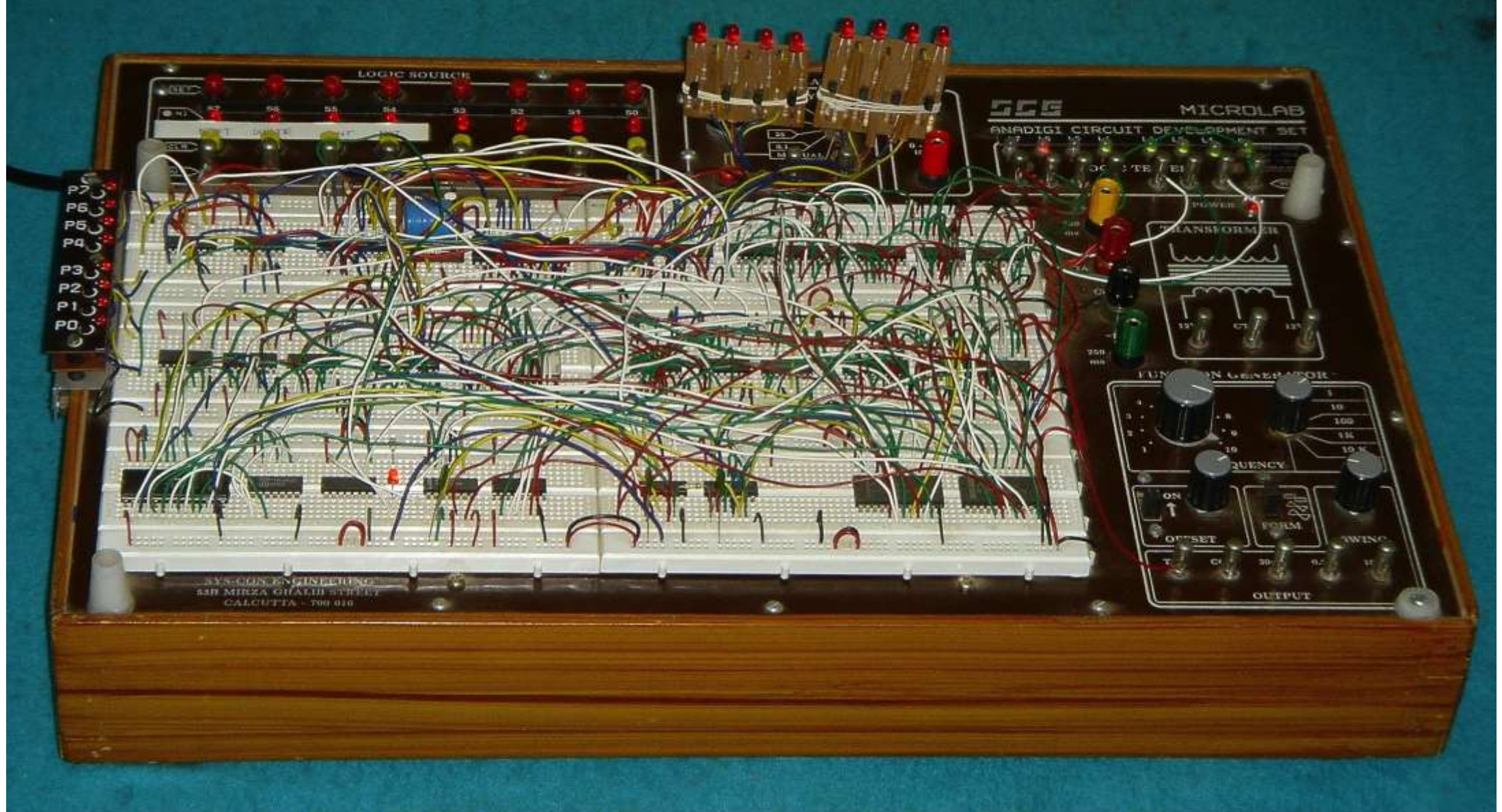


# **VERILOG**

## **Hardware Description Language**

**Before we start ...**

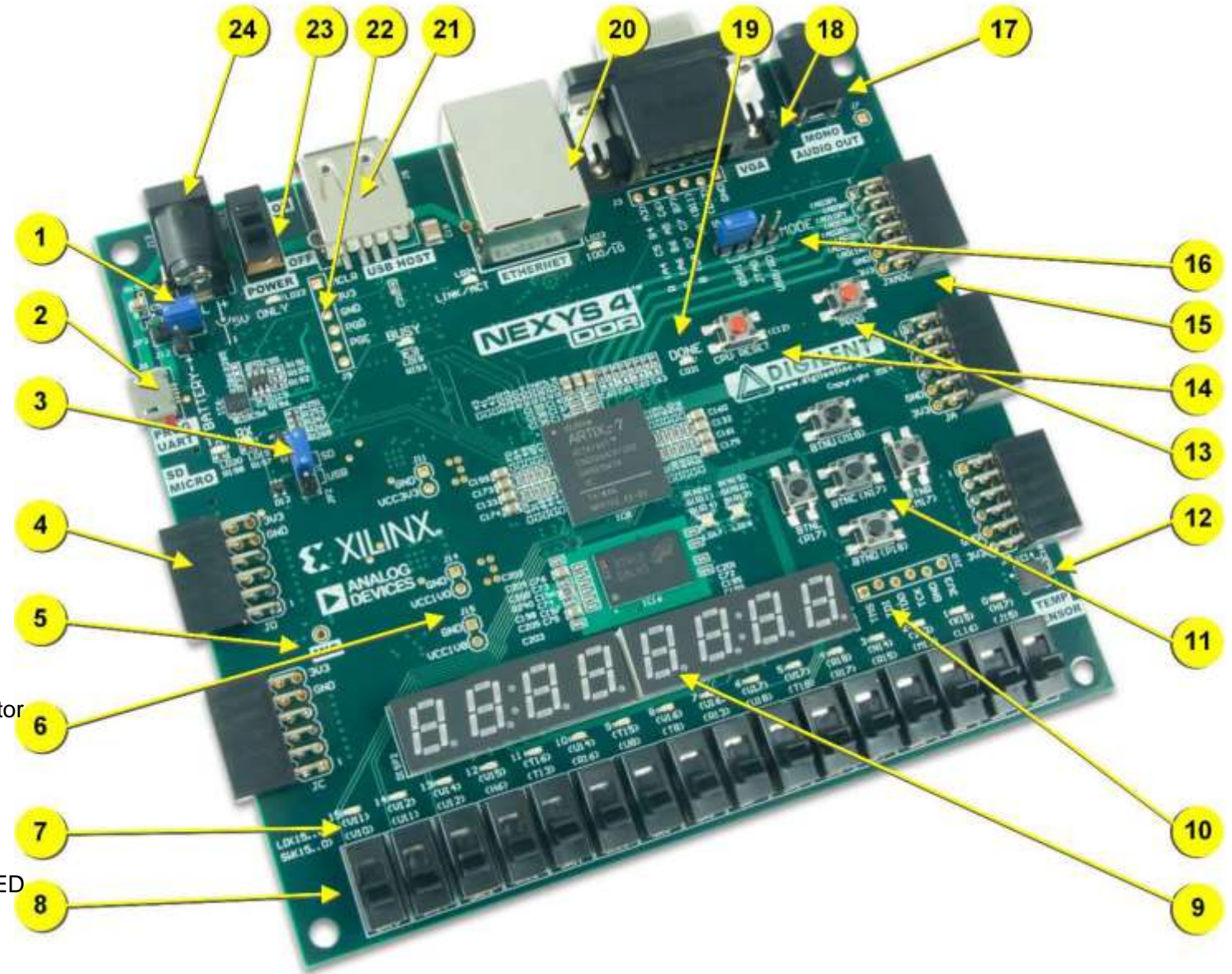


**4-Bit CPU implemented on breadboard**  
192 instructions : indexed addressing : battery back-up  
Hardware Lab. CSE-IIT-KGP : 2003-04



# FPGA Board

A programmable device where a design specified in Verilog can be downloaded.



- |  |   |
|--|---|
| 1 Power select jumper and battery header   | 13 FPGA configuration reset button      |
| 2 Shared UART/ JTAG USB port               | 14 CPU reset button (for soft cores)    |
| 3 External configuration jumper (SD / USB) | 15 Analog signal Pmod connector (XADC)  |
| 4 Pmod connector(s)                        | 16 Programming mode jumper              |
| 5 Microphone                               | 17 Audio connector                      |
| 6 Power supply test point(s)               | 18 VGA connector                        |
| 7 LEDs (16)                                | 19 FPGA programming done LED            |
| 8 Slide switches                           | 20 Ethernet connector                   |
| 9 Eight digit 7-seg display                | 21 USB host connector                   |
| 10 JTAG port for (optional) external cable | 22 PIC24 programming port (factory use) |
| 11 Five pushbuttons                        | 23 Power switch                         |
| 12 Temperature sensor                      | 24 Power jack                           |

# About Verilog

- Along with VHDL, Verilog is among the most widely used HDLs.
- Main differences:
  - VHDL was designed to support system-level design and specification.
  - Verilog was designed primarily for digital hardware designers developing FPGAs and ASICs.
- The differences become clear if someone analyzes the language features.

# Concept of Verilog “Module”

- In Verilog, the basic unit of hardware is called a **module**.
  - Modules cannot contain definitions of other modules.
  - A module can, however, be instantiated within another module.
  - Allows the creation of a hierarchy in a Verilog description.

# Basic Syntax of Module Definition

```
module  module_name  (list_of_ports) ;  
    input/output declarations  
    local net declarations  
    Parallel statements  
endmodule
```

## Example 1: A simple AND function

```
module simple_and (f, x, y);  
    input  x, y;  
    output f;  
    assign f = x & y;  
