Blocking and Non-blocking Assignments

Introduction

- Sequential statements within procedural blocks (“always” and “initial”) can use two types of assignments:
  a) Blocking assignment
     - Uses the ‘=’ operator
  b) Non-blocking assignment
     - Uses the ‘<=’ operator

Blocking Assignment (using '=')

- Most commonly used type.
- The target of assignment gets updated before the next sequential statement in the procedural block is executed.
- A statement using blocking assignment blocks the execution of the statements following it, until it gets completed.
- Recommended style for modeling combinational logic.

Non-Blocking Assignment (using ‘<=’)

- The assignment to the target gets scheduled for the end of the simulation cycle.
  - Normally occurs at the end of the sequential block.
  - Statements subsequent to the instruction under consideration are not blocked by the assignment.
- Recommended style for modeling sequential logic.
  - Can be used to assign several ‘reg’ type variables synchronously, under the control of a common clock.

Some Rules to be Followed

- Verilog synthesizer ignores the delays specified in a procedural assignment statement.
  - May lead to functional mismatch between the design model and the synthesized netlist.
- A variable cannot appear as the target of both a blocking and a non-blocking assignment.
  - Following is not permissible:

```
value = value + 1;
value <= init;
```

```
// Up-down counter (synchronous clear)

module  counter (mode, clr, ld, d_in, clk, count);
    input mode, clr, ld, clk;
    input [0:7] d_in;
    output [0:7]  count;
    reg [0:7]  count;
    always  @(posedge  clk)
        if (ld)
            count <= d_in;
        else if (clr)
                    count  <= 0;
             else if (mode)
                          count <= count + 1;
                   else
                          count <= count - 1;
endmodule
```

```
// Parameterized design:: an N-bit counter

module  counter (clear, clock, count);
    parameter N = 7;
    input  clear, clock;
    output [0:N] count;
    reg    [0:N] count;

    always  @(negedge  clock)
        if (clear)
            count <= 0;
        else
            count <= count + 1;
endmodule
```


75

Example: Ring Counter

```
module  ring_counter (clk, init, count);
    input  clk, init;
    output [7:0] count;
    reg [7:0] count;
    always  @(posedge clk)
    begin
        if (init)
            count = 8'b10000000;
        else  begin
                  count    = count << 1;
                  count[0] = count[7];
              end
    end
endmodule
```

Example: Ring Counter (Modified version 1)

```
module  ring_counter (clk, init, count);
    input  clk, init;
    output [7:0] count;
    reg [7:0] count;
    always  @(posedge clk)
    begin
        if (init)
            count = 8'b10000000;
        else  begin
                count    <= count << 1;
                count[0] <= count[7];
              end
    end
endmodule
```

Example: Ring Counter (Modified version 2)

```
module  ring_counter (clk, init, count);
    input  clk, init;
    output [7:0] count;
    reg [7:0] count;
    always  @(posedge clk)
    begin
        if (init)
            count = 8'b10000000;
        else  begin
                  count = {count[6:0], count[7]};
              end
    end
endmodule
```

About “Loop” Statements

- Verilog supports four types of loops:
  - ‘while’ loop
  - ‘for’ loop
  - ‘forever’ loop
  - ‘repeat’ loop
- Many Verilog synthesizers supports only ‘for’ loop for synthesis:
  - Loop bound must evaluate to a constant.
  - Implemented by unrolling the ‘for’ loop, and replicating the statements.

Modeling Memory

- Synthesis tools are usually not very efficient in synthesizing memory.
  - Best modeled as a component.
  - Instantiated in a design.
- Implementing memory as a two-dimensional register file is inefficient.

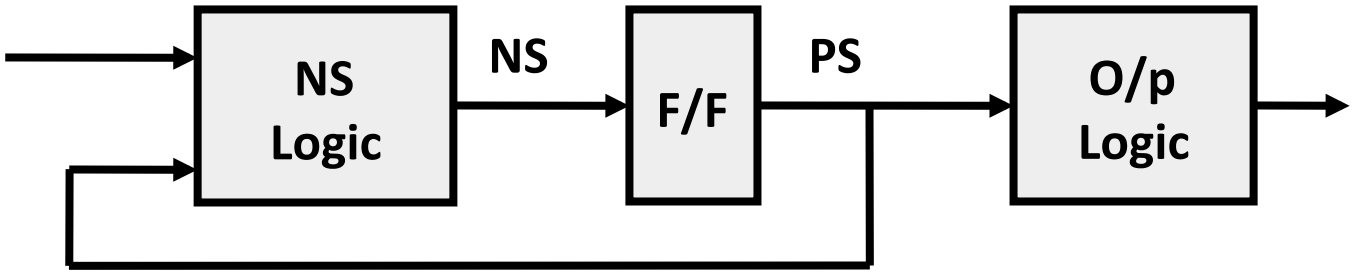
```
module  memory_example (en, clk, adbus, dbus, rw);
    parameter  N = 16;
    input  en, rw, clk;
    input  [N-1:0] adbus;
    output [N-1:0] dbus;
    ……
    ROM  Mem1  (clk, en, rw, adbus, dbus);
    ……
endmodule
```


