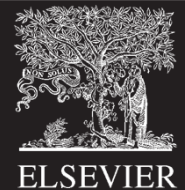


# Lecture 16




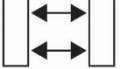
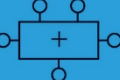

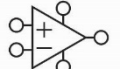


## ***Digital Design and Computer Architecture: ARM® Edition***

Sarah L. Harris and David Money Harris



# Chapter 4 :: Topics

- Introduction
- Combinational Logic
- Structural Modeling
- Sequential Logic
- More Combinational Logic
- Finite State Machines
- Testbenches

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



# 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 Vs. C

- HDL is not C, you need to think of hardware
- So you should forget about everything in C (well... not everything...)
- There are some similarities, as with any programming language, but syntax and logic are quite different.



# 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)



# SystemVerilog Modules



## Two types of Modules:

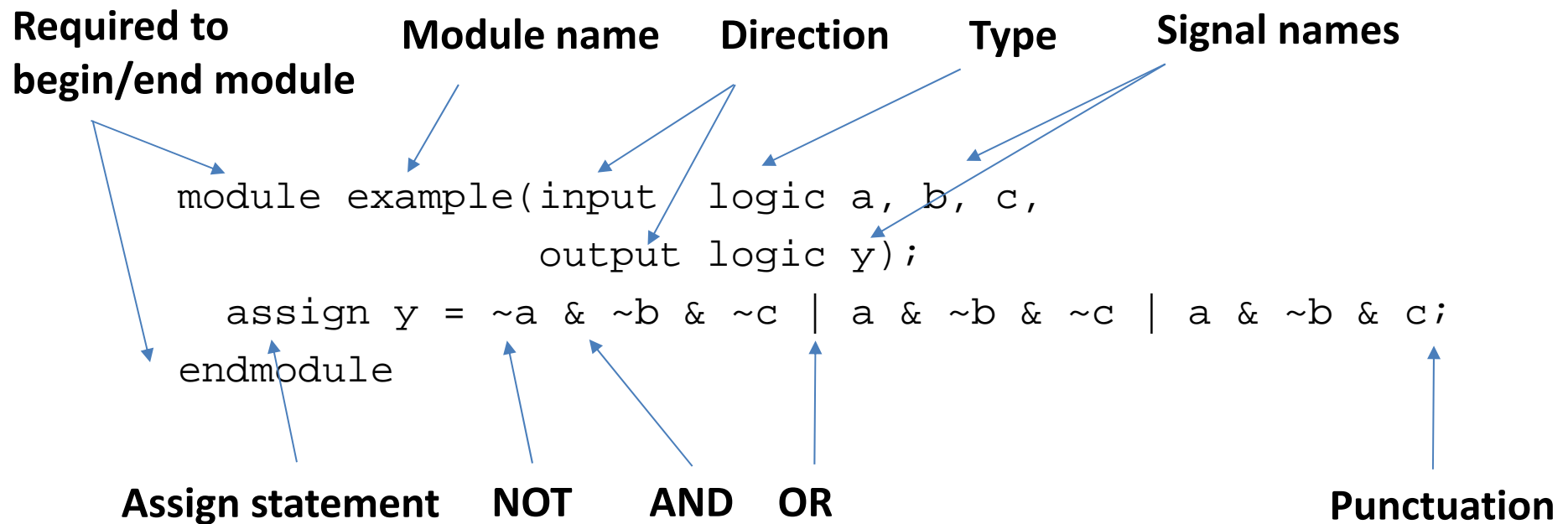
- **Behavioral:** describe what a module does
- **Structural:** describe how it is built from simpler modules