endmodule
```



## Example 2 :: A two-level circuit

```
module two_level (a, b, c, d, f);  
    input  a, b, c, d;  
    output f;  
    wire  t1, t2;  
    assign t1 = a & b;  
    assign t2 = ~(c | d);  
    assign f = t1 ^ t2;  
endmodule
```

## Example 3 :: A hierarchical design

```
module add3 (s, cy3, cy_in, x, y);  
    input [2:0] x, y;  
    input cy_in;  
    output [2:0] s;  
    output cy3;  
    wire [1:0] cy_out;  
    add B0 (cy_out[0], s[0], x[0], y[0], cy_in);  
    add B1 (cy_out[1], s[1], x[1], y[1], cy_out[0]);  
    add B2 (cy3, s[2], x[2], y[2], cy_out[1]);  
endmodule
```

```
module add (co, sum, a, b, cin);  
    input a, b, cin;  
    output co;  
    assign co = (a & b) | (b & cin)  
                | (cin & a);  
    assign sum = a ^ b ^ cin;  
endmodule
```

# Specifying Connectivity

- There are two alternate ways of specifying connectivity:

- Positional association

- The connections are listed in the same order

```
add A1 (c_out, sum, a, b, c_in);
```

- Explicit association

- May be listed in any order

```
add A1 (.in1(a), .in2(b), .cin(c_in), .sum(sum), .cout(c_out));
```

# Variable Data Types

- A variable belongs to one of two data types:
  - 1) **Net**
    - Must be continuously driven
    - Used to model connections between continuous assignments & instantiations
  - 2) **Register**
    - Retains the last value assigned to it
    - Often used to represent storage elements

# (1) “Net” Data Type

- Different ‘net’ types supported for synthesis:
  - wire, wor, wand, tri, supply0, supply1
- ‘wire’ and ‘tri’ are equivalent; when there are multiple drivers driving them, the outputs of the drivers are shorted together.
- ‘wor’ / ‘wand’ inserts an OR / AND gate at the connection.
- ‘supply0’ / ‘supply1’ model power supply connections.



```
module using_wire (A, B, C, D, f);  
    input    A, B, C, D;  
    output   f;  
    wire     f;        // net f declared as 'wire'  
  
    assign   f = A & B;  
    assign   f = C | D;  
endmodule
```

```
module using_supply_wire (A, B, C, f);  
    input    A, B, C;  
    output   f;  
    supply0  gnd;  
    supply1  vdd;  
    nand     G1 (t1, vdd, A, B);  
    xor      G2 (t2, C, gnd);  
    and      G3 (f, t1, t2);  
endmodule
```

## (2) “Register” Data Type

- Different ‘register’ types supported for synthesis:
  - reg, integer
- The ‘reg’ declaration explicitly specifies the size.