81

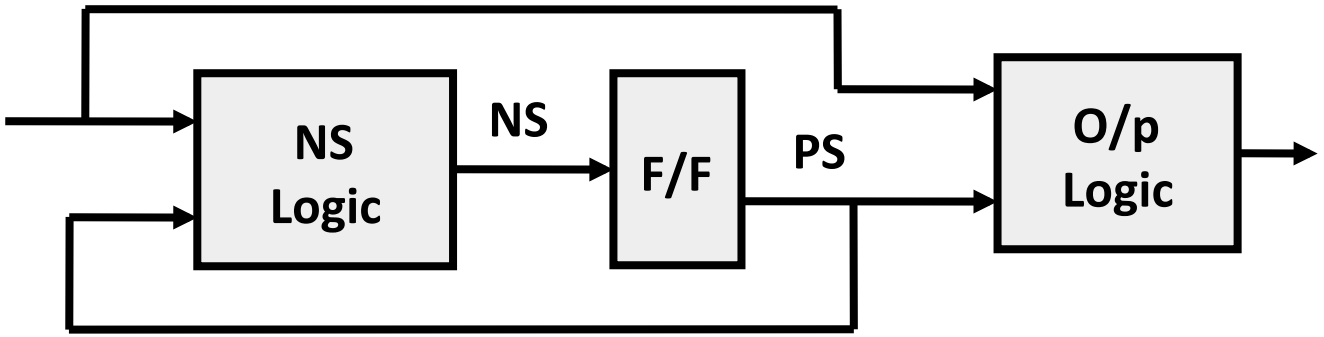
Modeling Finite State Machines

Introduction

- Two types of FSMs:
  a) Moore Machine

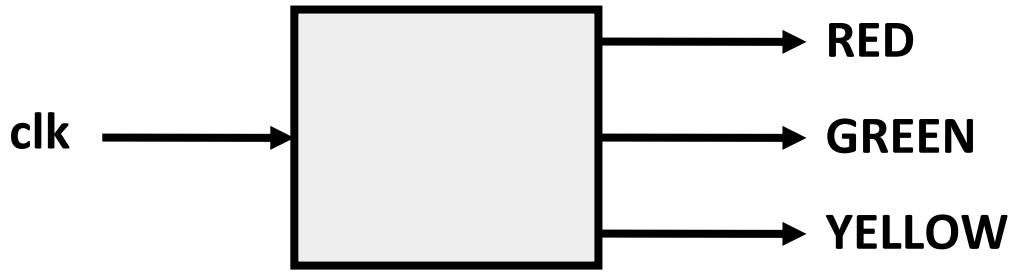


  b) Mealy Machine



Moore Machine: Example 1

- Traffic Light Controller
  - Simplifying assumptions made
  - Three lights only (RED, GREEN, YELLOW)
  - The lights glow cyclically at a fixed rate
    - Say, 10 seconds each
    - The circuit will be driven by a clock of appropriate frequency



```
le  traffic_light (clk, light);
input  clk;
output [0:2] light;       reg  [0:2] light;
parameter  S0=0, S1=1, S2=2;
parameter  RED=3'b100, GREEN=3'b010, YELLOW=3'b001;
reg [0:1]  state;
always  @(posedge  clk)
    case (state)
        S0:  begin          // S0 means RED
                light  <=  YELLOW;
                state  <=  S1;
             end
        S1:  begin          // S1 means YELLOW
                light  <=  GREEN;
                state  <=  S2;
             end
        S2:  begin          // S2 means GREEN
              light  <=  RED;
              state  <=  S0;
           end
                default: begin
                           light  <=  R
                           state  <=  S
                         end
          endcase
endmodule
```