# Behavioral SystemVerilog

## SystemVerilog:

$$y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$$

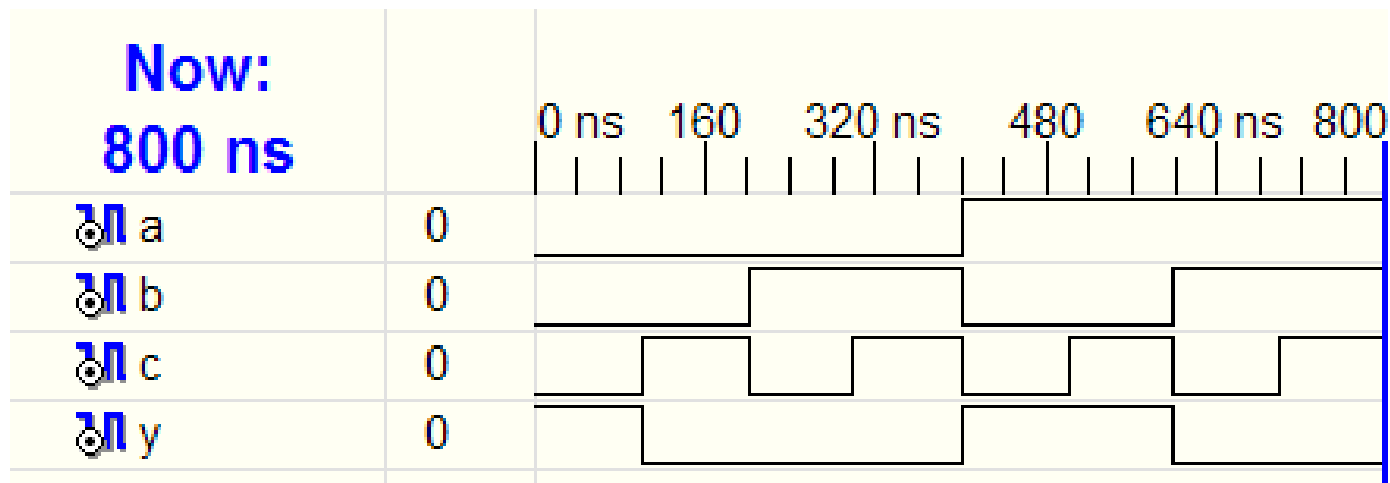


# HDL Simulation

## SystemVerilog:

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

$$y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$$



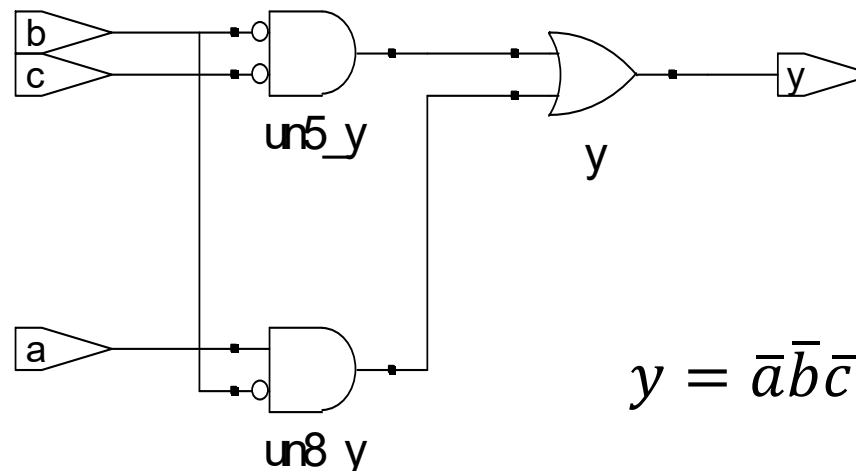


# HDL Synthesis

## SystemVerilog:

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

## Synthesis:



$$y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$$



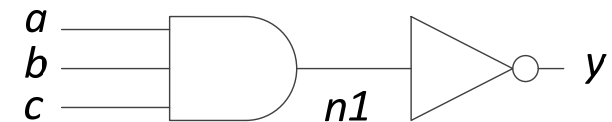
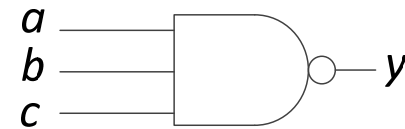
# Structural Modeling - Hierarchy

```
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
```

$$y = \overline{abc}$$



# Precedence

Highest

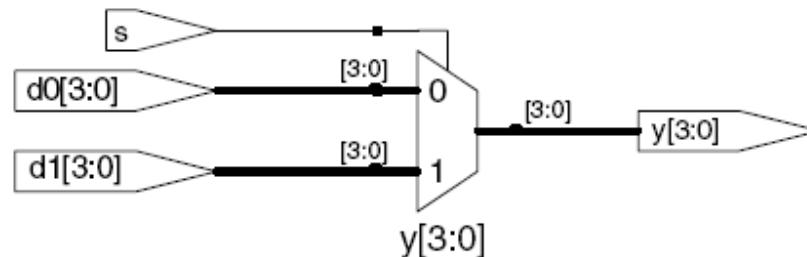
~	NOT
*, / , %	mult, div, mod
+, -	add, sub
<<, >>	shift
<<<, >>>	arithmetic shift
<, <=, >, >=	comparison
==, !=	equal, not equal
&, ~&	AND, NAND
^, ~^	XOR, XNOR
, ~	OR, NOR
? :	ternary operator

Lowest



# Conditional Assignment

```
module mux2(input  logic [3:0] d0, d1,  
            input  logic      s,  
            output logic [3:0] y);  
    assign y = s ? d1 : d0;    //If s is True then d1 will be  
endmodule                    assigned to Y otherwise d0
```



