ΗΥ 232
Οργάνωση και στον Σχεδίαση Η/Υ

Διάλεξη 8

Concepts of Digital Design
Introduction to Verilog

Νίκος Μπέλλας
Τμήμα Ηλεκτρολόγων και Μηχανικών Η/Υ

The Verilog Language

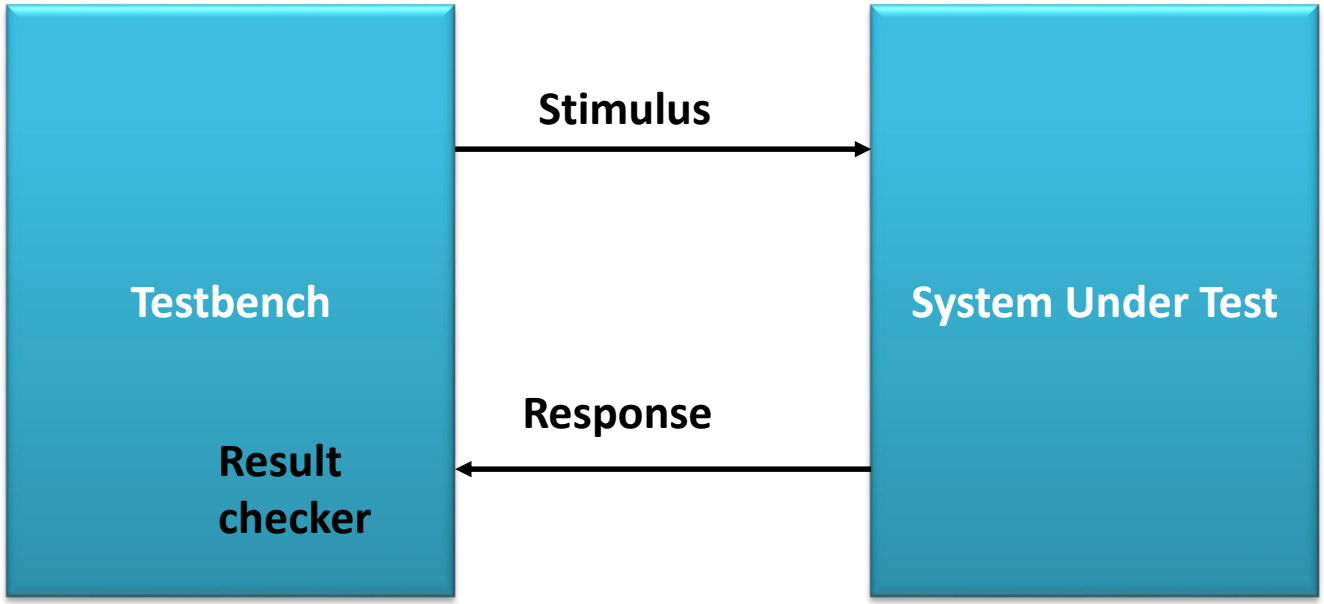
- A hardware description language (**HDL**) used for **modeling, simulation** and **synthesis** of digital electronic systems
- One of the two most commonly-used languages in digital hardware design (VHDL is the other)
- Virtually every chip (FPGA, ASIC, etc.) is designed in part using one of these two languages
- Combines **structural** and **behavioral** modeling styles
- IEEE standard 1364-1995
- **Verilog-2001** extends the initial Verilog-95 specification

Simulation

- **Simulation** is used to verify the functional characteristics of models at any level of abstraction
- One of the main uses of Verilog/VHDL is simulation
- To test if the RTL code meets the functional requirements of the specification, we must see if all the RTL blocks are functionally correct.
- To achieve this we need to write a testbench, which generates *clk*, *reset* and the required test vectors
- Simulation is needed after every refinement step
  - Behavioral, zero-delay gate level, timing, post P&R simulation

How Are Simulators Used?

- Testbench generates stimulus and checks response
- Coupled to model of the system
- Pair is run concurrently



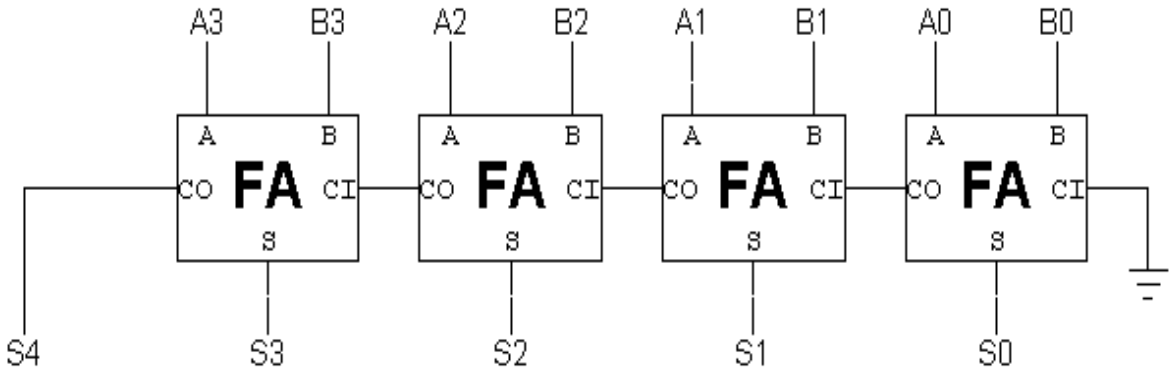
Learn by Example: Combinational Logic

Adder: a circuit that does addition

Here's an example of binary addition:

Adding two N-bit numbers produces an (N+1)-bit result

```
  1101
+0101
-----
 10010
```



"Ripple-carry adder"

1-bit Adder

Build *Truth Table* and use *Sum of Products (SOP)*

A	B	Cin	S	Cout
0	**0**	**0**	0	0
0	**0**	**1**	1	0
0	**1**	**0**	1	0
0	**1**	**1**	0	1
1	**0**	**0**	1	0
1	**0**	**1**	0	1
1	**1**	**0**	0	1
1	**1**	**1**	1	1

$$Sum: S = A \oplus B \oplus C$$

Carry Out:

$$CO = A'BC + AB'C + ABC' + ABC$$
$$= (A' + A)BC + (B' + B)AC + AB(C' + C)$$
$$= BC + AC + AB$$

1-bit Adder

module: Basic unit of description

One line comment

```
// This is a  behavioral module of 1-bit adder
module fulladd (Cin, x, y, s, Cout);
    input Cin, x, y;
    output s, Cout;

    assign s = x ^ y ^ Cin;
    assign Cout = (x & y) | (x & Cin) | (y & Cin);

endmodule
```

I/O ports with which the module communicates externally with other modules

Continuous assignment: the RHS is evaluated again every time the value of an operand changes. The LHS is assigned that value.

This is an example of **Behavioral Description**

4-bit Adder **Structural Description**


```
/* This is a  structural module of 1-bit adder
*/
module adder4 (carryin, X, Y, S, carryout);
    input carryin;
    input [3:0] X, Y;
    output [3:0] S;
    output carryout;
    wire [3:1] C;

    fulladd stage0 (carryin, X[0], Y[0], S[0], C[1]);
    fulladd stage1 (C[1], X[1], Y[1], S[1], C[2]);
    fulladd stage2 (C[2], X[2], Y[2], S[2], C[3]);
    fulladd stage3 (C[3], X[3], Y[3], S[3], carryout);

endmodule
```

Alternative N-bit Adder

```
module addern #(parameter n=32) (carryin, X, Y, S, carryout);
    input carryin;
    input [n-1:0] X, Y;
    output [n-1:0] S;
    output carryout;
    reg [n-1:0] S;
    reg carryout;
    reg [n:0] C;
    integer k;

    always @(X, Y, carryin)
    begin
        C[0] = carryin;
        for (k = 0; k < n; k = k+1)
        begin
            S[k] = X[k] ^ Y[k] ^ C[k];
            C[k+1] = (X[k] & Y[k]) | (X[k] & C[k]) | (Y[k] & C[k]);
        end
        carryout = C[n];
    end
endmodule
```

n=32 sets default value for parameter n

reg is a data type that indicates a driver that can store a value.
If a signal is not ***reg***, it is a ***wire***

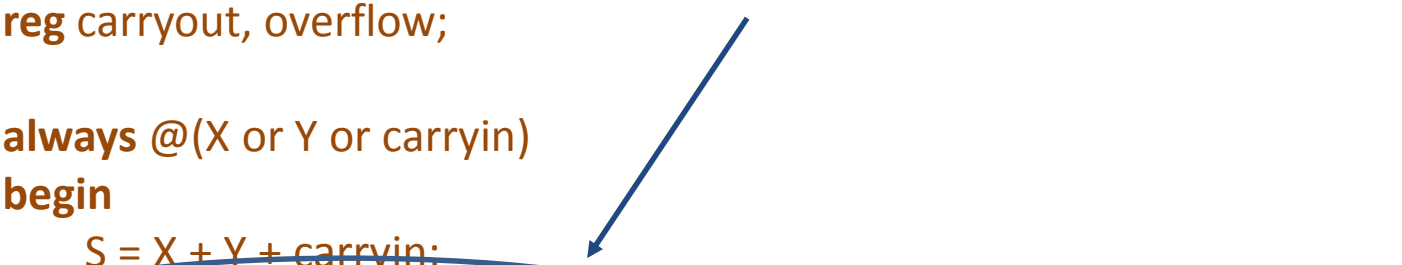
An **always** statement is only triggered at (@) the condition in the parenthesis (**sensitivity list**).
We execute the **always** statement only when *X* or *Y* o *carryin* change

Behavioral Description

Another one

```
module addern #(paramtere n=32) (carryin, X, Y, S, carryout, overflow);
    input carryin;
    input [n-1:0] X, Y;
    output [n-1:0] S;
    output carryout, overflow;
    reg [n-1:0] S;
    reg carryout, overflow;

    always @(X or Y or carryin)
    begin
        S = X + Y + carryin;
        carryout = (X[n-1] & Y[n-1]) | (X[n-1] & ~S[n-1]) | (Y[n-1] & ~S[n-1]);
        overflow = carryout ^ X[n-1] ^ Y[n-1] ^ S[n-1];
    end

endmodule
```

S, X, Y are arrays



Behavioral Description

Language Elements and Expressions

Numbers in Verilog

- Integer numbers
  - Simple decimal notation
    - 32, -15
  - Base format notation : *[size in bits]'base value*
  - *Base* is one of d or D (decimal), o or O (octal), b or B (binary), h or H (hex). Use s for signed.
    - 5'O37 // 5 bit octal number 37
    - 4'D2 // 4 bit decimal number 2
    - 4'B1x_01 // 4 bit binary number 1x01.
    - 7'Hx // 7 bit hex xxxxxxx
    - 8'shFF // Signed hex equal to -1
    - 4'd-4 // Not legal. The sign is before the size.
    - -4'd4 // 4-bit binary number -4. The sign is before the size
    - (2+3)d'10 // Not legal. Size cannot be an expression
  - The underscore character (_) is legal anywhere in a number except as the first character, where it is ignored
  - When used in a number, the question mark (?) character is the Verilog alternative for the z character.