`reg x, y;`                      // single-bit register variables

`reg [15:0] bus;`              // 16-bit bus, bus[15] is MSB

- For ‘integer’, it takes the default size, usually 32-bits.
  - The synthesis tool tries to determine the size.

# Other differences

- In arithmetic expressions,
  - An 'integer' is treated as a 2's complement signed integer.
  - A 'reg' is treated as an unsigned quantity.
- General rule of thumb
  - 'reg' used to model actual hardware registers such as counters, accumulator, etc.
  - 'integer' used for situations like loop counting.

# Specifying Constant Values

- A value may be specified in either the 'sized' or the 'un-sized' form.
  - Syntax for 'sized' form:

`<size>'<base><number>`

- Examples:

`8'b01110011 // 8-bit binary number`

`12'hA2D // 1010 0010 1101 in binary`

`12'hCx5 // 1100 xxxx 0101 in binary`

`25 // signed number, 32 bits`

`1'b0 // logic 0`

`1'b1 // logic 1`



# Parameters

- A parameter is a constant with a name.
- No size is allowed to be specified for a parameter.
  - The size gets decided from the constant itself (32-bits if nothing is specified).
- Examples:

```
parameter  HI = 25, LO = 5;
```

```
parameter  up = 2b'00, down = 2b'01, steady = 2b'10;
```

# Logic Values

- The common values used in modeling hardware are:
  - 0 :: Logic-0 or FALSE
  - 1 :: Logic-1 or TRUE
  - x :: Unknown (or don't care)
  - z :: High impedance
- Initialization:
  - All unconnected nets set to 'z'
  - All register variables set to 'x'

- Verilog provides a set of predefined logic gates.
  - AND, OR, XOR, NAND, NOR, NOT.
  - They respond to inputs (0, 1, x, or z) in a logical way.
  - Example :: AND

$$0 \& 0 \rightarrow 0$$

$$0 \& x \rightarrow 0$$

$$0 \& 1 \rightarrow 0$$

$$1 \& z \rightarrow x$$

$$1 \& 1 \rightarrow 1$$

$$z \& x \rightarrow x$$

$$1 \& x \rightarrow x$$

# Primitive Gates

- Primitive logic gates (instantiations):

```
and    G (out, in1, in2);
```

```
nand   G (out, in1, in2);
```

```
or     G (out, in1, in2);
```

```
nor    G (out, in1, in2);
```

```
xor    G (out, in1, in2);
```

```
xnor   G (out, in1, in2);
```

```
not    G (out1, in);
```

```
buf    G (out1, in);
```

- Primitive Tri-State gates (instantiation)

```
bufif1  G (out, in, ctrl);
```

```
bufif0  G (out, in, ctrl);
```

```
notif1  G (out, in, ctrl);
```

```
notif0  G (out, in, ctrl);
```



# Points to Note

- For all primitive gates,
  - The output port must be connected to a net (a wire).
  - The input ports may be connected to nets or register type variables.
  - They can have a single output but any number of inputs.
  - An optional delay may be specified.
    - Logic synthesis tools ignore time delays.

```
`timescale 10ns / 1ns
module exclusive_or (f, a, b);
    input a, b;
    output f;
    wire t1, t2, t3;
    nand #5 m1 (t1, a, b);
    and #5 m2 (t2, a, t1);
    and #5 m3 (t3, t1, b);
    or #5 m4 (f, t2, t3);
endmodule
```

# **Hardware Modeling Issues**

# Some Facts

- The values computed can be held in
  - A 'wire'
  - A 'flip-flop' (edge-triggered storage cell)
  - A 'latch' (level-sensitive storage cell)
- A variable in Verilog can be of
  - 'net data type'
    - Maps to a 'wire' during synthesis
  - 'register' data type
    - Maps either to a 'wire' or to a 'storage cell' depending on the context under which a value is assigned.

```
module reg_maps_to_wire (A, B, C, f1, f2);  
    input  A, B, C;  
    output f1, f2;  
    wire   A, B, C;  
    reg    f1, f2;  
    always @(A or B or C)  
    begin  
        f1 = ~(A & B);  
        f2 = f1 ^ C;  
    end  
endmodule
```

**The synthesis system  
will generate a wire  
for f1**



```
module a_problem_case (A, B, C, f1, f2);  
    input  A, B, C;  
    output f1, f2;  
    wire   A, B, C;  
    reg     f1, f2;  
    always @(A or B or C)  
    begin  
        f2 = f1 ^ f2;  
        f1 = ~(A & B);  
    end  
endmodule
```

**The synthesis system  
will not generate a  
storage cell for f1**

```
// A latch gets inferred here
module simple_latch (data, load, d_out);
    input    data, load;
    output   d_out;
    wire t;
    always @(load or data)
        begin
            if (!load)
                t = data;
            d_out = t;
        end
endmodule
```

**Else part missing; so  
latch is inferred.**

# Verilog Operators

- Arithmetic operators

`*`, `+`, `-`, `%`

- Logical operators

`!` → logical negation

`&&` → logical AND

`||` → logical OR

- Relational operators

`>`, `<`, `>=`, `<=`, `==`, `!=`

- Bitwise operators

`~`, `&`, `|`, `^`, `~^`

- Reduction operators (operate on all the bits within a word)

$\&$ ,  $\sim\&$ ,  $|$ ,  $\sim|$ ,  $\wedge$ ,  $\sim\wedge$

→ accepts a single word operand and produces a single bit as output

- Shift operators

$\gg$ ,  $\ll$

- Concatenation  $\{ \}$

- Replication  $\{ \{ \} \}$

- Conditional

$\langle \text{condition} \rangle ? \langle \text{expression1} \rangle : \langle \text{expression2} \rangle$

```
module operator_example (x, y, f1, f2);  
    input  x, y;  
    output f1, f2;  
    wire [9:0] x, y;  
    wire [4:0] f1;  
    wire f2;  
  
    assign f1 = x[4:0] & y[4:0];  
    assign f2 = x[2] | ~f1[3];  
    assign f2 = ~& x;  
    assign f1 = f2 ? x[9:5] : x[4:0];  
endmodule
```

```
// An 8-bit adder description

module parallel_adder (sum, cout, in1, in2, cin);
    input [7:0] in1, in2;
    input cin;
    output [7:0] sum;
    output cout;

    assign #20 {cout, sum} = in1 + in2 + cin;
endmodule
```

# **Description Styles in Verilog**

# Introduction

- Two different styles of description:
  - 1) Data flow
    - Continuous assignment
  - 2) Behavioral
    - Procedural assignment
      - a) Blocking
      - b) Non-blocking



# (1) Data-flow Style: Continuous Assignment

- Identified by the keyword “**assign**”.

```
assign    a = b & c;
```

```
assign    f[2] = c[0];
```

- Forms a static binding between
  - The ‘net’ being assigned on the LHS,
  - The expression on the RHS.
- The assignment is continuously active.
- Almost exclusively used to model combinational logic.