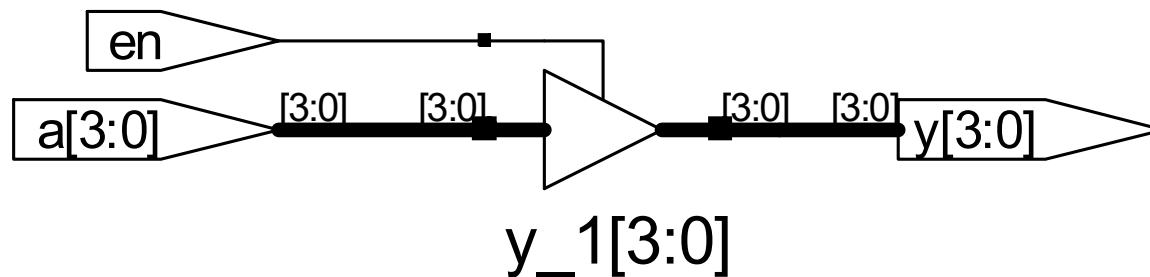
? : is also called a *ternary operator* because it operates on 3 inputs: s, d1, and d0.



# 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
```



# 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;
```

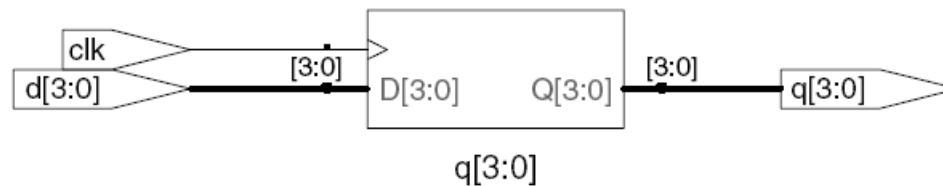
Whenever the event in `sensitivity list` occurs,  
statement is executed



# 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
```

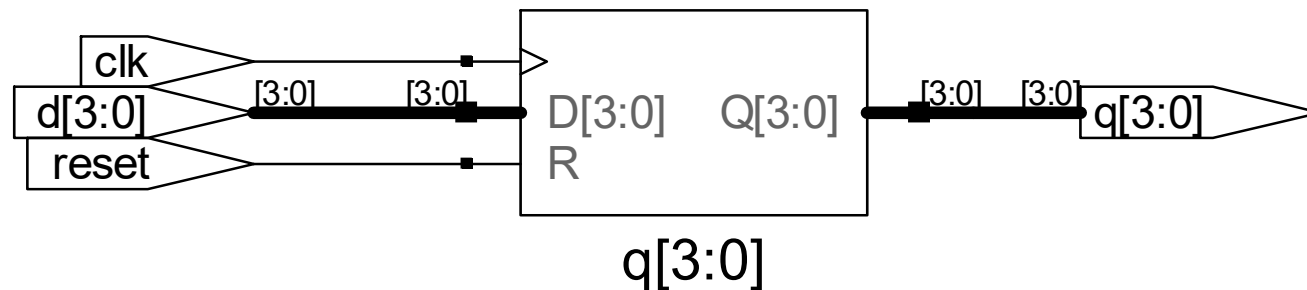
**Assignment (not smaller than equal...)**





# Resettable D Flip-Flop