Numbers in Verilog

- Real numbers
  - Decimal notation
    - 32.0, -11.06
    - 2. // Not legal. Must have a digit on either side of decimal
  - Scientific notation
    - 23.51e2 // 2351.0
    - 23_5.1e1 // 2351.0
    - 5E-4

Strings in Verilog

- A string is a sequence of chars within double quotes
  - “Verilog is cool!”

Four-valued Data

- Verilog's nets and registers hold four-valued data

- 0, 1
  - Obvious
- Z (high impedance)
  - Output of an undriven tri-state driver
  - Models case where nothing is setting a wire's value
- X (unknown or undecided)
  - Models when the simulator can't decide the value
  - Initial state of registers
  - When a wire is being driven to 0 and 1 simultaneously
  - Output of a gate with Z inputs

Two Main Data Types

- ***Wires*** represent connections between things
  - Do not hold their value
  - Take their value from a driver such as a gate or other module
  - A signal is wire by default

- ***Regs*** represent data storage
  - Behave exactly like memory in a computer
  - Hold their value until explicitly assigned in an *initial* or *always* block
  - Can be used to model latches, flip-flops, etc., but do not correspond exactly
  - Shared variables with all their attendant problems

Nets and Registers

- Wires and registers can be bits, vectors, and arrays

```
wire a;                    // Simple wire
wire [15:0] dbus;          // 16-bit bus
wire #(5,4,8) b;           // Wire with rise delay 5, fall
   delay 4, and turn-off delay (hiZ) 8
reg [-1:4] vec;            // Six-bit register
integer imem[0:1023];  // Array of 1024 integers
reg [31:0] dcache[0:63];  // A 32-bit mem with 64
   entries
```

Operators

- Arithmetic operators
  - X = (A+B)-C // binary plus and minus
  - X = -Y // unary minus
  - X = Y*Z // multiplication
  - X = A/B // division
  - X = A%B // modulus is the remainder with
    // the sign of the dividend.
    // 7%4 = 3, -7%4 = -3, 7%-4 = 3
  - X = A**B // exponent (Only in Verilog 2001)

Operators

- Relational operators
  - The result is 0 (false), 1 (true), or x (either operand has an x or z bit)
    - 23 > 45 // false (value 0)
    - 52 < 8'hxFF // result is x
    - 'b1000 >= 'b01110 // false. Smaller sized operand is zero-filled

Operators

- Equality operators.
  - 2'b10 == 4'b0010 // true
  - 'b11x0 == 'b11x0 // unknown because there is a bit in in either operand which is x (or z)
  - 'b11x0 === 'b11x0 // true. In case equality, x and z are compared strictly as values

Operators

- Logical operators.
  - A && B // logical AND
  - A || B // logical OR
  - !A // logical NOT

Operators

- Bit-wise operators
  - ~A (unary negation)
  - & (binary and)
  - | (binary or)
  - ^ (binary xor)
  - ~^ (binary XNOR)

AND (&)

	0	1	X	Z
0	0	0	0	0
1	0	1	X	X
X	0	X	X	X
Z	0	X	X	X

OR (|)

	0	1	X	Z
0	0	1	X	X
1	1	1	1	1
X	X	1	X	X
Z	X	1	X	X

Operators

- Reduction operators
  - A = ‘b0110, B = ‘b0100, C = ‘b0x1
  - &B is 0 // logical AND of all bits
  - |B is 1 // logical OR of all bits
  - ~&A is 1 // logical NAND of all bits
  - ^B is 1 // logical XOR of all bits
  - ~^B is 0 // logical XNOR of all bits
  - &C is 0 // at least a bit is 0
  - |C is 1 // at least a bit is 1
  - ^C is x // if any bit is x or z, result is x

Operators

- Shift operators
  - A >> B // logical right shift
  - A << B // logical left shift
  - A >>> B // right shift for signed numbers (Only Verilog 2001)
  - A <<< B // left shift for signed numbers (Only Verilog 2001)

- Conditional operator
  - cond_expr? expr1 : expr2

Operators

- Concatenation and Replication operators
  - wire [7:0] Dbus
  - wire [11:0] Abus
  - assign Dbus = {Dbus[3:0], Dbus[7:4]} // concatenation: swap nibbles
  - Abus ={3{4'b1011}} // 12'b1011_1011_1011

Two ways to express hardware in Verilog:
Structural and Behavioral Models

Two Main Components of Verilog

- Concurrent, event-triggered processes (behavioral)
  - *Initial* and *Always* blocks
  - Imperative code that can perform standard data manipulation tasks (assignment, if-then, case)
  - Processes run until they delay for a period of time or wait for a triggering event
- Structure (structural)
  - Verilog program built from modules with I/O interfaces
  - Modules may contain instances of other modules
  - Modules contain local signals, etc.
  - Module configuration is static and all run concurrently
- High level modules typically structural
- Low level modules (leaves) typically behavioral

Structural Modeling

- When Verilog was first developed (1984) most logic simulators operated on netlists
- Netlist: list of gates and how they're connected
- A natural representation of a digital logic circuit
- **Not** the most convenient way to express test benches

Behavioral Modeling

- A much easier way to write testbenches
- Also good for more abstract models of circuits
  - Easier to write
  - Simulates faster
- More flexible
- Provides sequencing
- Verilog succeeded in part because it allowed both the model and the testbench to be described together

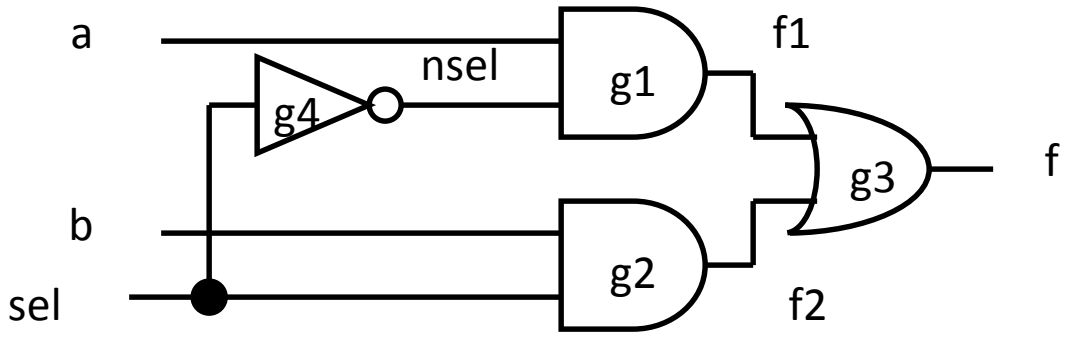
Multiplexer Built From Primitives

```
module mux(f, a, b, sel);
  output f;
  input a, b, sel;

  and g1(f1, a, nsel),
      g2(f2, b, sel);
  or  g3(f, f1, f2);
  not g4(nsel, sel);

endmodule
```

Identifiers not explicitly defined default to wires

Module may contain structure: instances of primitives and other modules

Predefined module types

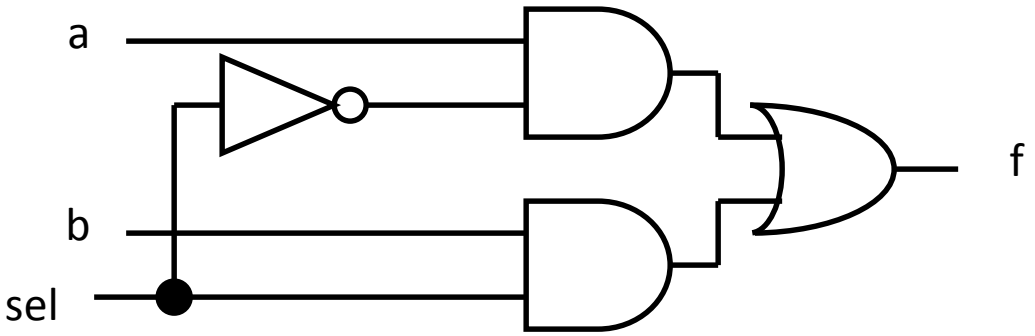


Multiplexer Built With Always

```
module mux(f, a, b, sel);
output f;
input a, b, sel;
reg f;

always @(a, b, sel)
  if (sel)
      f = b;
  else
      f = a;

endmodule
```

