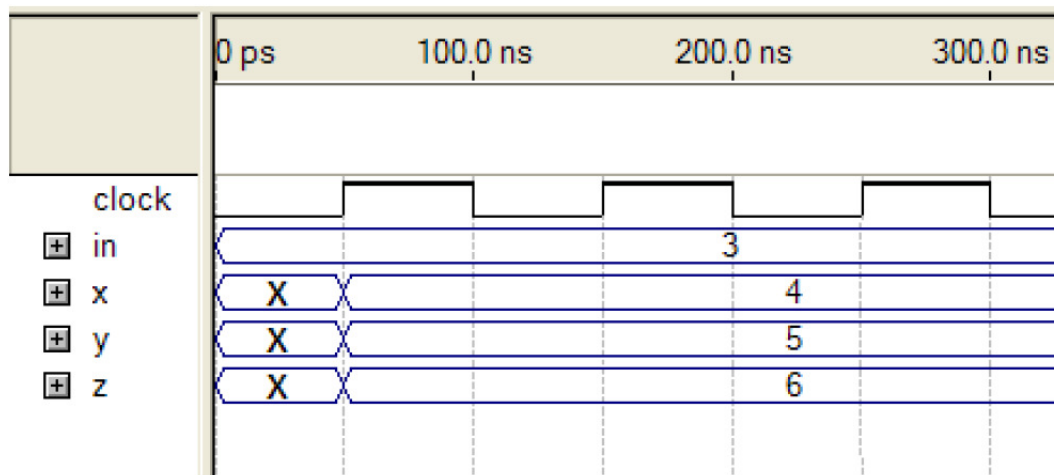
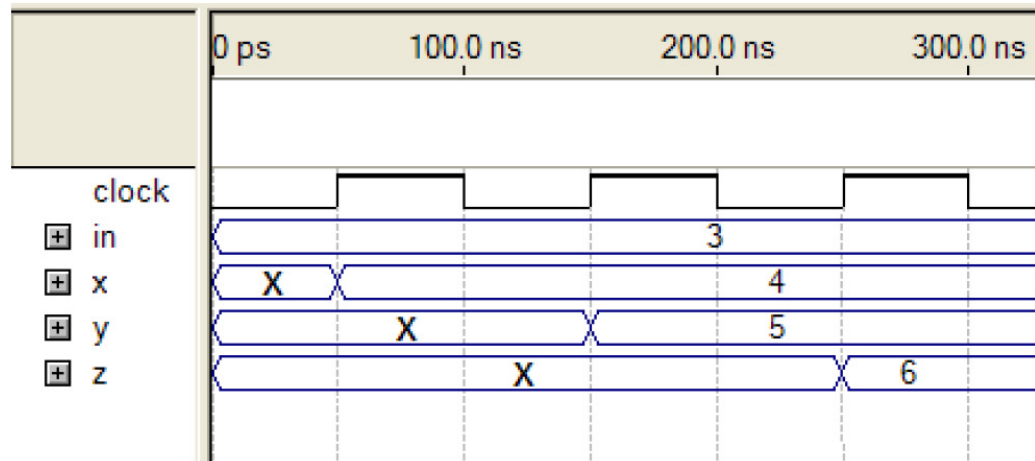