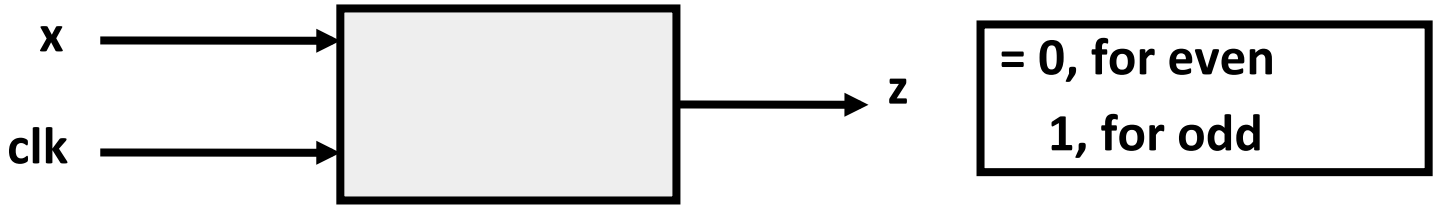
- Comment on the solution
  - Five flip-flops are synthesized
    - Two for ‘state’
    - Three for ‘light’ (outputs are also latched into flip-flops)
  - If we want non-latched outputs, we have to modify the Verilog code.
    - Assignment to ‘light’ made in a separate ‘always’ block.
    - Use blocking assignment.

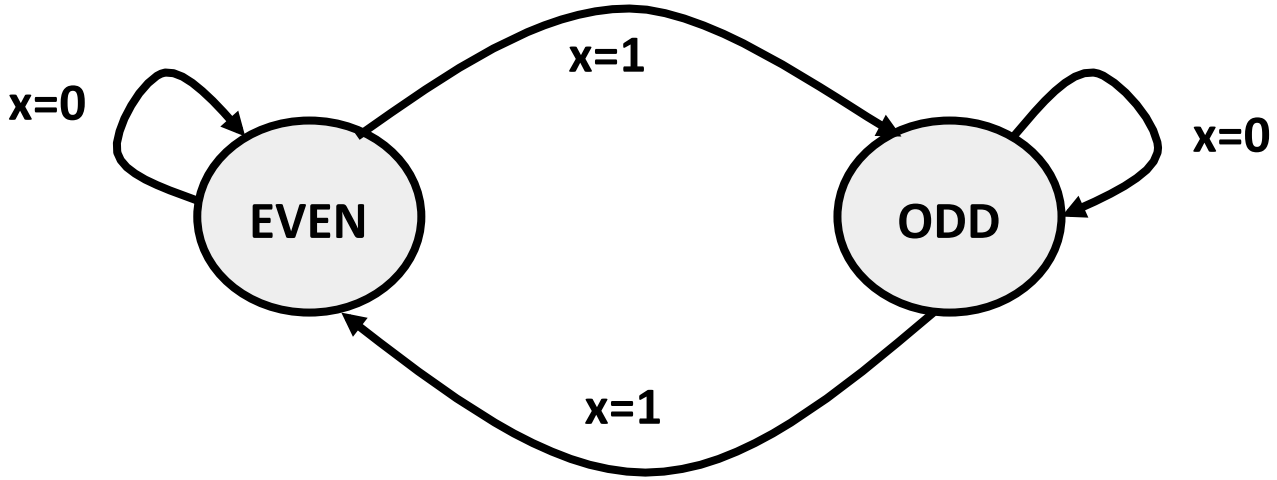
```
module  traffic_light_nonlatched_op  (clk, light);
    input  clk;
    output [0:2] light;       reg  [0:2] light;
    parameter  S0=0, S1=1, S2=2;
    parameter  RED=3'b100, GREEN=3'b010, YELLOW=3'b001;
    reg [0:1]  state;
    always  @(posedge clk)
        case (state)
            S0:       state  <=  S1;
            S1:       state  <=  S2;
            S2:       state  <=  S0;
            default:  state  <=  S0;
        endcase
    always  @(state)
        case (state)
            S0:       light  =  RED;
            S1:       light  =  YELLOW;
            S2:       light  =  GREEN;
            default:  light  =  RED;
        endcase
endmodule
```