Modules may contain one or more *always* blocks

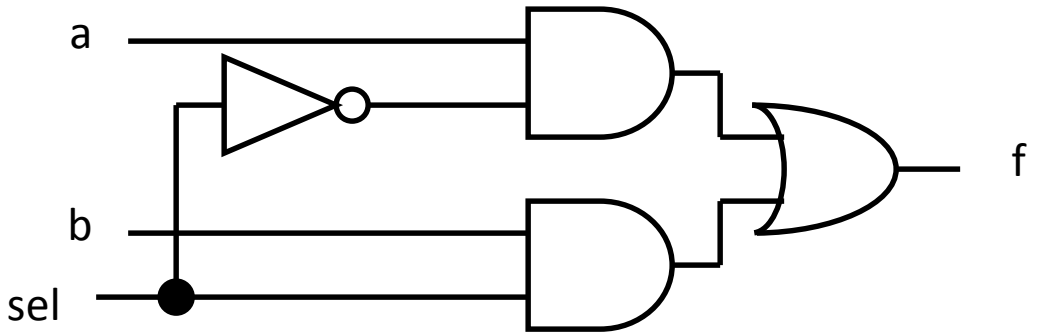
Sensitivity list contains signals whose change triggers the execution of the block



Mux with Continuous Assignment

```
module mux(f, a, b, sel);
output f;
input a, b, sel;

assign f = sel ? b : a;

endmodule
```

LHS is always set to the value on the RHS
Any change on the right causes reevaluation



Structural Modeling in more detail

Modules and Instances

- Basic structure of a Verilog module:

```
module mymod(port1, port2, … portN);
 output port1;
 output [3:0] port2;
 input [2:0] portN;
…
endmodule
```

Verilog convention lists outputs first. This is not necessary

Modules and Instances

- Verilog 2001 allows port direction and data type in the port list of modules as shown in the example below

```
module mymod(output port1,
                    output [3:0] port2,
                     …
                    input [2:0] portN);

…
endmodule
```

Instantiating a Module

- Instances of

```
module mymod(y, a, b);
```

- look like

```
mymod mm1(y1, a1, b1);              // Connect-by-position
mymod (y2, a1, b1),
      (y3, a2, b2);                 // Instance names omitted
mymod mm2(.a(a2), .b(b2), .y(c2));  // Connect-by-name
```

Gate-level Primitives

- Verilog provides the following keywords for gate level modeling:

and	nand	logical AND/NAND
or	nor	logical OR/NOR
xor	xnor	logical XOR/XNOR
buf	not	buffer/inverter
bufif0	notif0	Tristate buf with low enable
bifif1	notif1	Tristate buf with high enable

Switch-level Primitives

- Verilog also provides mechanisms for modeling CMOS transistors that behave like switches
- A more detailed modeling scheme that can catch some additional electrical problems when transistors are used in this way
- Rarely used because circuits generally aren't built this way
- More seriously, model is not detailed enough to catch many of the problems
- These circuits are usually simulated using SPICE-like simulators based on nonlinear differential equation solvers

Behavioral Modeling in more detail

Types of Assignments in Verilog

- **Continuous** and **Procedural** Assignments
- **Continuous assignments** model combinational behavior only
  - They assign a value to a wire (never to a reg)
  - *assign LHS_target = RHS_expression*
  - The continuous statement executes every time an event happens in the RHS expression
  - The expression is evaluated
  - If the result is different, the new value is assigned to the LHS target

Continuous Assignment

- Convenient for logical or data path specifications

Define bus widths

```
wire [8:0] sum;
wire [7:0] a, b;
wire carryin;

assign sum = a + b + carryin;
```

Continuous assignment: permanently sets the value of sum to be a+b+carryin Recomputed when a, b, or carryin changes

Types of Assignments in Verilog

- **Procedural assignments** model combinational and sequential behavior
- They appear only within an **initial statement** or an **always statement**
- Two different types of procedural assignments
  - Blocking
  - Non-blocking

Initial and Always Blocks

- Basic components for behavioral modeling

```
initial
  begin
    … imperative statements …
  end
```

Runs when simulation starts
Terminates when control reaches the end
Good for providing stimulus in testbenches

```
always
  begin
    … imperative statements …
  end
```

Runs when simulation starts
Restarts when control reaches the end
Good for modeling/specifying hardware

Timing Controls

- Two forms of timing control:
  - Delay control
  - Event control

Delay Controls

- **Delay** control is of the form

  # delay <procedural statement>

  It specifies the time units from the time the statement is initially encountered to the time it is executed

e.g.

#2 input = 4'b0101; // wait for 2 units, and then make the assignment

input = #1 x // assign to *input* the value that *x* will have after 1 time unit. Different than #1 input = x

Event Controls

- **Edge-triggered** event control is of the form

  @ event <procedural statement>

  e.g. @ (posedge clk) curr_state = next_state

  @ (X or Y) A <= 0; // when X or Y change

- **Level-triggered** event control is of the form

  wait (condition) <procedural statement>

  The statement executes only if the condition is true, else it waits until the condition becomes true.

  e.g. wait(DataReady);

  Data = Bus

Initial and Always

- Run until they encounter a delay

```
initial begin
  #10 a = 1; b = 0;
  #10 a = 0; b = 1;
end
```

- or a wait for an event

```
always @(posedge clk) q = d;
always
   begin
        wait(i); a = 0; wait(~i); a = 1;
   end
```

Blocking Procedural Assignment

- Inside an initial or always block:

```
sum = a + b + cin;
```

- Just like in C: RHS evaluated and assigned to LHS before next statement executes

- RHS may contain wires and regs
  - Two possible sources for data
- LHS must be a reg
  - Primitives or cont. assignment may set wire values

Imperative Statements

```
if (select == 1)      y = a;
else                  y = b;

case (op)
  2'b00: y = a + b;
  2'b01: y = a – b;
  2'b10: y = a ^ b;
  default: y = 'hxxxx;
endcase
```

For Loops

- A increasing sequence of values on an output

```
reg [3:0] i, output;

for ( i = 0 ; i <= 15 ; i = i + 1 ) begin
  output = i;
  #10;
end
```

While Loops

- A increasing sequence of values on an output

```
reg [3:0] i, output;

i = 0;
while (i <= 15) begin
  output = i;
  #10 i = i + 1;
end
```

Blocking vs. Non Blocking Assignments

- Verilog has two types of procedural assignments

- Fundamental problem:
  - In a synchronous system, all flip-flops sample simultaneously
  - In Verilog, always @(posedge clk) blocks run in some undefined sequence

A Flawed Shift Register

- This doesn't work as you expect:

```
reg d1, d2, d3, d4;

always @(posedge clk) d2 = d1;
always @(posedge clk) d3 = d2;
always @(posedge clk) d4 = d3;
```

- These run in some order, but you don't know which

Non-blocking Assignments

- This version does work:

```
reg d1, d2, d3, d4;

always @(posedge clk) d2 <= d1;
always @(posedge clk) d3 <= d2;
always @(posedge clk) d4 <= d3;
```

Nonblocking rule: RHS evaluated when assignment runs

LHS updated only after all events for the current instant have run. It runs after a *Delta* time

Nonblocking Can Behave Oddly

- A sequence of nonblocking assignments don't communicate

```
a = 1;
b = a;
c = b;
```

Blocking assignment:
a = b = c = 1

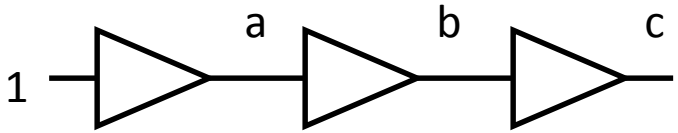
```
a <= 1;
b <= a;
c <= b;
```

Nonblocking assignment after δ time:
a = 1
b = old value of a
c = old value of b

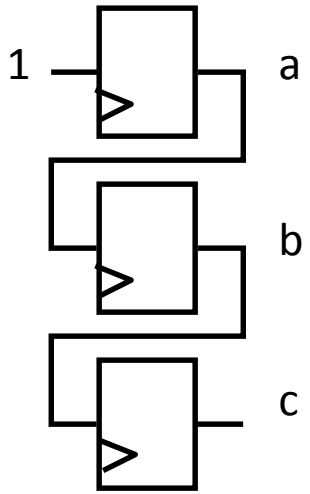
Nonblocking Looks Like Latches

- RHS of nonblocking taken from latches
- RHS of blocking taken from wires

```
a = 1;
b = a;
c = b;
```



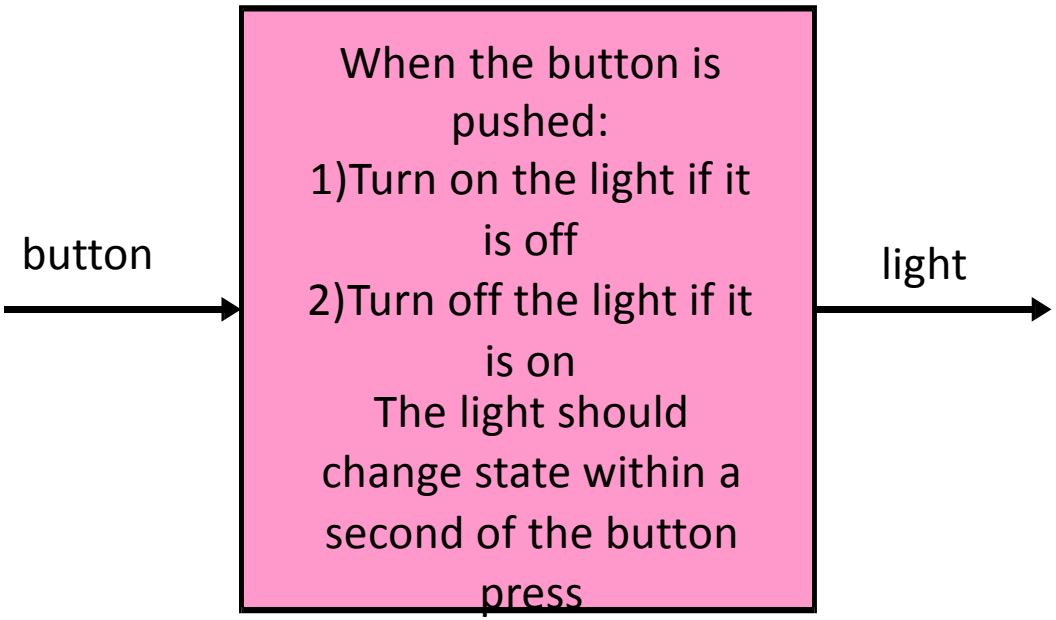

```
a <= 1;
b <= a;
c <= b;
```



Sequential Logic and Finite State Machines in Verilog

Why we need sequential logic?