Blocking vs. Nonblocking Assignments: Another Example

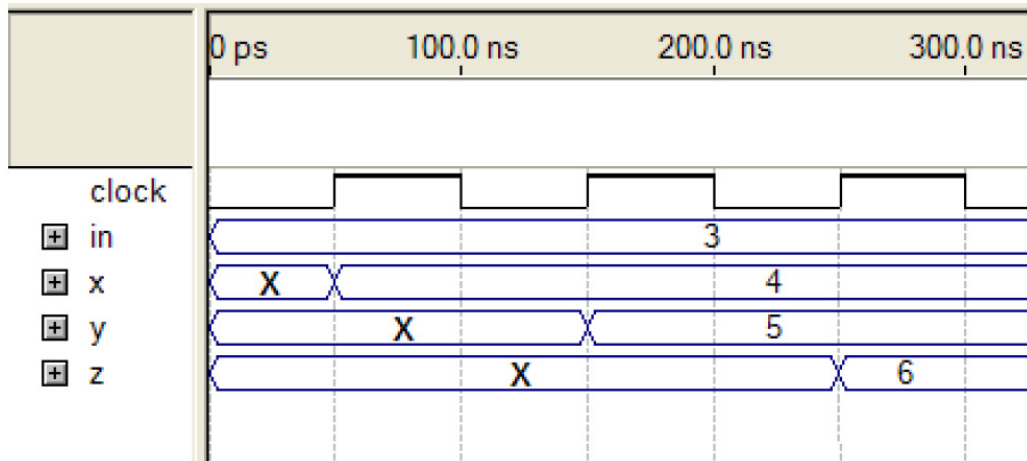
```
module test(  
    input wire clock,  
    input wire [3:0]in,  
    output reg [3:0]x,  
    output reg [3:0]y,  
    output reg [3:0]z );  
  
always @(posedge clock)  
begin  
    x <= in + 1;  
    y <= x + 1;  
    z <= y + 1;  
end  
  
endmodule
```

```
module test(  
    input wire clock,  
    input wire [3:0]in,  
    output reg [3:0]x,  
    output reg [3:0]y,  
    output reg [3:0]z );  
  
always @(posedge clock)  
begin  
    x = in + 1;  
    y = x + 1;  
    z = y + 1;  
end  
  
endmodule
```


Blocking vs. Nonblocking Assignments: Another Example

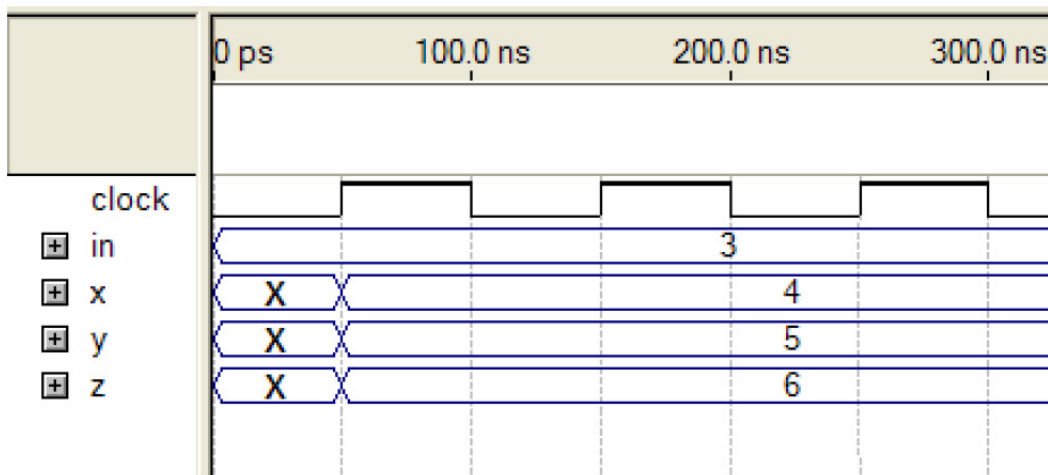


Blocking vs. Nonblocking Assignments: Another Example



```
always @(posedge clock)
begin
    x <= in + 1;
    y <= x + 1;
    z <= y + 1;
end

endmodule
```



```
always @(posedge clock)
begin
    x = in + 1;
    y = x + 1;
    z = y + 1;
end

endmodule
```

```
always @(posedge clock)
begin
    x = in + 1;
    y = in + 2; // y = x+1 = (in+1)+1
    z = in + 3; // z = y+1 = ((in+1)+1)+1
end
```

Rules for Signal Assignment

- Use **always @ (posedge clk)** and nonblocking assignments (**<=**) to model synchronous sequential logic

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

- Use continuous assignments (**assign ...**) to model simple combinational logic.