```
module flopr(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
```



# Other Behavioral Statements

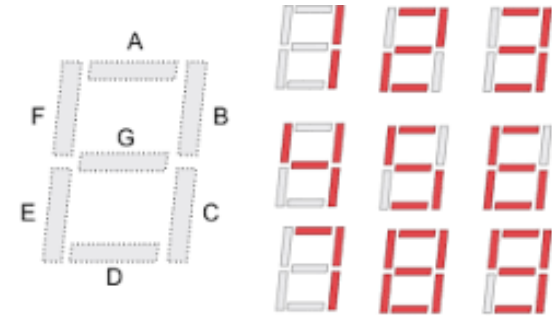
- Statements that must be inside `always` statements:
  - `if / else`
  - `case, casez`



# Combinational Logic using case

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

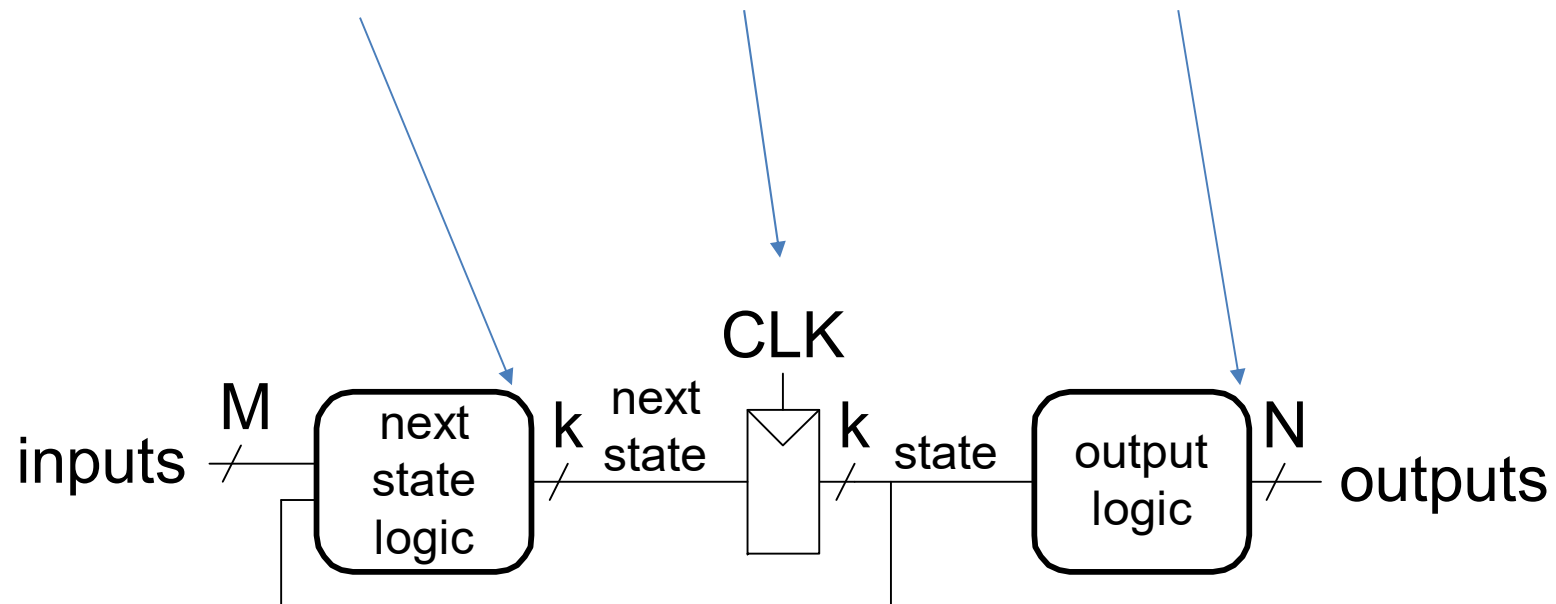
always_comb
  case (data)
    //                abc_defg
    0: segments =      7'b111_1110;
    1: segments =      7'b011_0000;
    2: segments =      7'b110_1101;
    3: segments =      7'b111_1001;
    4: segments =      7'b011_0011;
    5: segments =      7'b101_1011;
    6: segments =      7'b101_1111;
    7: segments =      7'b111_0000;
    8: segments =      7'b111_1111;
    9: segments =      7'b111_0111;
    default: segments = 7'b000_0000; // required
  endcase
endmodule
```



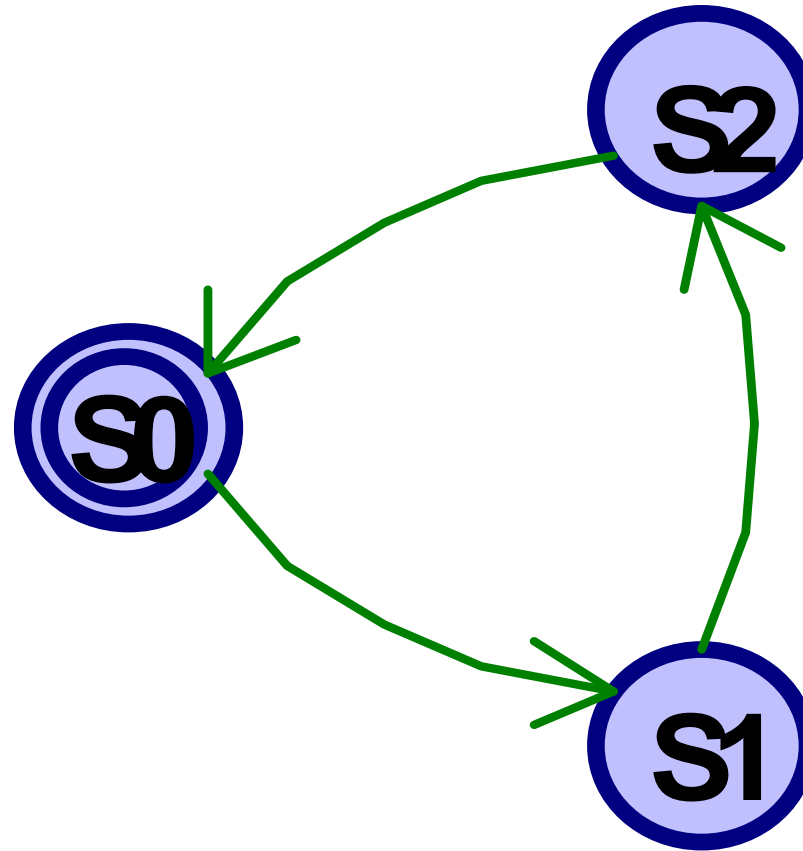
# Finite State Machines (FSMs)