What if you were given the following design specification:

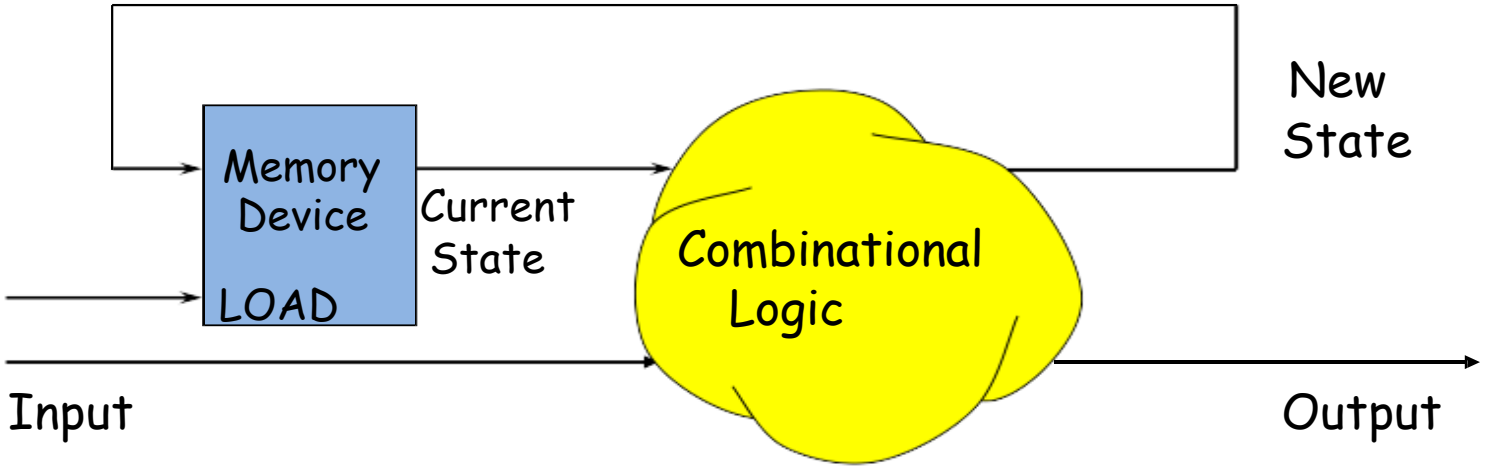


What makes this circuit different from those we've discussed before?

1. "State" – i.e. the circuit has memory
2. The output was changed by a input "event" (pushing a button) rather than an input "value"

Digital State

One model of what we'd like to build



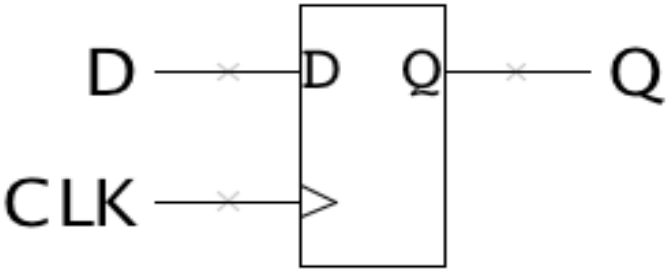
Plan: Build a Sequential Circuit with stored digital STATE –

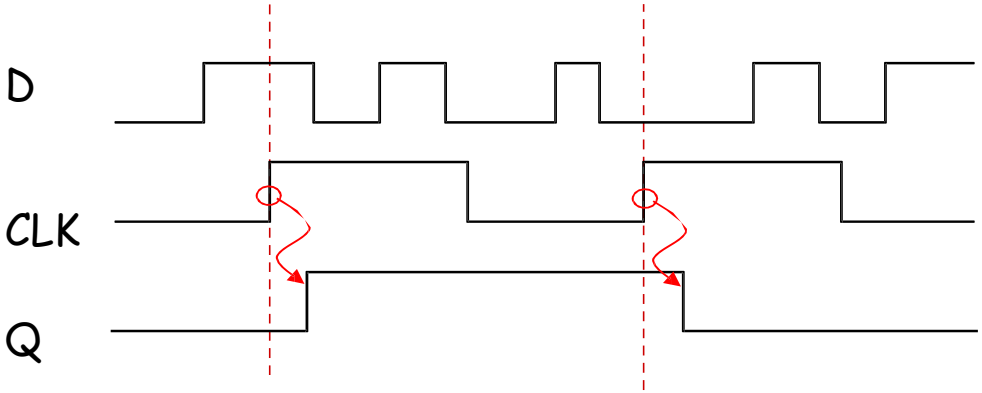
- Memory stores CURRENT state
- Combinational Logic computes
  - NEXT state (from input, current state)
  - OUTPUT bit (from input, current state)
- State changes on LOAD control input

If Output depends on Input and current state, circuit is called a **Mealy** machine.
If Output depends only on the current state, circuit is called a **Moore** machine.

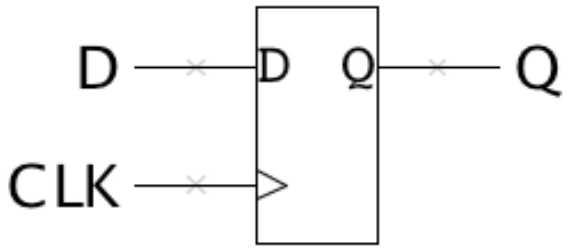
Our building block: the D FF

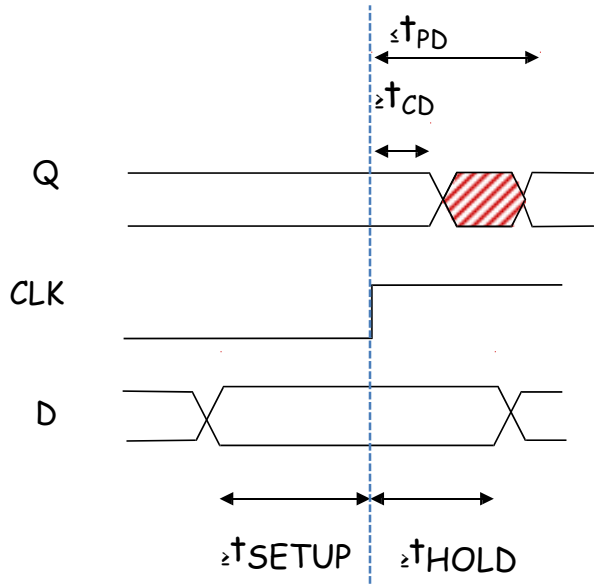
The edge-triggered D register: on the rising edge of CLK, the value of D is saved in the register and then shortly afterwards appears on Q.





D-Register Timing





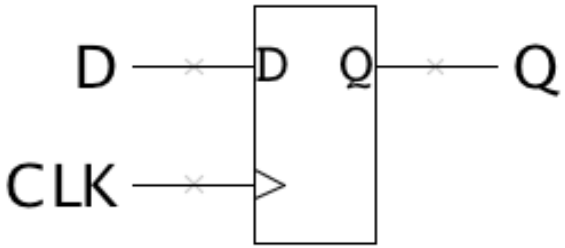
t_{PD}: maximum propagation delay, CLK → Q

t_{CD}: minimum contamination delay, CLK →Q

t_{SETUP}: setup time
How long D must be stable before the rising edge of CLK

t_{HOLD}: hold time
How long D must be stable after the rising edge of CLK

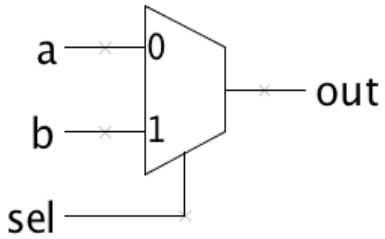
The Sequential always Block

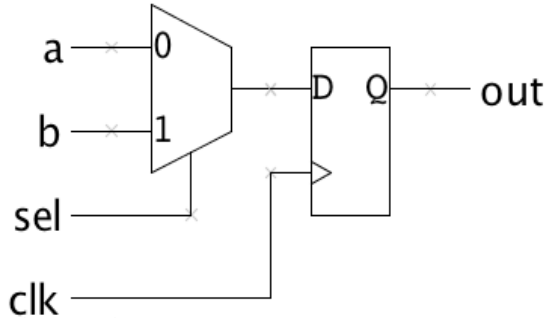



```
module dff (input D, CLK, output reg Q);
always @(posedge clk)
   begin
       Q<=D;
    end
endmodule;
```

```
module CL(input a,b, sel,
              output reg out);
always @(*)
     begin
          if (sel) out = b;
          else out = a;
    end
endmodule;
```

```
module SL(input a,b, sel,
              output reg out);
always @(posedge  clk)
     begin
          if (sel) out <= b;
          else out <= a;
    end
endmodule;
```