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

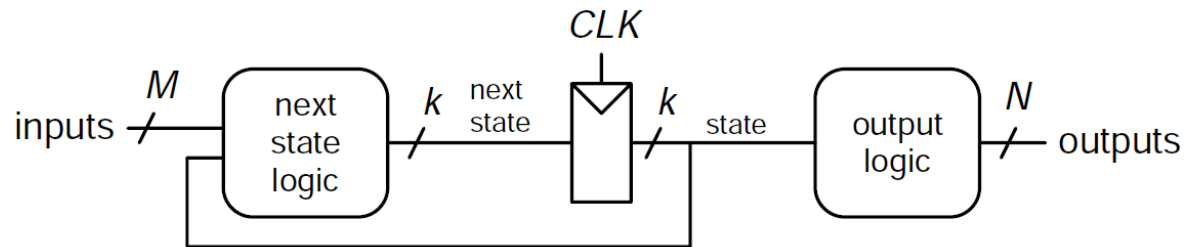
- Use **always @ (*)** and blocking assignments (**=**) to model more complicated combinational logic where the **always** statement is helpful.
- Do not make assignments to the same signal in more than one **always** statement or continuous assignment statement.

Finite State Machines (FSMs)

Synchronous sequential circuits can be drawn in the forms shown below. These forms are called *finite state machines (FSMs)*. They get their name because a circuit with k registers can be in one of a finite number (2^k) of unique states. An FSM has M inputs, N outputs, and k bits of state. It also receives a clock and, optionally, a reset signal. An FSM consists of two blocks of combinational logic, *next state logic* and *output logic*, and a register that stores the state. On each clock edge, the FSM advances to the next state, which was computed based on the current state and inputs.

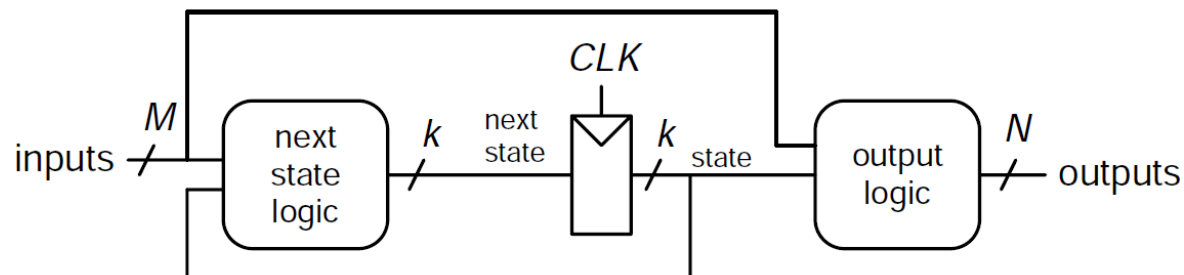
In *Moore machines*, the outputs depend only on the current state of the machine.

In *Mealy machines*, the outputs depend on both the current state and the current inputs.



(a)

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

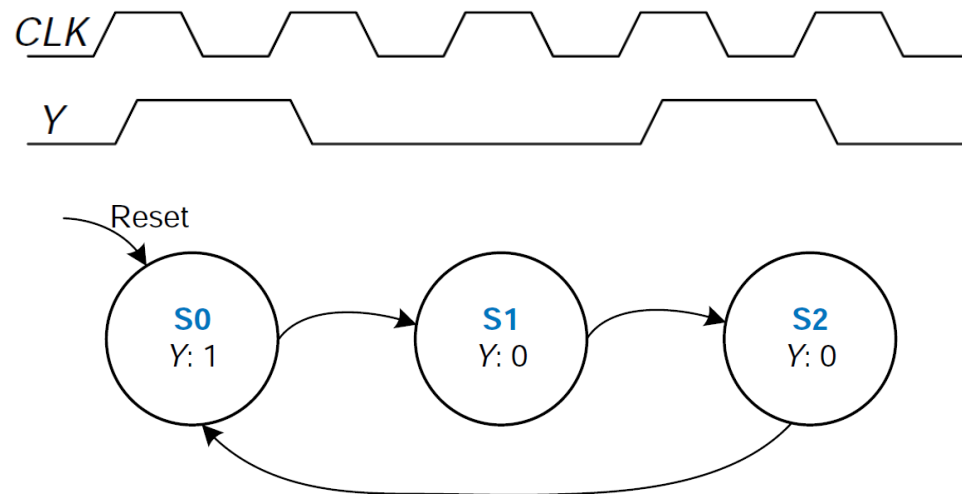


(b)