- **Three blocks:**

- next state logic, state register, output logic



# FSM Example: Divide by 3



The double circle indicates the reset state



# FSM in SystemVerilog

```

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

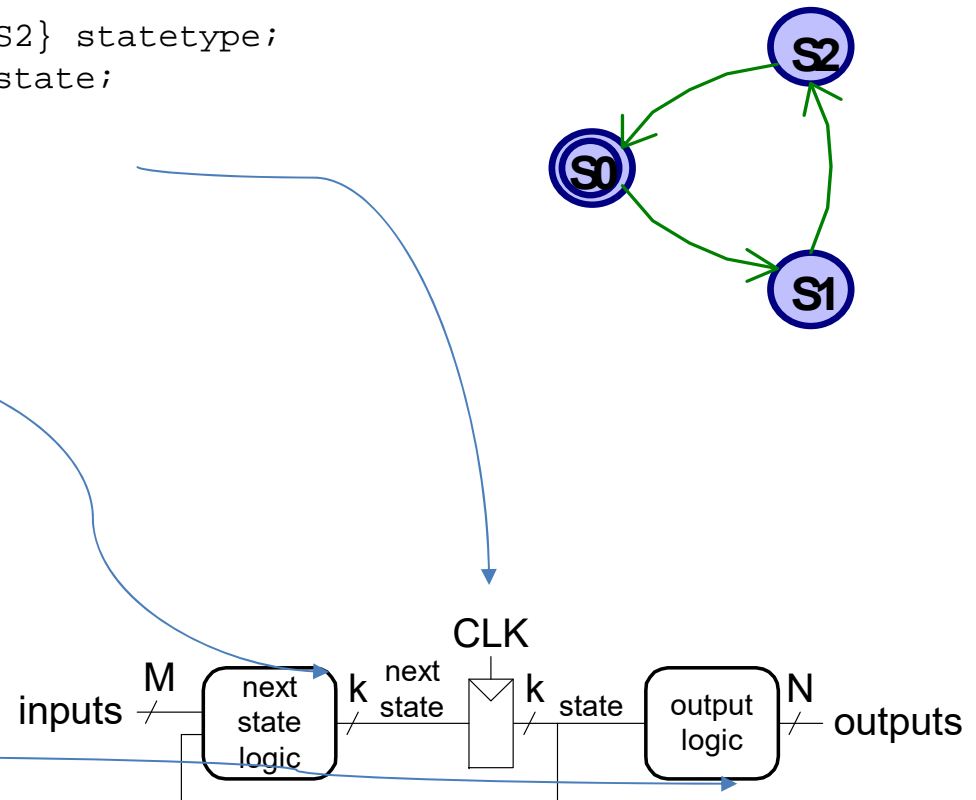
    typedef enum logic [1:0] {S0, S1, S2} statetype;
    statetype [1:0] currentstate, nextstate;

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

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

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

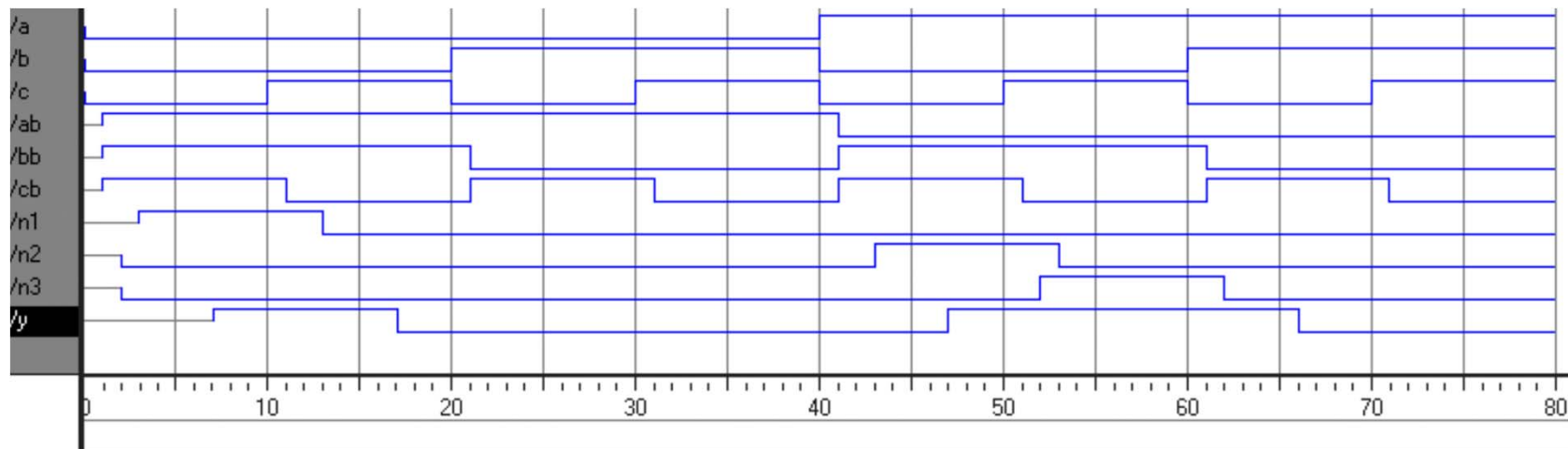
```