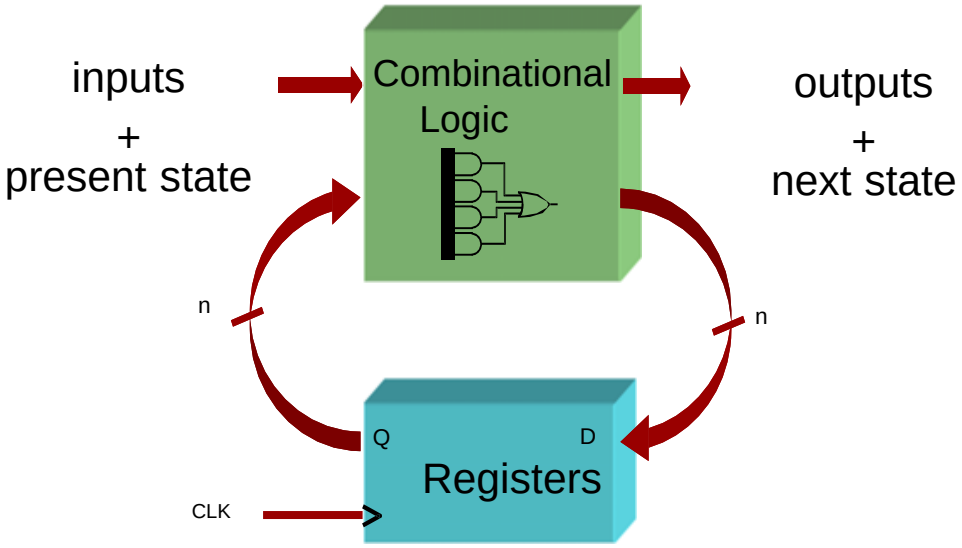




Finite State Machines and Verilog

Finite State Machines

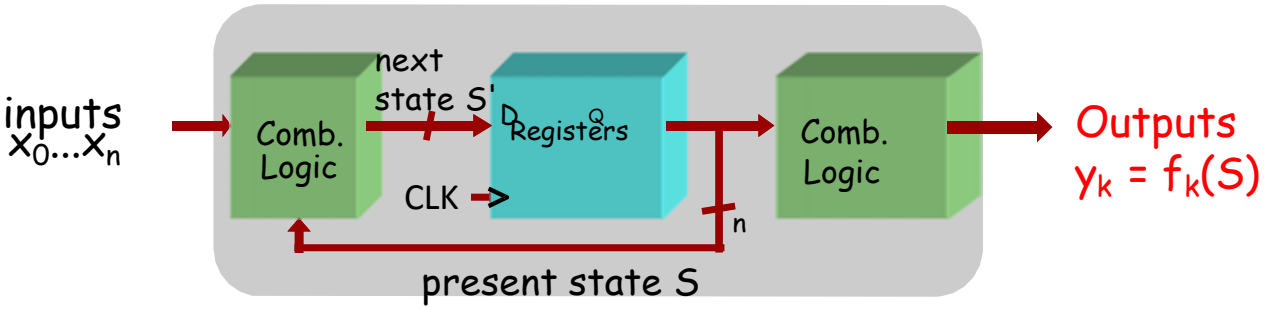
- Finite State Machines (FSMs) are a useful abstraction for sequential circuits with centralized "states" of operation
- At each clock edge, combinational logic computes outputs and next state as a function of inputs and present state



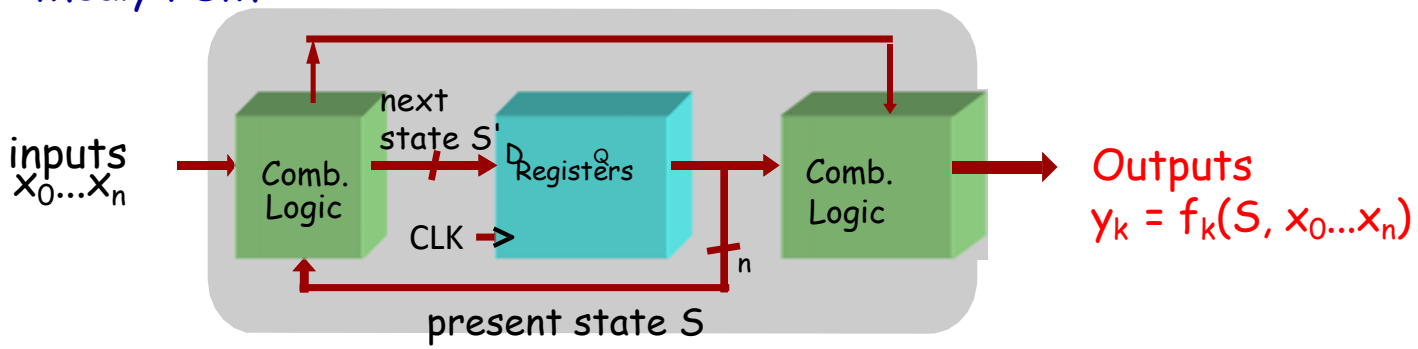
Two types of Finite State Machines

Moore and Mealy FSMs : different output generation

- Moore FSM:

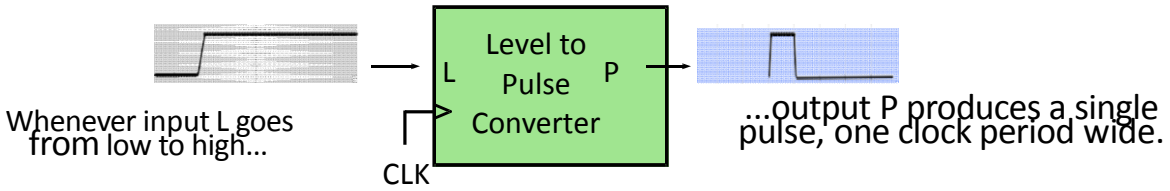


- Mealy FSM:



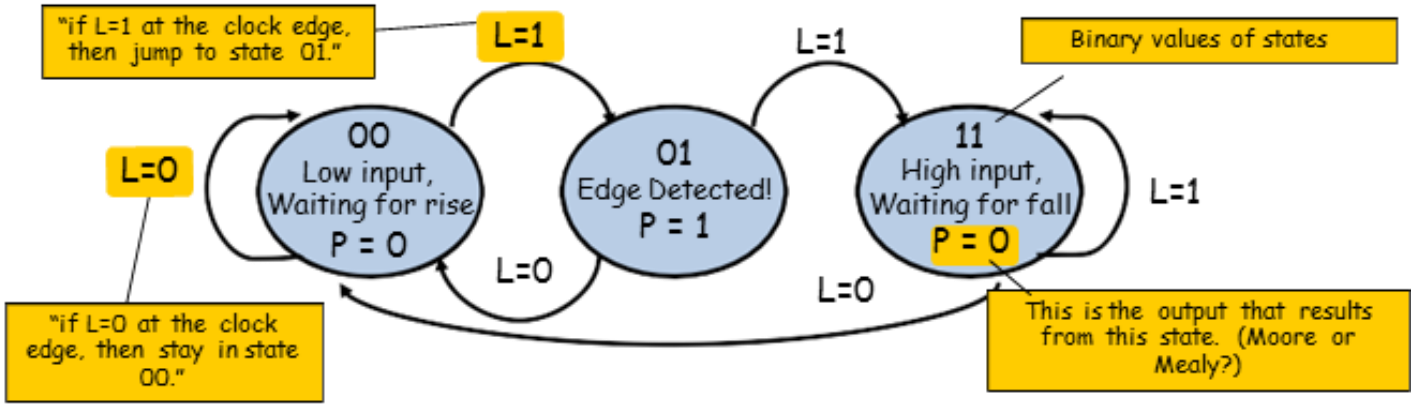
Design Example: Level-to-Pulse

- A **level-to-pulse converter** produces a single- cycle pulse each time its input goes high.
- It's a synchronous rising-edge detector.
- Sample uses:
  - Buttons and switches pressed by humans for arbitrary periods of time
  - Single-cycle enable signals for counters



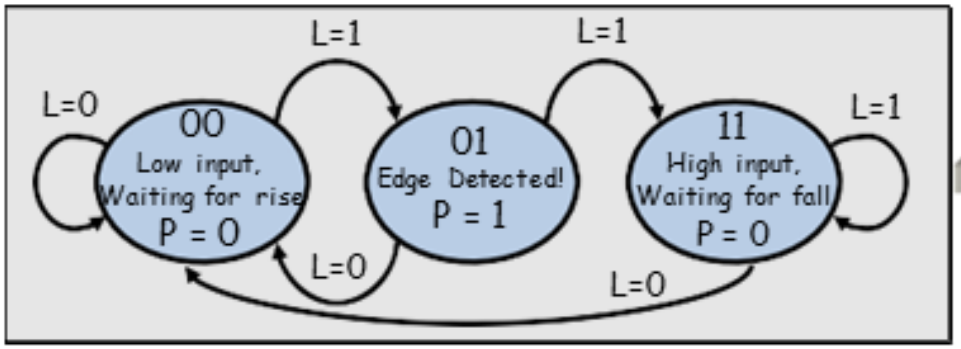
Step 1: State Transition Diagram

- State transition diagram is a useful FSM representation and design aid:



Step 2: Logic Derivation

Transition diagram is readily converted to a state transition table (just a truth table)



Current State		In	Next State		Out
S_1	S_0	L	S_1^+	S_0^+	P
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	1
1	1	0	0	0	0
1	1	1	1	1	0

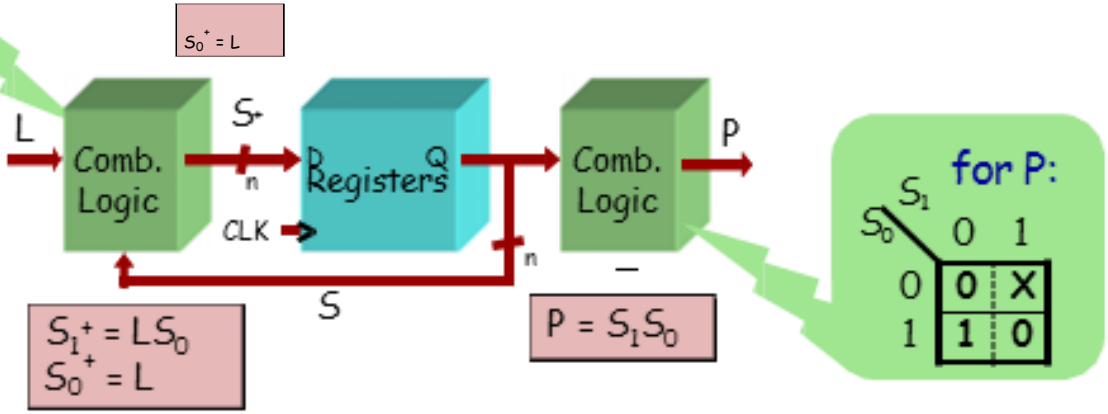
- Combinational logic may be derived using Karnaugh maps