- (a) Moore machine
- (b) Mealy machine

FSM Example: Divide by 3

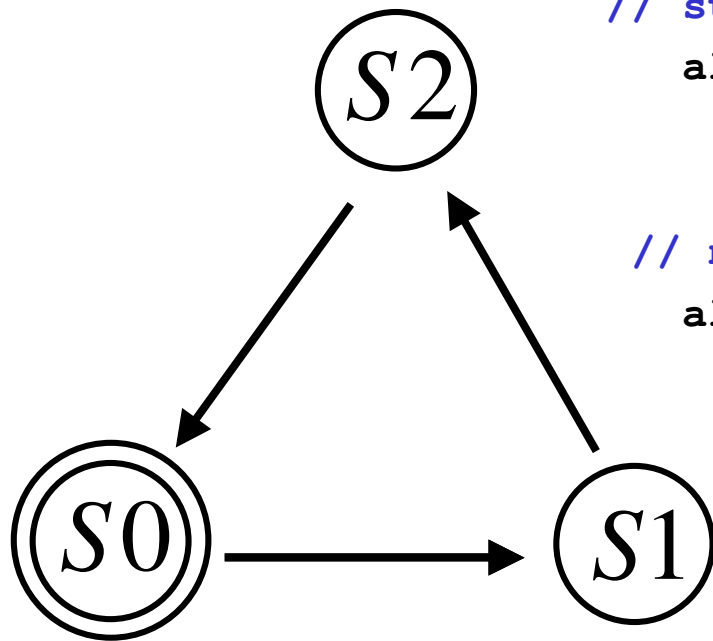
A *divide-by- N counter* has one output and no inputs. The output Y is HIGH for one clock cycle out of every N . In other words, the output divides the frequency of the clock by N . The waveform and state transition diagram for a divide-by-3 counter is shown below:



Divide-by-3 counter
state transition table

Current State	Next State
S0	S1
S1	S2
S2	S0

FSM Example: Divide by 3



The double circle indicates
the reset state

```
module divideby3FSM (input clk, reset, output q);  
    reg [1:0] state, nextstate;
```

```
    parameter S0 = 2'b00;  
    parameter S1 = 2'b01;  
    parameter S2 = 2'b10;
```

The **parameter** statement is used to define constants within a module. Naming the states with parameters is not required, but it makes changing state encodings much easier and makes the code more readable.

```
// state register
```

```
always @ (posedge clk, posedge reset)  
    if (reset) state <= S0;  
    else      state <= nextstate;
```

```
// next state logic
```

```
always @ (*)  
    case (state)  
        S0:      nextstate = S1;  
        S1:      nextstate = S2;  
        S2:      nextstate = S0;  
        default: nextstate = S0;
```

```
    endcase
```

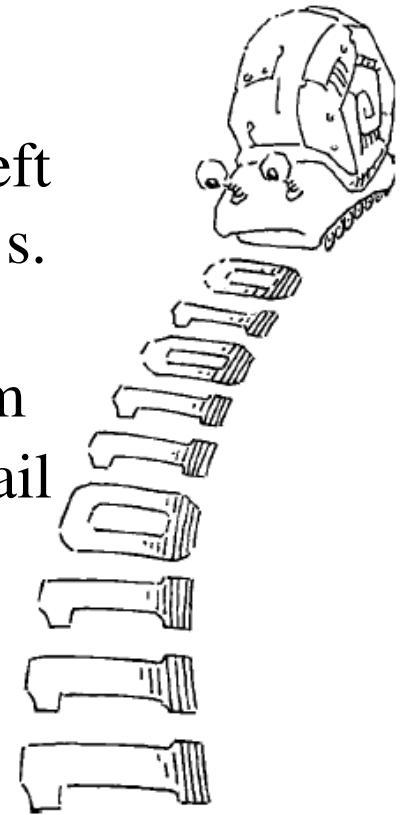
```
// output logic
```

```
    assign q = (state == S0);
```

```
endmodule
```