# Delays

```
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
```

$$y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$$



# Testbenches

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





# Testbench Example

- Write SystemVerilog code to implement the following function in hardware:

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

- Name the module `sillyfunction`



# Testbench Example

- Write SystemVerilog code to implement the following function in hardware:

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

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



# Simple Testbench

```
module testbench1();  
    logic a, b, c;  
    logic y;  
    // instantiate device under test  
    sillyfunction dut(a, b, c, y);  
    // apply inputs  
    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
```



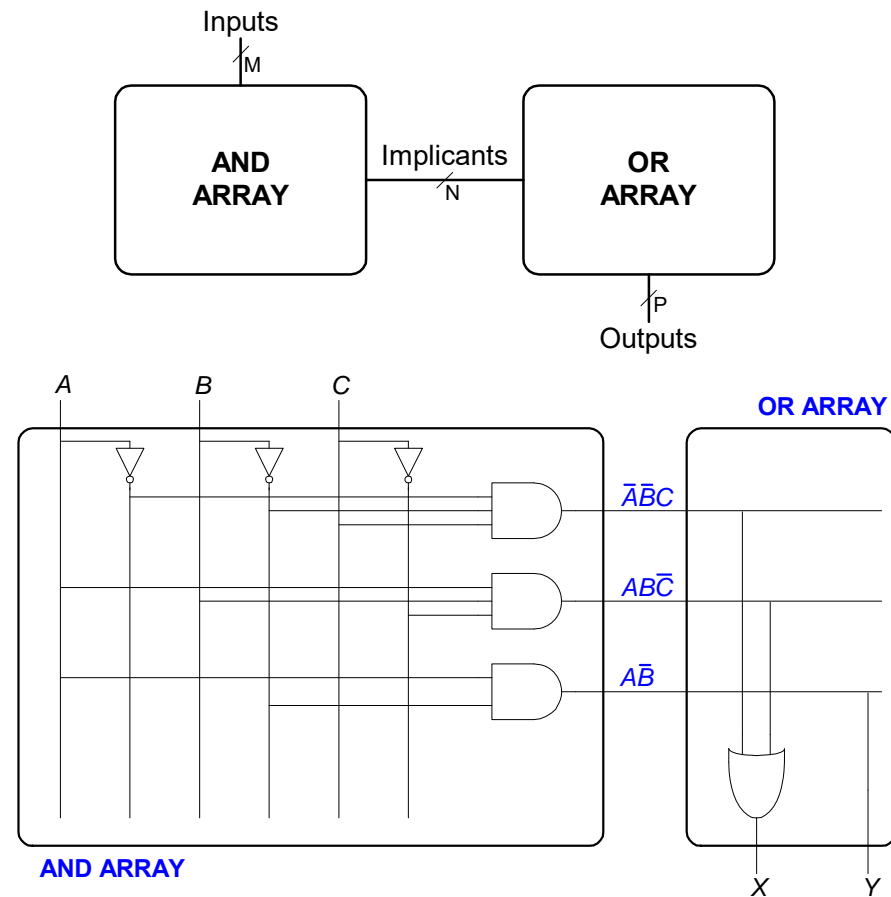
# Logic Arrays

- **PLAs** (Programmable logic arrays)
  - AND array followed by OR array
  - Combinational logic only
  - Fixed internal connections
- **FPGAs** (Field programmable gate arrays)
  - Array of Logic Elements (LEs)
  - Combinational and sequential logic
  - Programmable internal connections