- A Verilog module can contain any number of continuous assignment statements.
- For an “assign” statement,
  - The expression on RHS may contain both “register” or “net” type variables.
  - The LHS must be of “net” type, typically a “wire”.
- Several examples of “assign” illustrated already.

```
module    generate_mux (data, select, out);  
    input  [0:7] data;  
    input  [0:2] select;  
    output out;  
  
    assign out = data[select];  
endmodule
```

**Non-constant index in  
expression on RHS  
generates a MUX**

```
module generate_decoder (out, in, select);  
    input  in;  
    input [0:1] select;  
    output [0:3] out;  
  
    assign out[select] = in;  
endmodule
```

**Non-constant index in  
expression on LHS  
generates a decoder**

```
module generate_set_of_MUX (a, b, f, sel);  
    input  [0:3] a, b;  
    input  sel;  
    output [0:3] f;  
  
    assign f = sel ? a : b;  
endmodule
```

**Conditional operator  
generates a MUX**

```
module level_sensitive_latch (D, Q, En);  
    input  D, En;  
    output Q;  
  
    assign Q = en ? D : Q;  
endmodule
```


**Using “assign” to describe sequential logic**

## (2) Behavioral Style: Procedural Assignment

- The procedural block defines
  - A region of code containing sequential statements.
  - The statements execute in the order they are written.
- Two types of procedural blocks in Verilog
  - a)The “**always**” block
    - A continuous loop that never terminates.
  - b)The “**initial**” block
    - Executed once at the beginning of simulation (used in Test-benches).

- A module can contain any number of “always” blocks, all of which execute concurrently.
- Basic syntax of “always” block:

```
always @(event_expression)
    begin
        statement;
        :
        statement;
    end
```



The diagram illustrates the basic syntax of an "always" block. It shows the keywords `always`, `@(event_expression)`, `begin`, `statement;`, a colon `:`, another `statement;`, and `end`. A right-facing curly brace groups the two `statement;` lines, and an arrow points from the brace to a rectangular box containing the text "Sequential statements".

- The `@(event_expression)` is required for both combinational and sequential logic descriptions.



- Only “reg” type variables can be assigned within an “always” block.

Why??

- The sequential “always” block executes only when the event expression triggers.
- At other times the block is doing nothing.
- An object being assigned to must therefore remember the last value assigned (not continuously driven).
- So, only “reg” type variables can be assigned within the “always” block.
- Any kind of variable may appear in the event expression (reg, wire, etc.).

# Sequential Statements in Verilog

1. `begin`  
    `sequential_statements`  
`end`
2. `if (expression)`  
    `sequential_statement`  
    `[else`  
        `sequential_statement]`
3. `case (expression)`  
    `expr:        sequential_statement`  
    `.....`  
    `default:  sequential_statement`  
`endcase`

**begin...end  
not required if there  
is only 1 stmt.**

4. `forever`  
    `sequential_statement`
5. `repeat (expression)`  
    `sequential_statement`
6. `while (expression)`  
    `sequential_statement`
7. `for (expr1; expr2; expr3)`  
    `sequential_statement`
8. `#(time_value)`  
    → Makes a block suspend for "time\_value" time units.
9. `@(event_expression)`  
    → Makes a block suspend until event\_expression triggers.

```
// A combinational logic example

module mux21 (in1, in0, s, f);
    input  in1, in0, s;
    output f;
    reg f;

    always @(in1 or in0 or s)
        if (s)
            f = in1;
        else
            f = in0;
endmodule
```

```
// A sequential logic example

module dff_negedge (D, clock, Q, Qbar);
    input  D, clock;
    output Q, Qbar;
    reg    Q, Qbar;

    always @(negedge clock)
        begin
            Q = D;
            Qbar = ~D;
        end
endmodule
```

```
// Another sequential logic example

module  incomp_state_spec (curr_state, flag);
    input  [0:1] curr_state;
    output [0:1] flag;
    reg     [0:1] flag;

    always @(curr_state)
        case (curr_state)
            0,1 : flag = 2;
            3   : flag = 0;

        endcase
endmodule
```

The variable 'flag' is not assigned a value in all the branches of case.  
→ Latch is *inferred*

```
// A small change made

module  incomp_state_spec (curr_state, flag);
    input  [0:1] curr_state;
    output [0:1] flag;
    reg     [0:1] flag;

    always @(curr_state)
        flag = 0;
        case (curr_state)
            0,1 : flag = 2;
            3   : flag = 0;
        endcase
endmodule
```

**'flag' defined for all  
values of curr\_state.  
→ Latch is *avoided***

```

module  ALU_4bit (f, a, b, op);

    input [1:0] op;      input [3:0] a, b;
    output [3:0] f;      reg    [3:0] f;

    parameter  ADD=2'b00, SUB=2'b01, MUL=2'b10, DIV=2'b11;

    always  @(a or b or op)
        case (op)
            ADD : f = a + b;
            SUB : f = a - b;
            MUL : f = a * b;
            DIV : f = a / b;
        endcase
endmodule