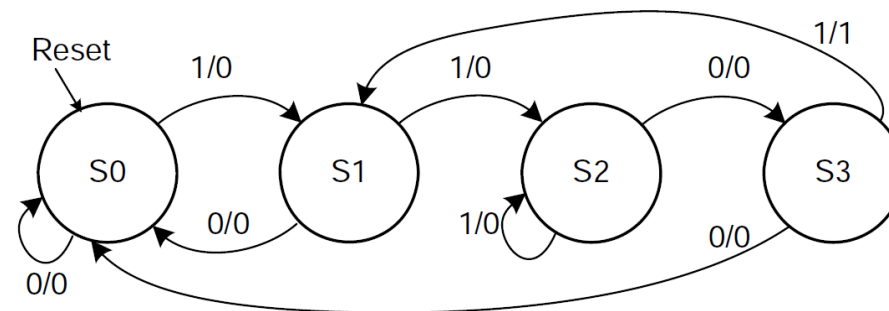
Another FSM Example: Pattern Recognition

A pet robotic snail has an FSM brain. The snail crawls from left to right along a paper tape containing a sequence of 1's and 0's. On each clock cycle, the snail crawls to the next bit. The snail smiles when the last four bits that it has crawled over are, from left to right, **1101**. Design the FSM to compute when the snail should smile. The output Y is TRUE when the snail smiles.



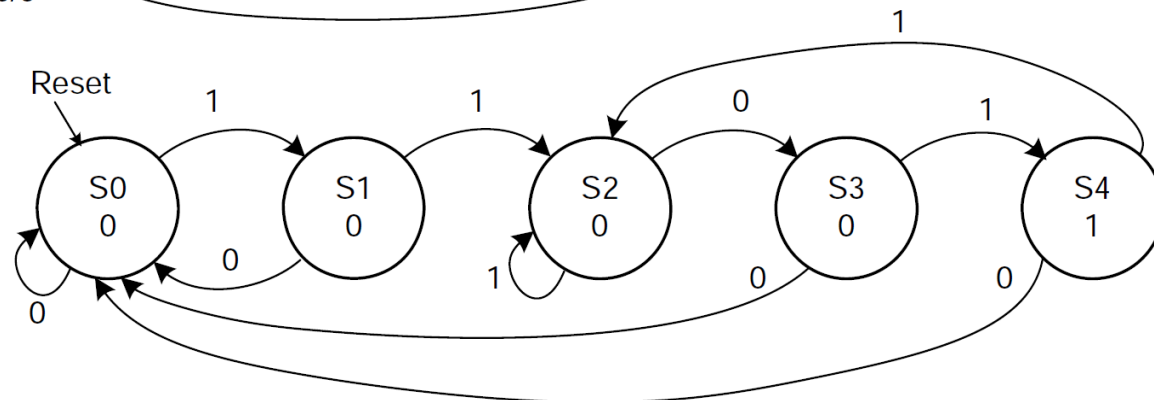
Mealy FSM

the outputs depend on both the current state and the current inputs.

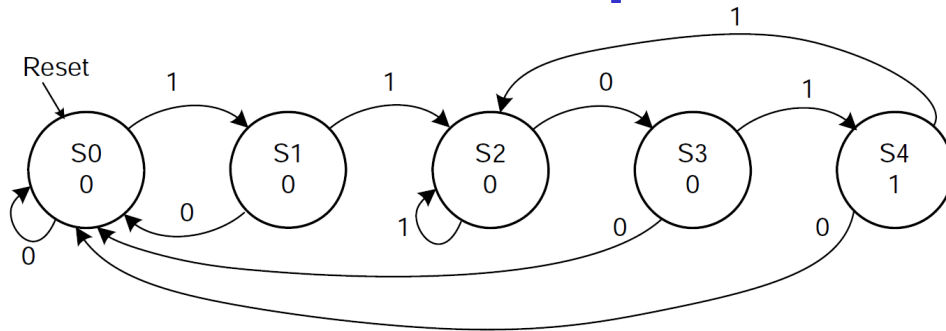


Moore FSM

the outputs depend only on the current state of the machine.