87

Moore Machine: Example 2

- Serial parity detector





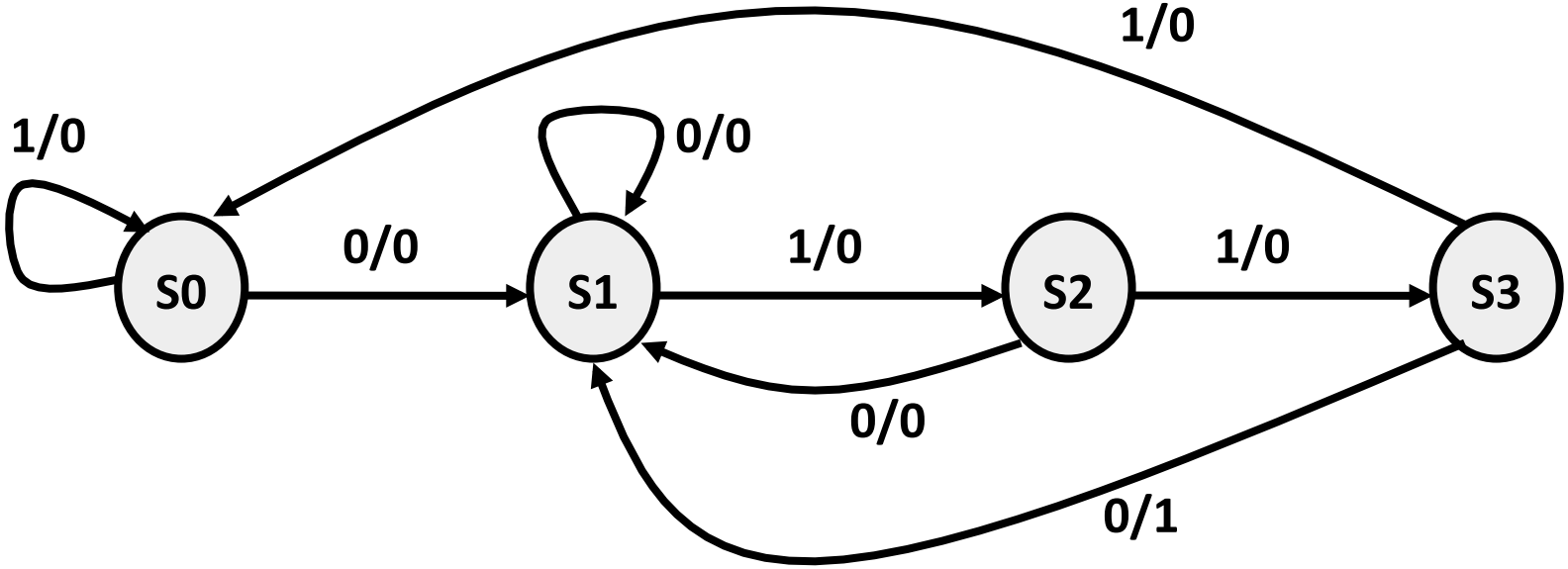
```
module  parity_gen (x, clk, z);
    input  x, clk;
    output  z;       reg  z;
    reg  even_odd;              // The machine state
    parameter  EVEN=0, ODD=1;

    always  @(posedge  clk)
        case (even_odd)
            EVEN:  begin
                      z  <=  x ? 1 : 0;
                      even_odd  <=  x ? ODD : EVEN;
                   end
            ODD:   begin
                      z  <=  x ? 0 : 1;
                      even_odd  <=  x ? EVEN : ODD;
                   end
        endcase
endmodule
```

- If no output latches need to be synthesized, we can follow the principle shown in the last exan

Mealy Machine: Example

- Sequence detector for the pattern ‘0110’.