```



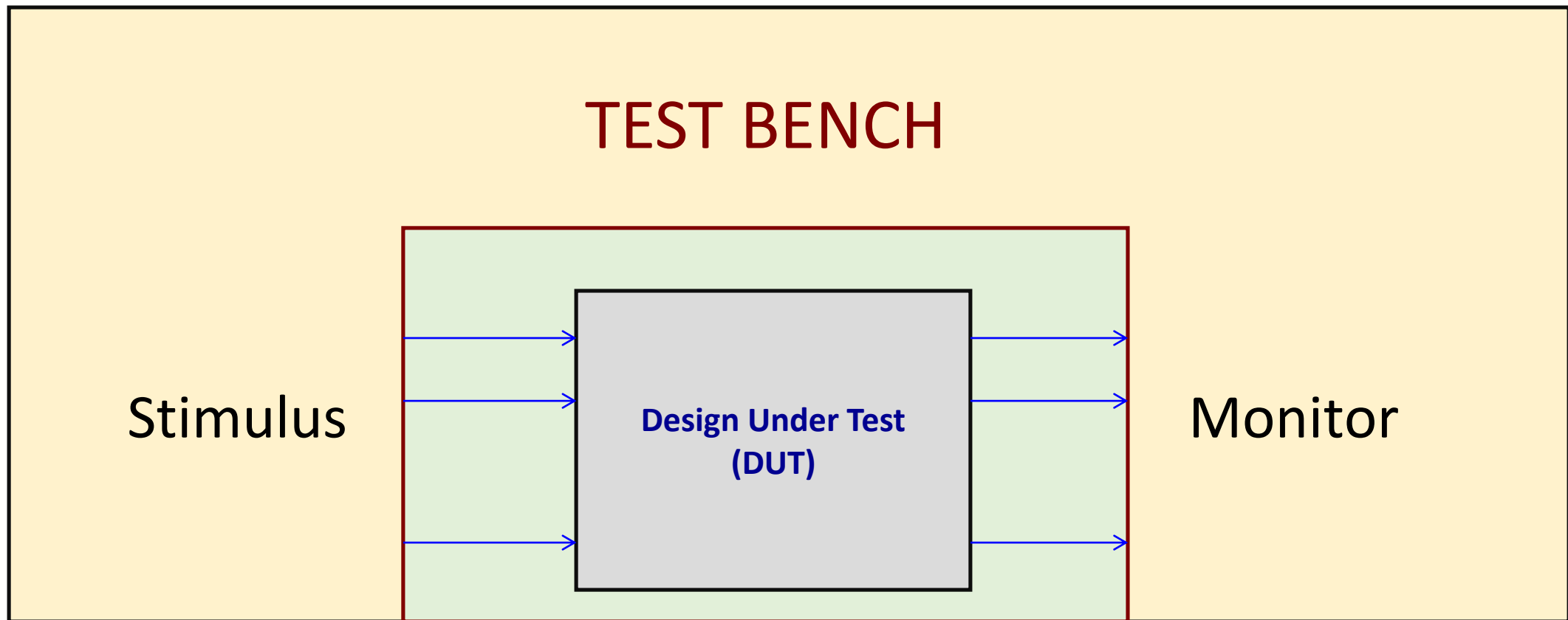
```
module priority_encoder (in, code);  
    input  [0:3] in;  
    output [0:1] code;  
    reg [0:1] code;  
    always @(in)  
        case (1'b1)  
            input[0] : code = 2'b00;  
            input[1] : code = 2'b01;  
            input[2] : code = 2'd10;  
            input[3] : code = 2'b11;  
        endcase  
endmodule
```

# **Verilog Test Bench**

# Verilog Test Bench

- What is test bench?
  - A Verilog procedural block that executes only once.
  - Used for simulation.
  - Test bench generates clock, reset, and the required test vectors for a given *design-under-test* (DUT).
  - The test bench can monitor the DUT outputs and present them in a way as specified by the creator.
    - Print the values of the signal lines.
    - Dump the values in a file from where waveforms can be viewed.

- Basic requirements:
  - The inputs of the DUT need to be connected to the test bench.
  - The outputs of the DUT needs also to be connected to the test bench.
- Points to note:
  - Test benches use the “*initial*” procedural block that executes only once.
  - Can also use “*always*” for generating some test inputs, like a clock signal.



# A Simple Example

```
module example (A,B,C,D,E,F,Y) ;  
  input A,B,C,D,E,F;  
  output Y;  
  wire  t1, t2, t3, Y;  
  nand  #1 G1  (t1,A,B) ;  
  and   #2 G2  (t2,C,~B,D) ;  
  nor   #1 G3  (t3,E,F) ;  
  nand  #1 G4  (Y,t1,t2,t3) ;  
endmodule
```

```
module  testbench;  
  reg  A,B,C,D,E,F;  wire Y;  
  example DUT(A,B,C,D,E,F,Y) ;  
  
  initial  
    begin  
      $monitor ($time," A=%b, B=%b, C=%b,  
        D=%b, E=%b, F=%b, Y=%b",  
        A,B,C,D,E,F,Y) ;  
      #5 A=1; B=0; C=0; D=1; E=0; F=0;  
      #5 A=0; B=0; C=1; D=1; E=0; F=0;  
      #5 A=1; C=0;  
      #5 F=1;  
      #5 $finish;  
    end  
endmodule
```

# How to write test benches?

- Create a dummy template
  - Declare inputs to the design-under-test (DUT) as “*reg*”, and the outputs as “*wire*”.
    - Because we have to initialize the DUT inputs inside procedural block(s), typically “*initial*”, where only “*reg*” type variables can be assigned.
  - Instantiate the DUT.
- Initialization and Monitoring
  - Assign some known values to the DUT inputs.
  - Monitor the DUT outputs for functional verification.

- For synchronous sequential circuits:
  - We need some clock generation logic.
  - Various ways to specify clock signal.
- Test bench can include various simulator directives:
  - *\$display*, *\$monitor*, *\$dumpfile*, *\$dumpvars*, *\$finish*, etc.
- Important point:
  - We do not need test bench when we are synthesizing a design.
  - Required only during simulation.



# The Simulator Directives

- `$display ("<format>", expr1, expr2, ...)` ;
  - Used to print the immediate values of text or variables to stdout.
  - Syntax is very similar to "printf" in C.
  - Additional format specifiers are supported, like "b" (binary), "h" (hexadecimal), etc.
- `$monitor ("<format>", var1, var2, ...)` ;
  - Similar in syntax to *\$display*, but does not print immediately.
  - It will print the value(s) whenever the value of some variable(s) in the given list changes.
  - Has the functionality of *event-driven* print.

- **\$finish;**
  - Terminates the simulation process.
- **\$dumpfile (<filename>) ;**
  - Specifies the file that will be used for storing the values of the selected variables so that they can be graphically visualized later.
  - The file typically has an extension *.vcd (Value Change Dump)*, and contains information about any value changes on the selected variables.
- **\$dumpoff;**
  - This directive stops the dumping of variables. All variables are dumped with “x” values and the next change of variables will not be dumped.
- **\$dumpon ;**
  - This directive starts previously stopped dumping of variables.