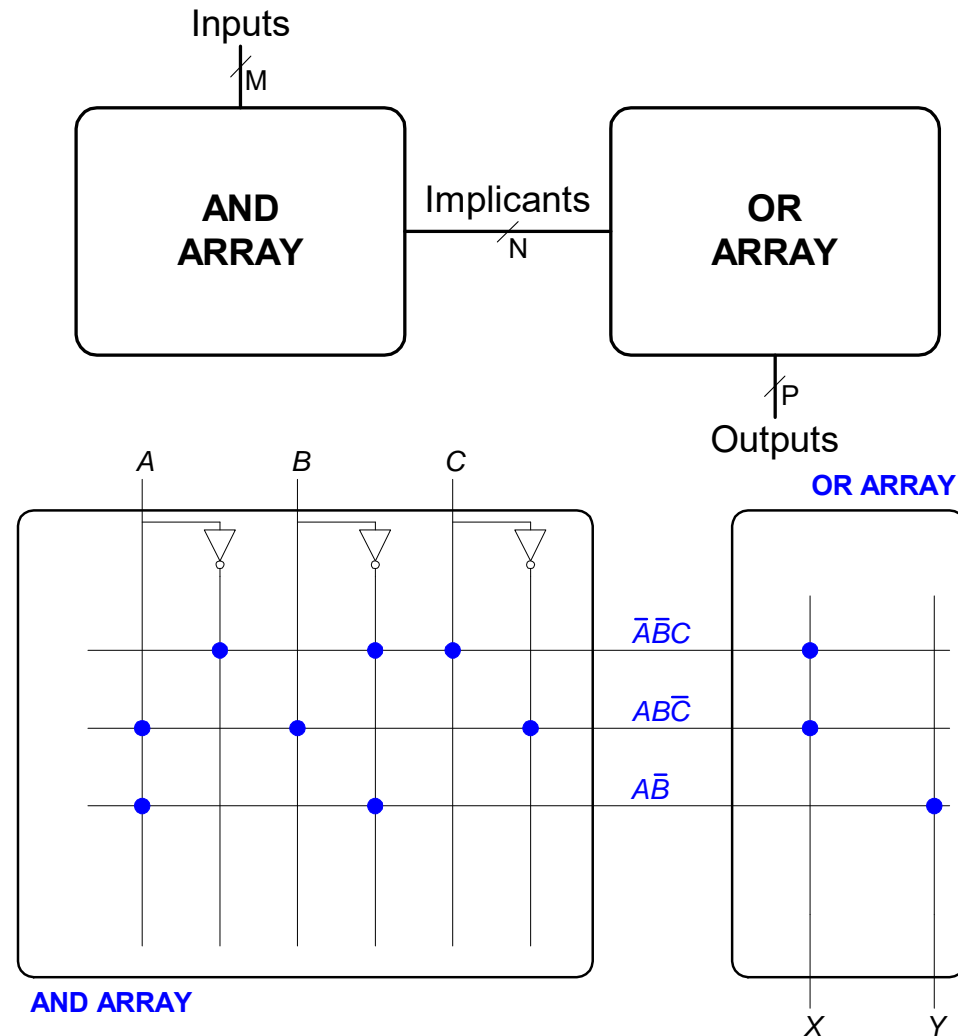
# PLAs

- $X = \bar{A}\bar{B}C + AB\bar{C}$
- $Y = AB$



# PLAs: Dot Notation

- $X = ABC + \bar{A}BC$
- $Y = AB$

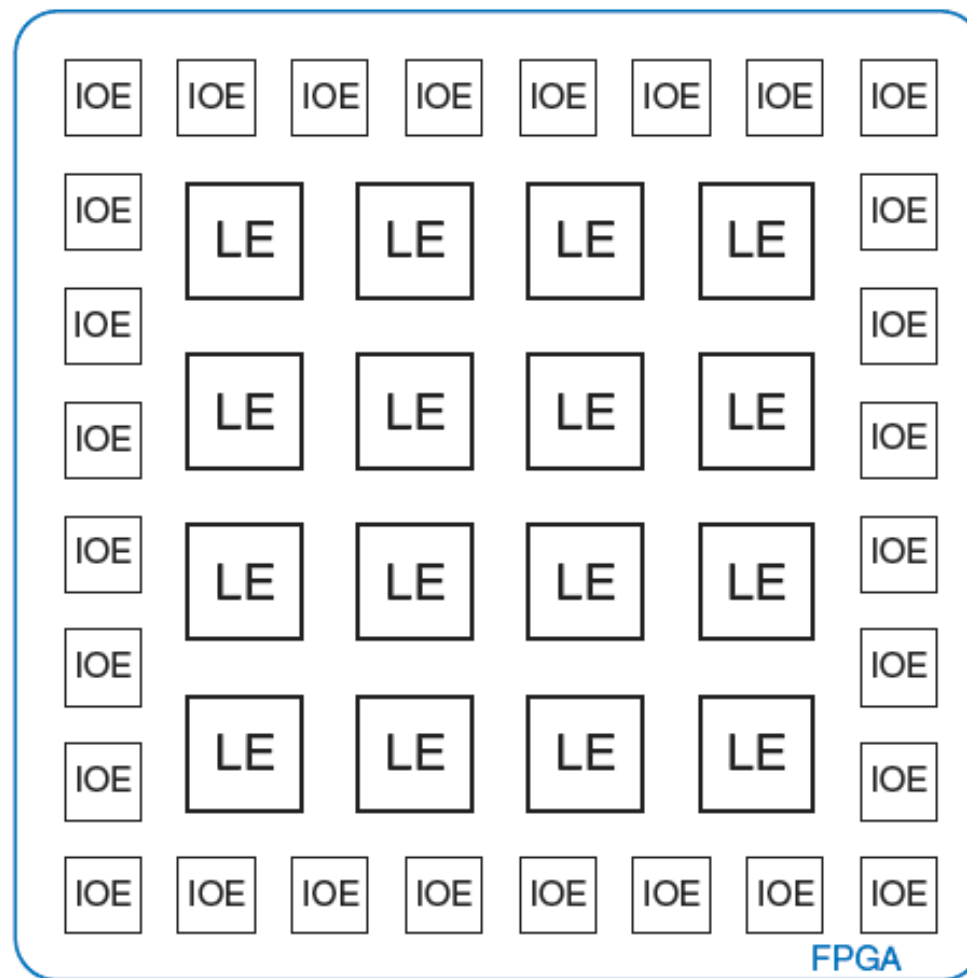


# FPGA: Field Programmable Gate Array

- Composed of:
  - **LEs** (Logic elements): perform logic
  - **IOEs** (Input/output elements): interface with outside world
  - **Programmable interconnection:** connect LEs and IOEs
  - Some FPGAs include other building blocks such as multipliers and RAMs