S_1S_0 for S_1^+:

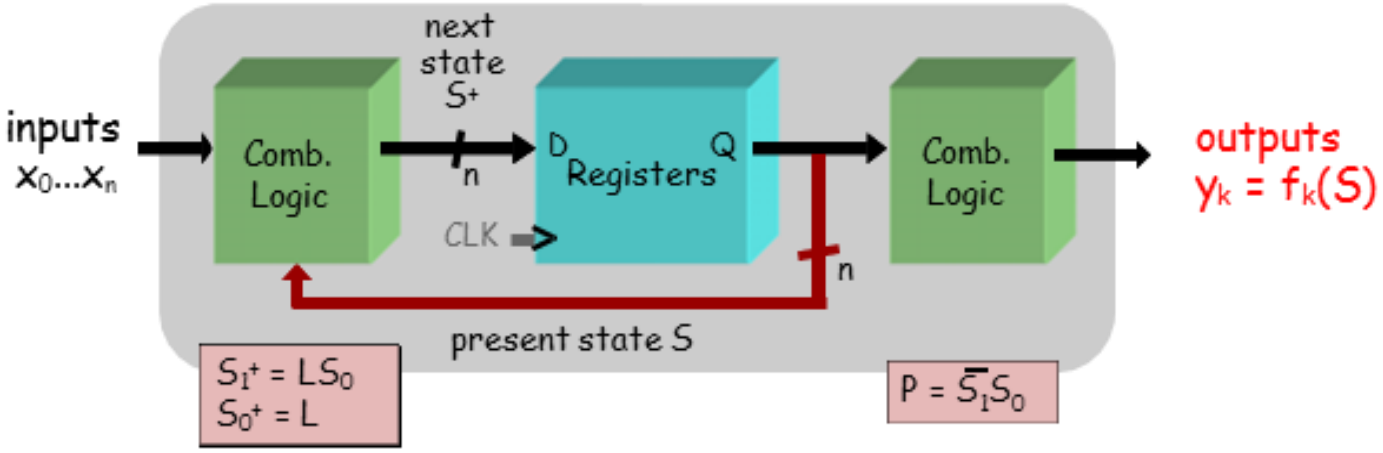
L \ S_1S_0	00	01	11	10
0	0	0	0	X
1	0	1	1	X

S_1S_0 for S_0^+:

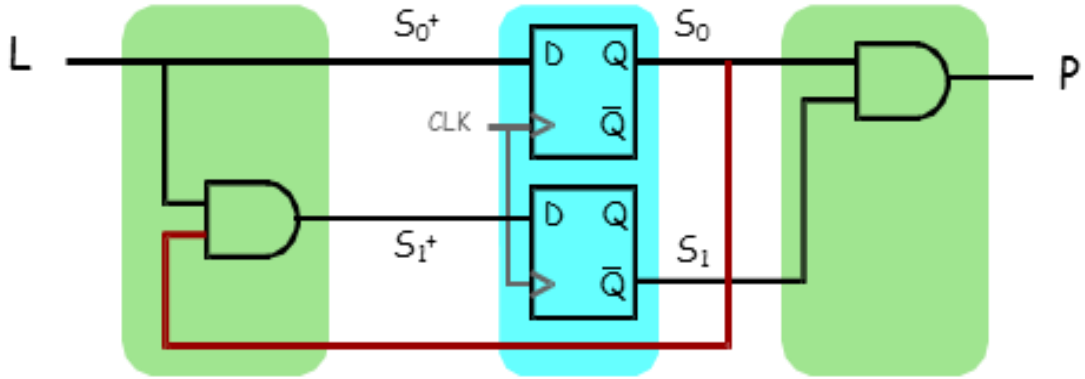
L \ S_1S_0	00	01	11	10
0	0	0	0	X
1	1	1	1	X



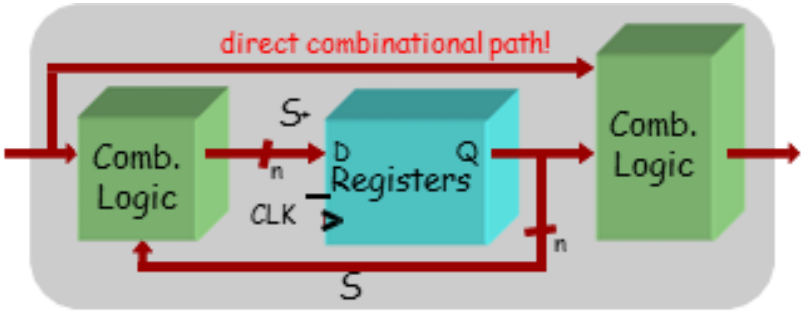
Moore Level-to-Pulse Converter



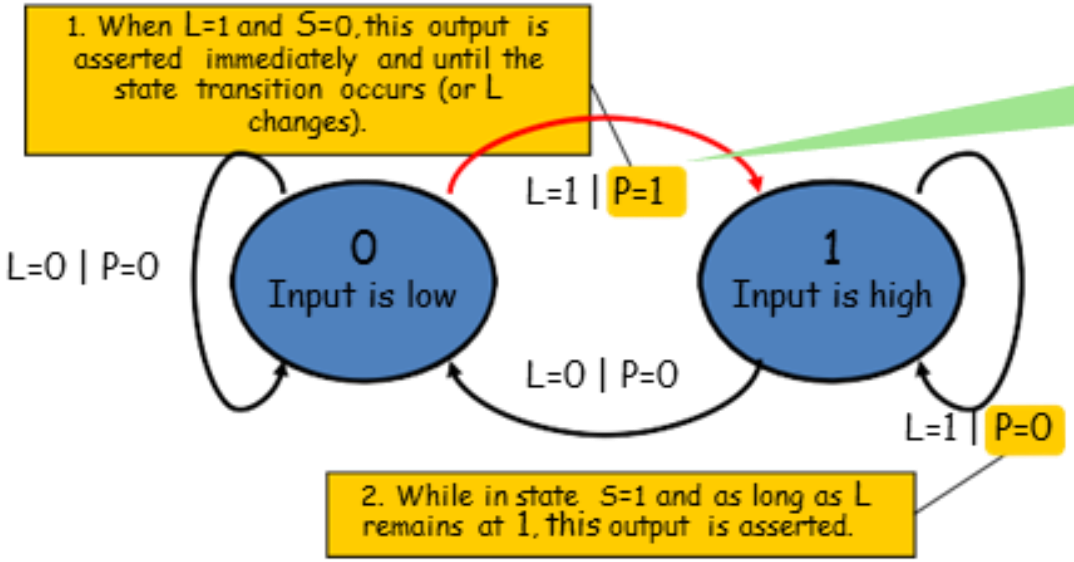
Moore FSM circuit implementation of level-to-pulse converter:

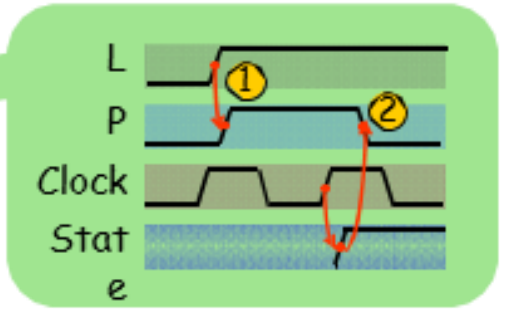


Design of a Mealy Level-to-Pulse



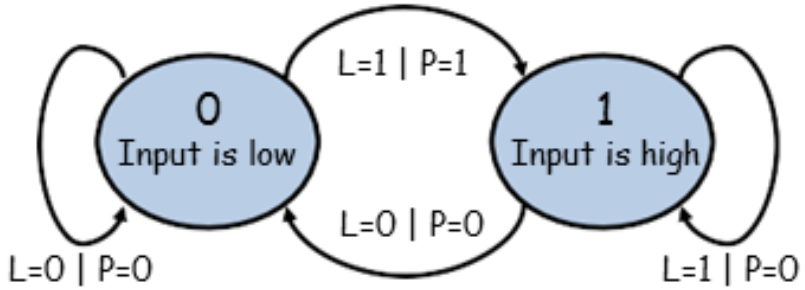
- Since outputs are determined by state and inputs, Mealy FSMs may need fewer states than Moore FSM implementations





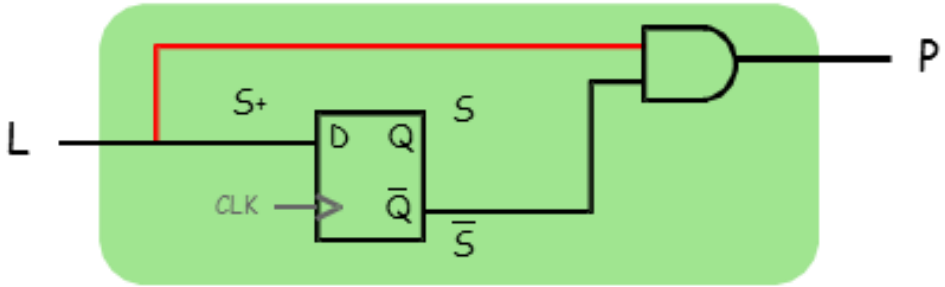
Output transitions immediately. State transitions at the clock edge.