- **\$dumpvars (level, list\_of\_variables\_or\_modules) ;**
  - Specifies which variables should be dumped to the *.vcd* file.
  - Both the parameters are optional; if both are omitted, all variables are dumped.
  - If *level=0*, then all variables within the modules from the list will be dumped. If any module from the list contains module instances, then all variables from these modules will also be dumped.
  - If *level=1*, then only listed variables and variables of listed modules will be dumped.
- **\$dumpall ;**
  - The current values of all variables will be written to the file, irrespective of whether there has been any change in their values or not.
- **\$dumplimit (filesize) ;**
  - Used to set the maximum size of the *.vcd* file.

# Example 1: Full Adder

```
module full_adder (s, co, a, b, c);  
    input a, b, c;  
    output s, co;  
    assign s = a ^ b ^ c;  
    assign co = (a & b) | (b & c) | (c & a);  
endmodule
```

```
module testbench_1;
  reg a, b, c; wire sum, cout;
  full_adder FA (sum, cout, a, b, c);

  initial
    begin
      $monitor ($time," a=%b, b=%b, c=%b, sum=%b, cout=%b",
                a, b, c, sum, cout);
      #5 a=0; b=0; c=1;
      #5 b=1;
      #5 a=1;
      #5 a=0; b=0; c=0;
      #5 $finish;
    end
endmodule
```

```

module testbench_2;
  reg a, b, c; wire sum, cout;
  integer i;
  full_adder FA (sum, cout, a, b, c);

  initial
    begin
      for (i=0; i<8; i=i+1)
        begin
          {a,b,c} = i; #5;
          $display ("T=%2d, a=%b, b=%b, c=%b, sum=%b, cout=%b",
                    $time, a, b, c, sum, cout);
        end
      #5 $finish;
    end
endmodule

```

```

T= 5, a=0, b=0, c=0, sum=0, cout=0
T=10, a=0, b=0, c=1, sum=1, cout=0
T=15, a=0, b=1, c=0, sum=1, cout=0
T=20, a=0, b=1, c=1, sum=0, cout=1
T=25, a=1, b=0, c=0, sum=1, cout=0
T=30, a=1, b=0, c=1, sum=0, cout=1
T=35, a=1, b=1, c=0, sum=0, cout=1
T=40, a=1, b=1, c=1, sum=1, cout=1

```

## Example 2: Generating random test vectors

```
module adder (out, cout, a, b);  
    input [7:0] a, b;  
    output [7:0] out;  
    output cout;  
  
    assign #5 {cout,out} = a + b;  
endmodule
```

- The system task *\$random* can be used to generate a random number.
- It is called as : *\$random (<seed>)*
  - The value of *<seed>* is optional and is used to ensure that the same sequence of random numbers are generated each time the test is run.

```

module test_adder;
    reg [7:0] a, b;
    wire [7:0] sum;    wire cout;
    integer myseed;

    adder ADD (sum, cout, a, b);

    initial
        begin
            repeat (5)
                begin
                    a = $random(myseed);
                    b = $random(myseed); #10;
                    $display ("T: %3d, a: %h, b: %h, sum: %h", $time, a, b, sum);
                end
            end
        end
    endmodule

```

T:	10,	a:	00,	b:	52,	sum:	52
T:	20,	a:	ca,	b:	08,	sum:	d2
T:	30,	a:	0c,	b:	6a,	sum:	76
T:	40,	a:	b1,	b:	71,	sum:	22
T:	50,	a:	23,	b:	df,	sum:	02



# **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
```

# 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
```

**WRONG**



# 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.

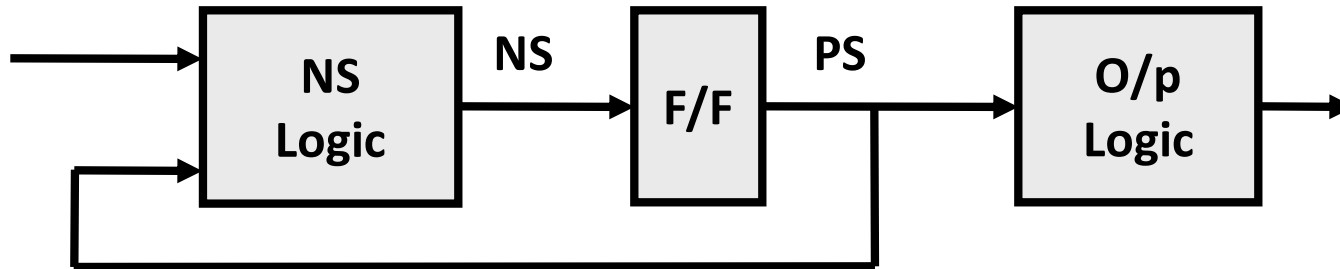
```
module memory_example (en, clk, adbus, dbus, rw);  
    parameter N = 16;  
    input en, rw, clk;  
    input [N-1:0] adbus;  
    output [N-1:0] dbus;  
  
    .....  
    ROM Mem1 (clk, en, rw, adbus, dbus);  
  
    .....  
endmodule
```

# **Modeling Finite State Machines**

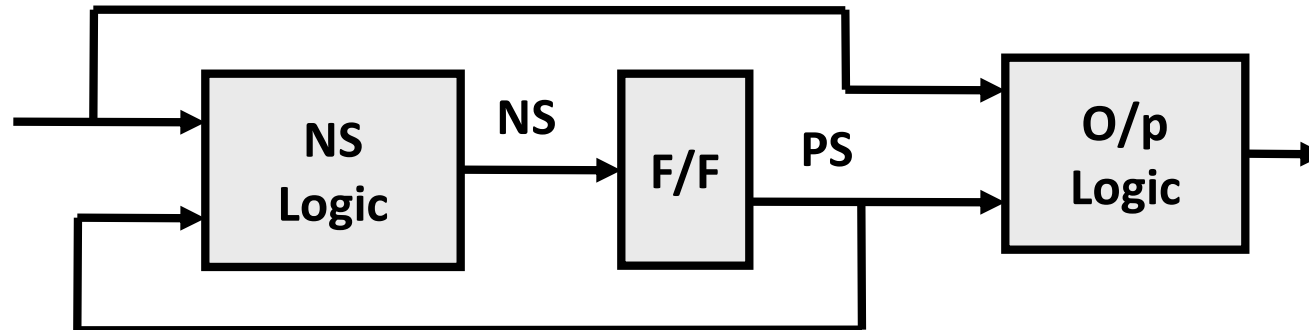
# Introduction

- Two types of FSMs:

a) Moore Machine

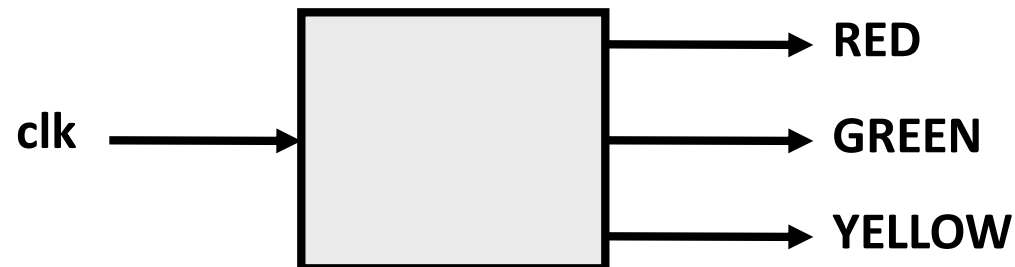


a) Mealy Machine



# 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





```

module 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
        end
end

```

```

                                default: begin
                                    light  <=  RED;
                                    state  <=  S0;
                                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.

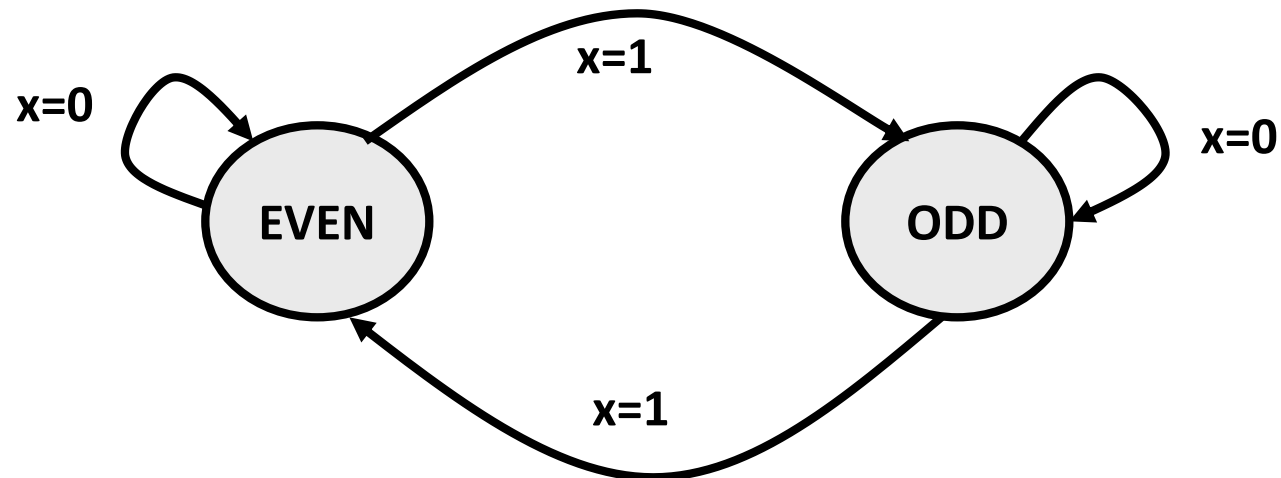
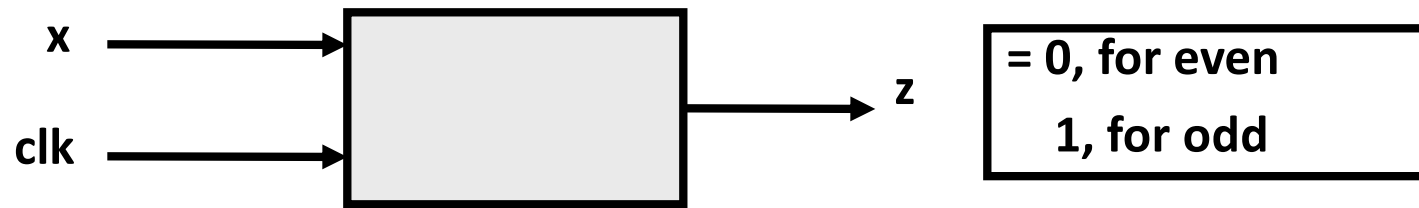
```