Another FSM Example: Pattern Recognition: Moore FSM



```

module patternMoore (input clk,
                    input reset,
                    input a,
                    output y);

```

```

    reg [2:0] state, nextstate;

```

```

    parameter S0 = 3b000;
    parameter S1 = 3b001;
    parameter S2 = 3b010;
    parameter S3 = 3b011;
    parameter S4 = 3b100;

```

```

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

```

```

// next state logic
always @ (*)

```

```

    case (state)

```

```

        S0: if (a) nextstate = S1;
            else nextstate = S0;

```

```

        S1: if (a) nextstate = S2;
            else nextstate = S0;

```

```

        S2: if (a) nextstate = S2;
            else nextstate = S3;

```

```

        S3: if (a) nextstate = S4;
            else nextstate = S0;

```

```

        S4: if (a) nextstate = S2;
            else nextstate = S0;

```

```

        default: nextstate = S0;

```

```

    endcase

```

```

// output logic

```

```

assign y = (state == S4);

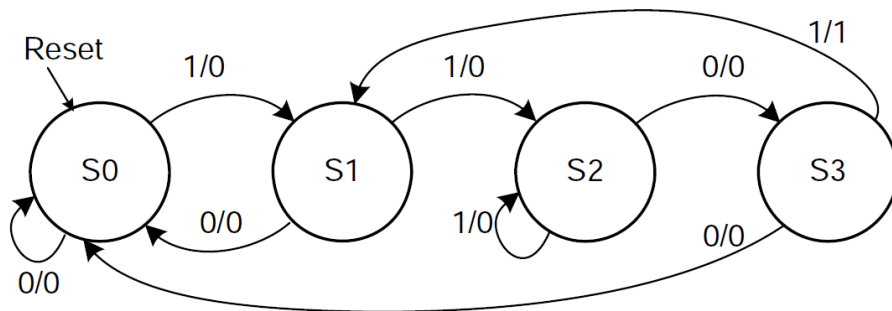
```

```

endmodule

```


Another FSM Example: Pattern Recognition: Mealy FSM



```

module patternMealy (input clk,
                    input reset,
                    input a,
                    output y);

  reg [1:0] state, nextstate;

  parameter S0 = 2b00;
  parameter S1 = 2b01;
  parameter S2 = 2b10;
  parameter S3 = 2b11;
  // state register
  always @ (posedge clk, posedge reset)
    if (reset) state <= S0;
    else      state <= nextstate;

```

```

// next state logic
always @ (*)
  case (state)
    S0: if (a) nextstate = S1;
        else nextstate = S0;
    S1: if (a) nextstate = S2;
        else nextstate = S0;
    S2: if (a) nextstate = S2;
        else nextstate = S3;
    S3: if (a) nextstate = S1;
        else nextstate = S0;
    default: nextstate = S0;
  endcase

// output logic
assign y = (a & state == S3);

endmodule

```

Parameterized Modules

HDLs permit variable bit widths
using parameterized modules.

2:1 mux:

```
module mux2
```

```
    #(parameter width = 8)    // name and default value
```

```
    (input  [width-1:0] d0, d1,
```

```
     input                                s,
```

```
     output [width-1:0] y);
```

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

```
endmodule
```

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

```
    mux2 mux1(d0, d1, s, out);
```

Instance with 12-bit bus width:

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

Do not confuse the use of the #
sign indicating delays with the
use of #(...) in defining
and overriding parameters.

Testbenches

- HDL code written to test another HDL module, the *device under test* (dut), also called the *unit under test* (uut)
- Not synthesizable
- Types of testbenches:
 - Simple testbench
 - Self-checking testbench
 - Self-checking testbench with testvectors

Write 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 to expected outputs and report errors

Example

Write Verilog code to implement the following function in hardware: $y = \overline{b}\overline{c} + a\overline{b}$

Name the module **sillyfunction**

Verilog

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

```
module testbench1();
    reg  a, b, c;
    wire 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

```
module testbench2();  
    reg  a, b, c;  
    wire 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;  
        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
```