Design of a Mealy Level-to-Pulse



Pres. State	In	Next State	Out
S	L	S+	P
0	0	0	0
0	1	1	1
1	0	0	0
1	1	1	0

Mealy FSM circuit implementation of level-to-pulse converter:



- FSM's state simply remembers the previous value of L
- Circuit benefits from the Mealy FSM's implicit single-cycle assertion of outputs during state transitions

Second FSM Example

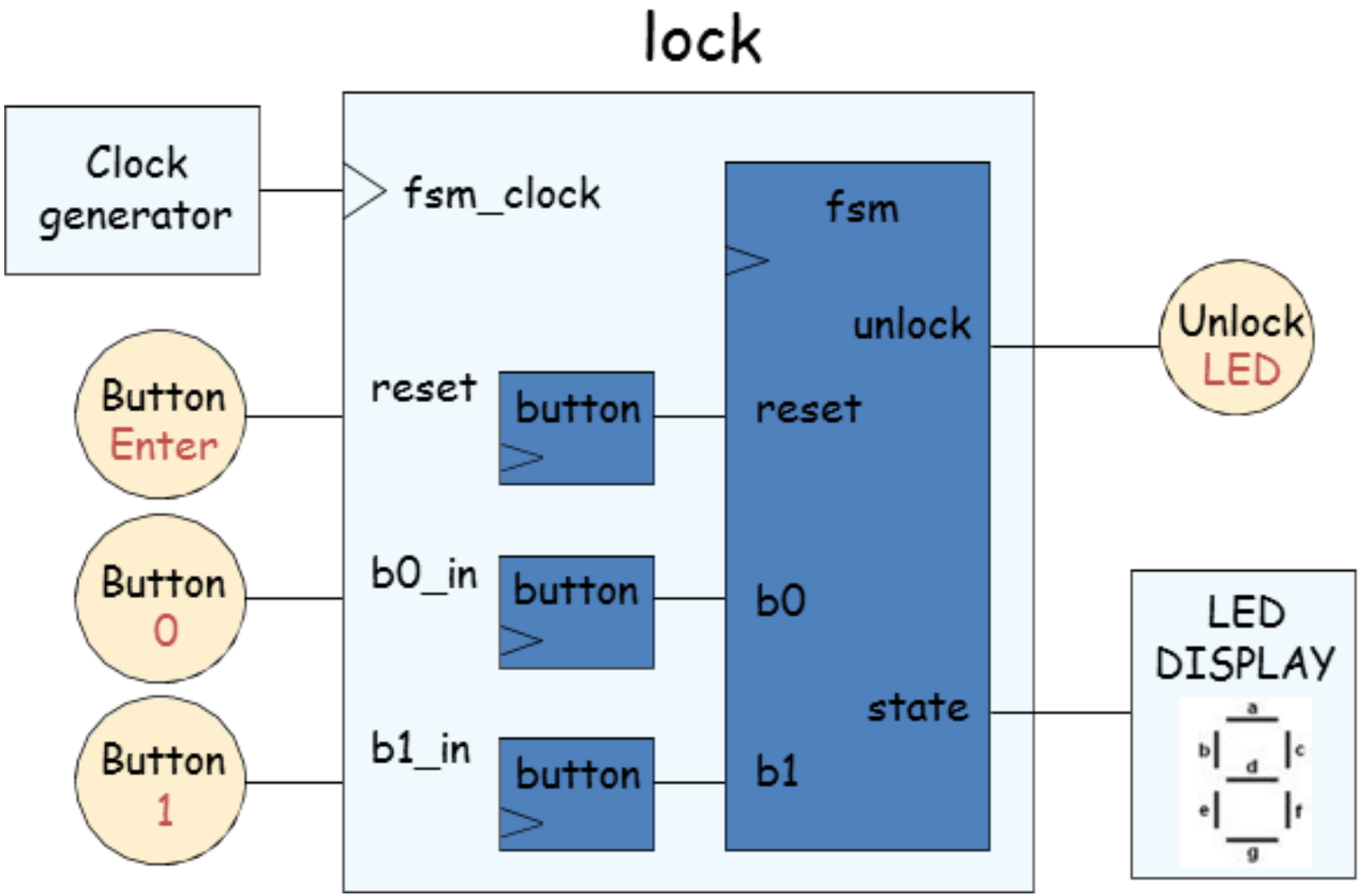
GOAL:

Build an electronic combination lock with a reset button, two number buttons (0 and 1), and an unlock output. The combination should be 01011.

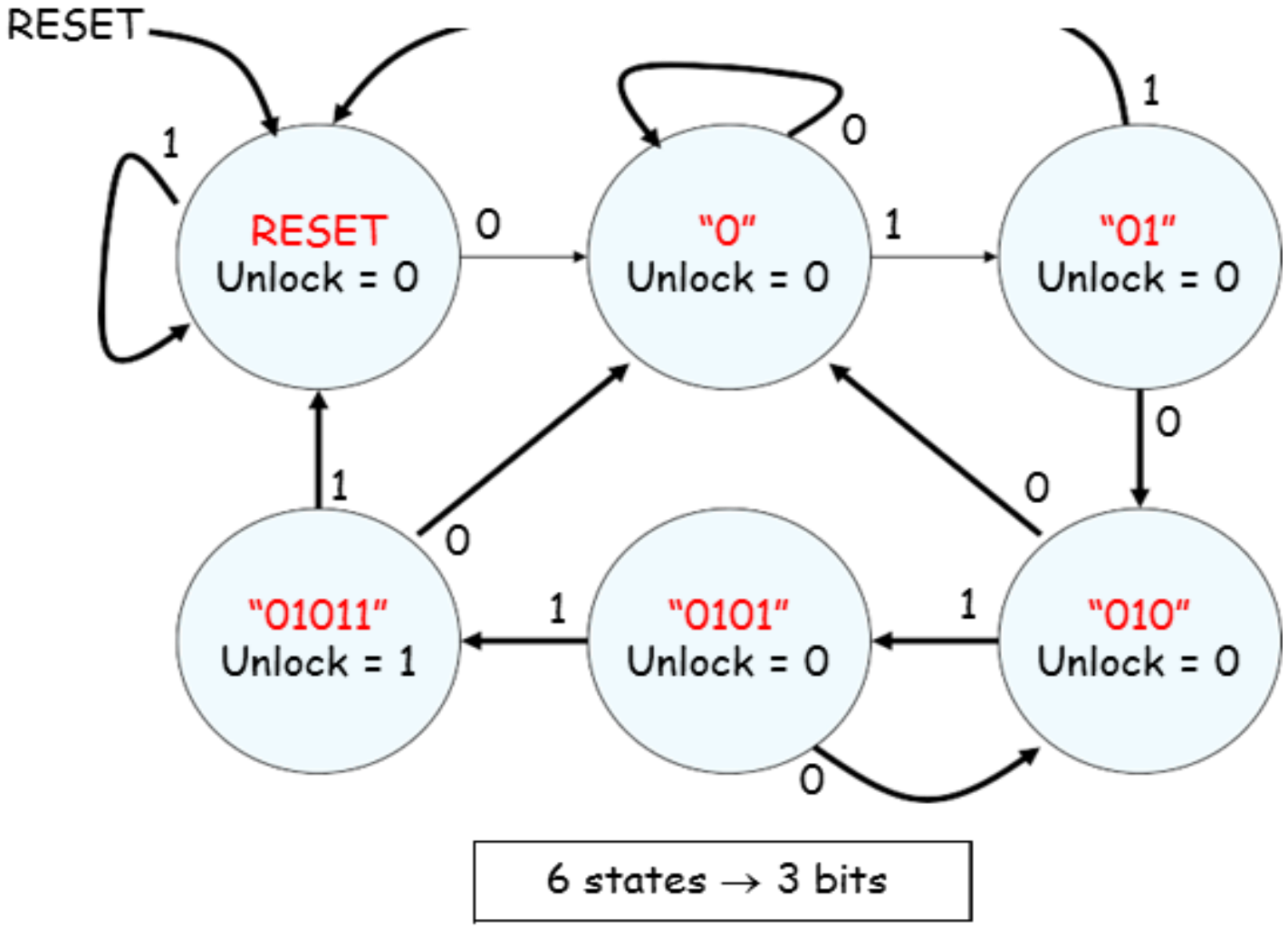
STEPS:

1. Design lock FSM (block diagram, state transitions)
2. Write Verilog module(s) for FSM

Step 1A: Block Diagram

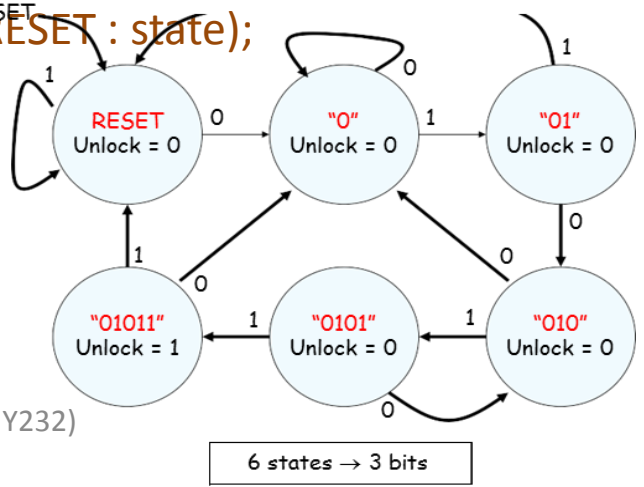


Step 1B: State transition diagram



Step 2. Verilog Implementation of the FSM

```
module lock(input clk, reset, b0, b1, output out);
wire reset;
parameter S_RESET = 0; parameter S_0= 1;          // state assignments
parameter S_01 = 2;  parameter S_010 = 3;
parameter S_0101 = 4;  parameter S_01011 = 5;
always @(*)
      begin                    // First always computes next state
          case (state)
            S_RESET: next_state = b0 ? S_0 :  (b1 ? S_RESET : state);
            S_0: next_state = b0 ? S_0 :  (b1 ? S_01 : state);
            S_01: next_state = b0 ? S_010 :  (b1 ? S_RESET : state);
            S_010: next_state = b0 ? S_0 :  (b1 ? S_0101 : state);
            S_0101: next_state = b0 ? S_010 :  (b1 ? S_01011 : state);
            S_01011: next_state = b0 ? S_0 :  (b1 ? S_RESET : state);
            default: next_state = S_RESET;
          endcase
    end   // always
```