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

```

# 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

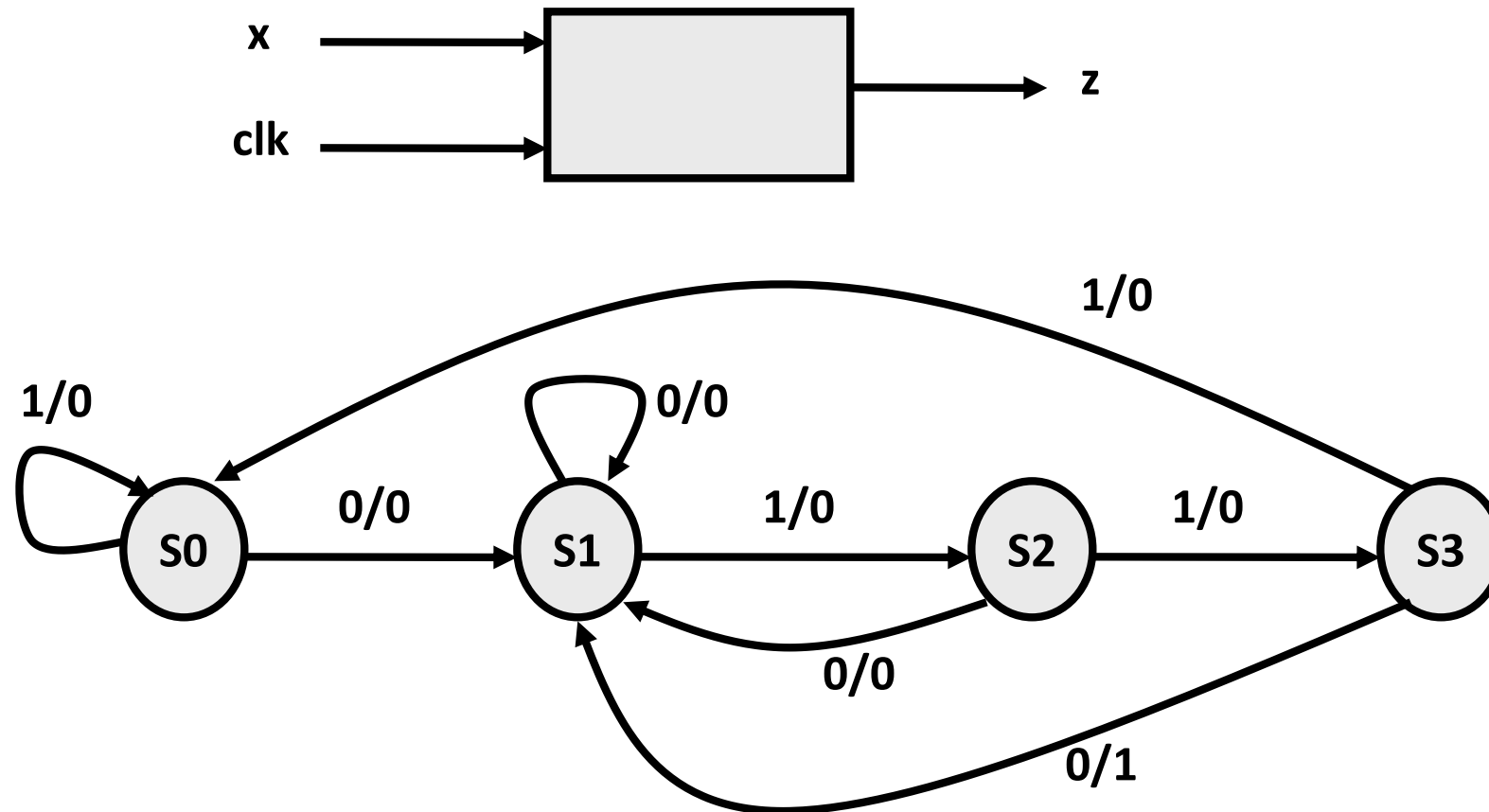
        endcase
endmodule

```

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

# Mealy Machine: Example

- Sequence detector for the pattern '0110'.



```

module 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;

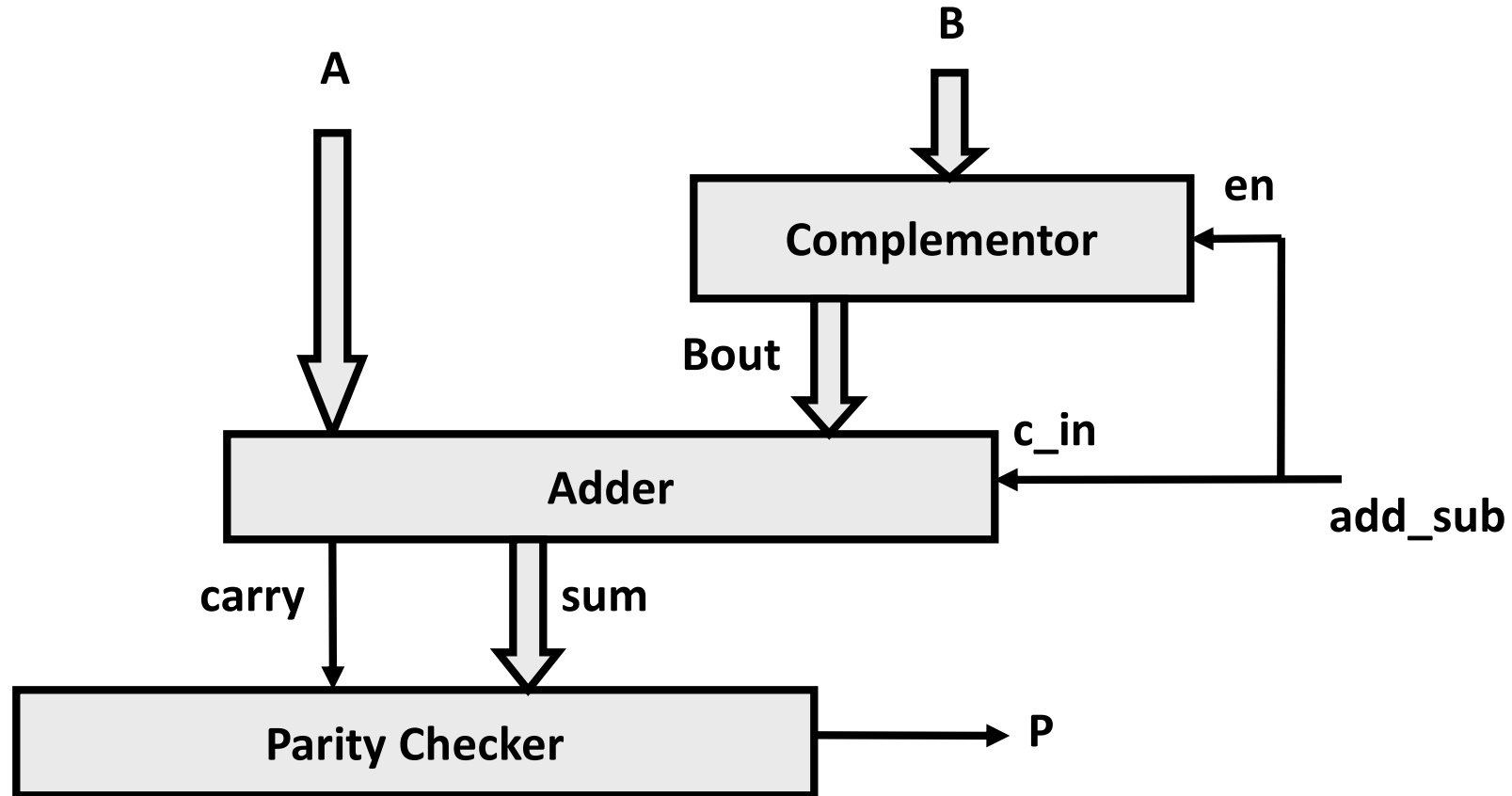
```

```

            S2: begin
                    z = x ? 0 : 0;
                    NS = x ? S3 : S1;
                end;
            S3: begin
                    z = x ? 0 : 1;
                    NS = x ? S0 : S1;
                end;
        endcase
    endmodule

```

# Example with Multiple Modules





```

module 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;
endmodule

```

```

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

```