# General FPGA Layout





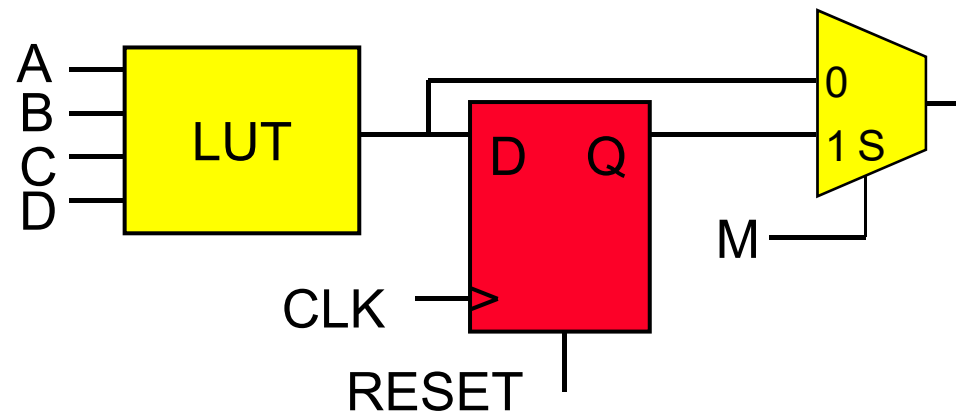
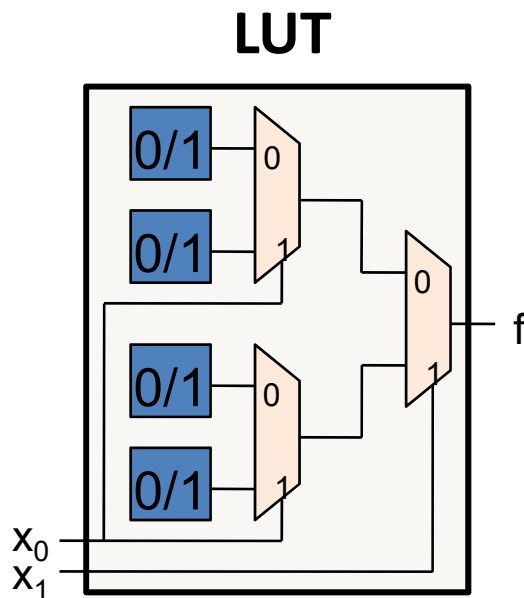
# LE: Logic Element

- Composed of:
  - **LUTs** (lookup tables): perform combinational logic
  - **Flip-flops**: perform sequential logic
  - **Multiplexers**: connect LUTs and flip-flops



# A simple FPGA cell

The simplest FPGA cell consists of a single table (e.g. Look-Up-Table - LUT), a D flip-flop and a MUX.



# Altera Cyclone IV LE