Step 2. Verilog Implementation of the FSM

```
always @(posedge clk)      // Second always computes next state
  if (reset == 1'b1)
    state <= S_RESET;
  else
   state  <=next_state;


 assign out = (state == S_01011);
endmodule
```

Modeling FSMs Behaviorally

- There are many ways to do it:

1. Define the next-state logic combinationally and define the state-holding latches explicitly

1. Define the behavior in a single always @(posedge clk) block

Testbenches

Writing Testbenches

```
module test;
reg a, b, sel;

mux m(y, a, b, sel);

initial
   begin
      $monitor($time, "a = %b, b=%b, sel=%b, y=%b",
               a, b, sel, y);
      a = 0; b= 0; sel = 0;
      #10 a = 1;
      #10 sel = 1;
      #10 b = 1;
   end
endmodule
```

Inputs to device under test

Device under test (DUT)

$monitor is a built-in event driven "printf"

Stimulus generated by sequence of assignments and delays

Writing Testbenches

```
module first_counter_tb();
// Declare inputs as regs and outputs as wires
 reg clock, reset, enable;
wire [3:0] counter_out;

//Initialize all variables
initial
     begin
       $display ("time\t clk reset enable counter");
       $monitor ("%g\t %b %b %b %b",
       $time, clock, reset, enable, counter_out);
       clock = 1; // initial value of clock
       reset = 0; // initial value of reset
       enable = 0; // initial value of enable
       #5 reset = 1; // Assert the reset
       #10 reset = 0; // De-assert the reset
       #10 enable = 1; // Assert enable
       #100 enable = 0; // De-assert enable
      #5 $finish; // Terminate simulation
    end

// Clock generator
always begin
      #5 clock = ~clock; // Toggle clock every 5 ticks
end

// Connect DUT to test bench
first_counter U_counter ( clock, reset,  enable, counter_out );

 end module
```

Simulating Verilog

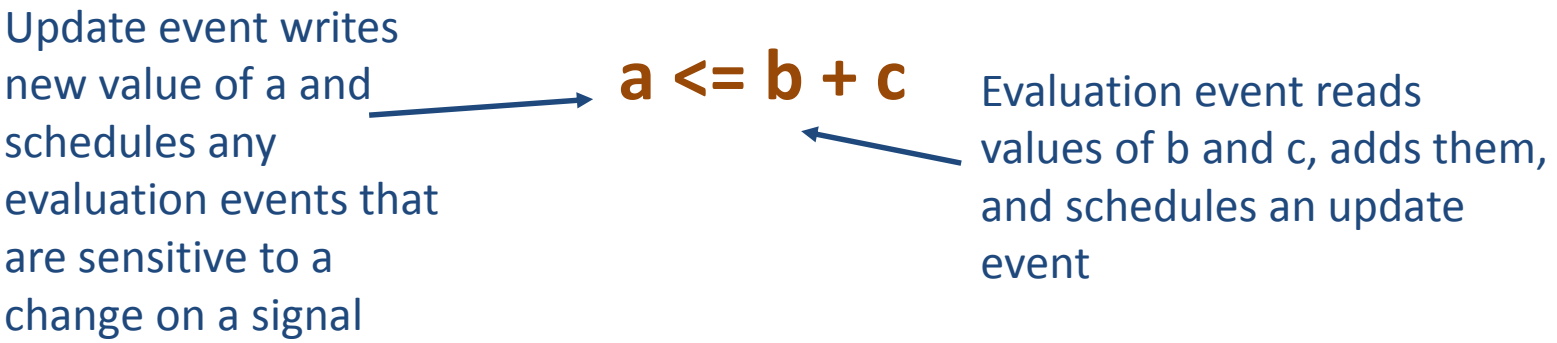
Simulation Behavior

- Scheduled using an event queue
- Non-preemptive, no priorities
- A process must explicitly request a context switch
- Events at a particular time unordered

- Scheduler runs each event at the current time, possibly scheduling more as a result

Two Types of Events

- **Evaluation** events compute functions of inputs
- **Update** events change outputs
- Split necessary for delays, nonblocking assignments, etc.



Simulation Behavior

- Concurrent processes (initial, always) run until they stop at one of the following

- #42
  - Schedule process to resume 42 time units from now
- wait(cf & of)
  - Resume when expression “cf & of” becomes true
- @(a or b or y)
  - Resume when a, b, or y changes
- @(posedge clk)
  - Resume when clk changes from 0 to 1

Simulation Behavior

- Infinite loops are possible and the simulator does not check for them
- This runs forever: no context switch allowed, so ready can never change

```
while (~ready)
  count = count + 1;
```

- Instead, use

```
wait(ready);
```

Simulation Behavior

- Race conditions abound in Verilog

- These can execute in either order: final value of a undefined:

```
always @(posedge clk) a = 0;
always @(posedge clk) a = 1;
```

Verilog and Logic Synthesis

Logic Synthesis

- Verilog is used in two ways
  - Model for discrete-event simulation
  - Specification for a logic synthesis system

- Logic synthesis converts a **subset of Verilog** language into an efficient netlist
- One of the major breakthroughs in designing logic chips in the last 20 years
- Most chips are designed using at least some logic synthesis

Logic Synthesis Tools

- Mostly commercial tools
  - Very difficult, complicated programs to write well
  - Limited market
  - Commercial products in $10k - $100k price range
- Major vendors
  - Synopsys Design Compiler, FPGA Express
  - Cadence BuildGates
  - Synplicity (FPGAs)
  - Exemplar (FPGAs)
- Academic tools
  - SIS (UC Berkeley)

Logic Synthesis

- Takes place in two stages:
- Translation of Verilog (or VHDL) source to a netlist
  - Register inference
- Optimization of the resulting netlist to improve speed and area

Logic Optimization

- Netlist optimization the critical enabling technology
- Takes a slow or large netlist and transforms it into one that implements the same function more cheaply

- Typical operations
  - Constant propagation
  - Common subexpression elimination
  - Function factoring

- Time-consuming operation
  - Can take hours for large chips

Translating Verilog into Gates

- Parts of the language easy to translate
  - Structural descriptions with primitives
    - Already a netlist
  - Continuous assignment
    - Expressions turn into little datapaths
- Behavioral statements the bigger challenge

What Can Be Translated

- Structural definitions
  - Everything
- Behavioral blocks
  - Depends on sensitivity list
  - Only when they have reasonable interpretation as combinational logic, edge, or level-sensitive latches
  - Blocks sensitive to both edges of the clock, changes on unrelated signals, etc. cannot be synthesized
- User-defined primitives
  - Primitives defined with truth tables
  - Some sequential UDPs can’t be translated (not latches or flip-flops)

What Isn't Translated

- Initial blocks
  - Used to set up initial state or describe finite testbench stimuli
  - Don't have obvious hardware component
- Delays
  - May be in the Verilog source, but are ignored by synthesizer
- A variety of other obscure language features
  - In general, things heavily dependent on discrete-event simulation semantics
  - Certain "disable" statements

Register Inference

- The main trick
- ***reg*** does not always equal latch
- **Rule:**
- **Combinational if outputs always depend exclusively on sensitivity list**
- **Sequential if outputs may also depend on previous values**

Register Inference

- Combinational:

```
reg y;
always @(a or b or sel)
  if (sel) y = a;
  else y = b;
```

Sensitive to changes on all of the variables it reads.
Important: All outputs of the combinational circuit should be specified for every possible execution path.

y is always assigned

- Sequential:

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

q only assigned at posedge clk

Register Inference

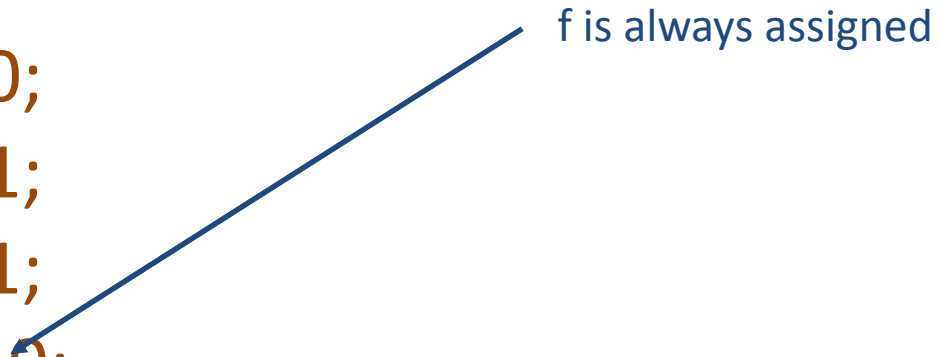
- A common mistake is not completely specifying a case statement
- This implies a latch:

```
always @(a or b)
    case ({a, b})
      2'b00 : f = 0;
      2'b01 : f = 1;
      2'b10 : f = 1;
endcase
```

Register Inference

- The solution is to always have a default case

```
always @(a or b)
   case ({a, b})
      2'b00: f = 0;
      2'b01: f = 1;
      2'b10: f = 1;
     default: f = 0;
endcase
```



Inferring Latches with Reset

- Latches and Flip-flops often have reset inputs
- Can be synchronous or asynchronous

- Asynchronous positive reset:

```
always @(posedge clk or posedge reset)
  if (reset)
      q <= 0;
  else
      q <= d;
```

Inferring Latches with Reset

- Synchronous positive reset:

```
always @(posedge clk)
  if (reset)
      q <= 0;
  else
      q <= d;
```

Simulation-synthesis Mismatches

- Many possible sources of conflict

- Synthesis ignores delays (e.g., #10), but simulation behavior can be affected by them
- Simulator models X explicitly, synthesis doesn't
- Behaviors resulting from shared-variable-like behavior of regs is not synthesized
  - always @(posedge clk) a = 1;
  - New value of *a* may be seen by other @(posedge clk) statements in simulation, never in synthesis