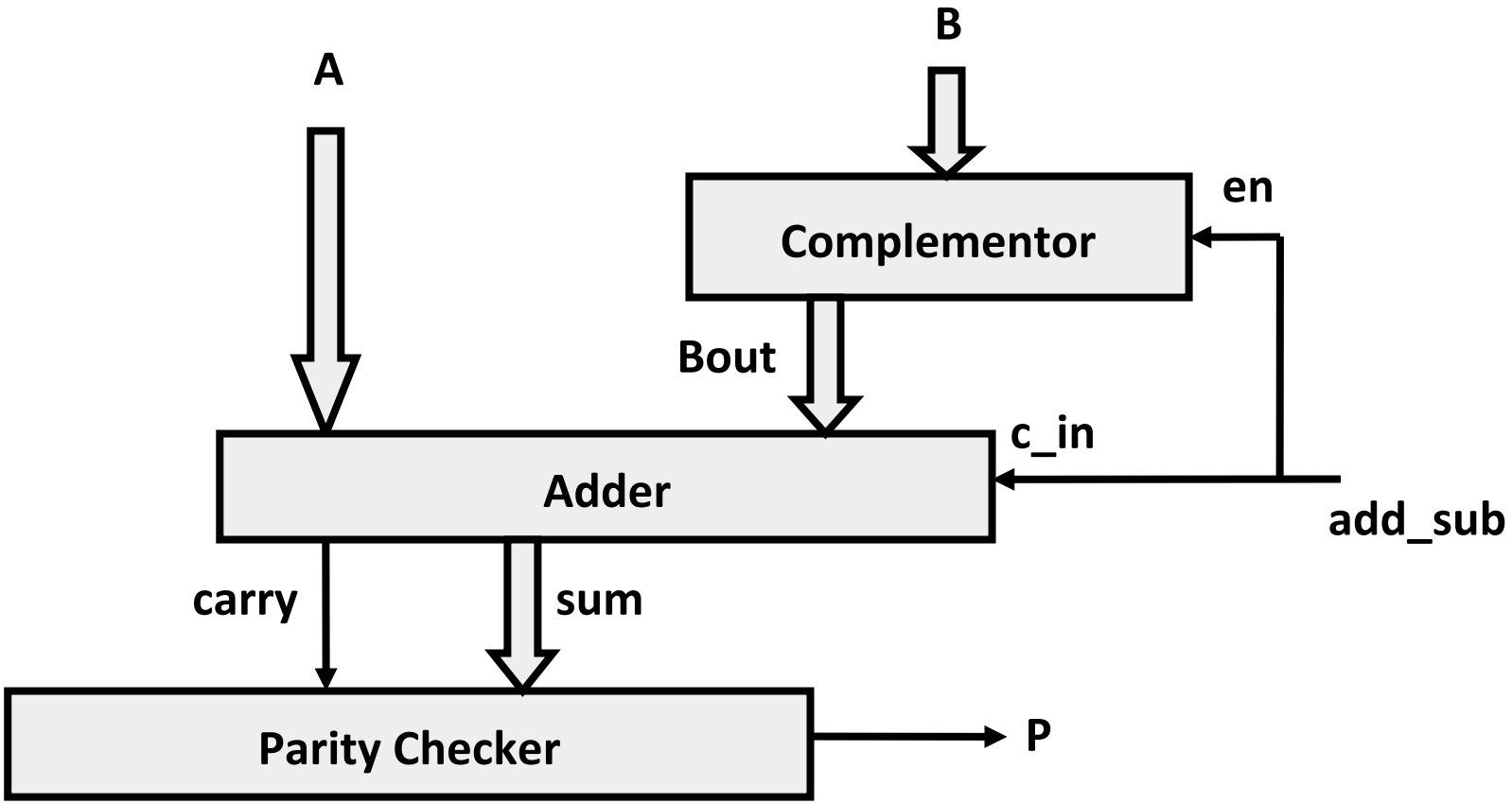


```
dule  seq_detector (x, clk, z)
 input  x, clk;
 output  z;          reg  z;
 parameter  S0=0, S1=1, S2=2, S3=3;
 reg [0:1]  PS, NS;

 always  @(posedge  clk)
     PS  <=  NS;
 always  @ (PS  or  x)
     case (PS)
         S0:  begin
                  z   =  x ? 0 : 0;
                  NS  =  x ? S0 : S1;
              end;
         S1:  begin
                  z   =  x ? 0 : 0;
                  NS  =  x ? S2 : S1;
              end;
         S2:  begin
                  z   =  x ? 0 : 0;
                  NS  =  x ? S3 : S
              end;
         S3:  begin
                  z   =  x ? 0 : 1;
                  NS  =  x ? S0 : S
              end;
      endcase
    endmodule
```


91

Example with Multiple Modules



```
ule  complementor (Y, X, comp);
 input [7:0] X;
 input  comp;
 output [7:0] Y;
 reg [7:0]  Y;

 always  @(X or comp)
     if  (comp)
         Y = ~X;
     else
         Y = X;
module
```

```
module  adder (sum, cy_out, in1, in2, cy_in
    input [7:0] in1, in2;
    input  cy_in;
    output [7:0] sum;       reg [7:0] sum;
    output  cy_out;         reg  cy_out;

    always  @(in1 or in2 or cy_in)
       {cy_out, sum} = in1 + in2 + cy_in;
endmodule
```

```
module parity_checker (out_par, in_word);
    input  [8:0] in_word;
    output  out_par;

    always  @(in_word)
        out_par = ^(in_word);
endmodule
```

```
// Top level module
module  add_sub_parity (p, a, b, add_sub);
    input [7:0] a, b;
    input  add_sub;              // 0 for add, 1 for subtract
    output  p;                   // parity of the result
    wire [7:0] Bout, sum;
    wire  carry;

    complementor    M1 (Bout, B, add_sub);
    adder           M2 (sum, carry, A, Bout, add_sub);
    parity_checker  M3 (p, {carry, sum});
endmodule
```