Outline

- Introduction
- Hardware Description Languages
- Different Levels of Abstraction
- Getting Started With Verilog
- Verilog Language Features
- Test benches and Simulation

Introduction

As digital and electronic circuit designs grew in size and complexity, capturing a large design at the gate level of abstraction with schematic-based design became

- **Too complex,**
- **Prone to error,**
- **Extremely time-consuming.**

Complex digital circuit designs require a lot more time for development, synthesis, simulation and debugging.

Solution? Computer Aided Design (CAD) tools

- Based on Hardware Description Languages

Nowadays, billions of transistors per chip!

Moore's Law?

Hardware Description Language (HDL)

HDLs are specialized computer languages used to program electronic and digital logic circuits.

- High level languages with which we can specify our HW to analyze its design before actual fabrication.

Two most popular HDLs:

- **Verilog**
- **VHDL**

Other popular HDLs:

- **SystemC**
- **SystemVerilog**
- **...**

Design Flow (Simplified)

Some other steps in design:

- **Simulation** to verify the design (at different levels)
- **Formal verification**
- etc.

Different Levels of Abstraction

Behavioral vs. Structural Design

- **Behavioral**: the highest level of abstraction - specifying the functionality in terms of its ***behavior*** (e.g. Boolean equations, truth tables, algorithms, code, etc.). **WHAT, not HOW.**

- **Structural**: a netlist specification of components and their interconnections (e.g. gates, transistors, even functional modules).

We will use both.

Different Levels of Abstraction

Example: Full Adder

- **Behavioral modeling**:

A	B	Cin	Sum	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

Truth table

Cout = B Cin + A Cin + A B

S = A′ B′ Cin + A′ B Cin′ + A B′ Cin′ + A B Cin
= A′ (B′ Cin + B Cin′) + A (B′ Cin′ + B Cin)
= A′ Z + A Z′
= A xor Z = A xor (B xor Cin)

In terms of Boolean expressions

- **Structural modeling**:

Circuit diagram

Different Levels of Abstraction in Verilog HDL

Example: Full Adder

- **Gate-Level modeling**:
  - Verilog HDL supports built-in primitive gates modeling.

```
module full_adder(x,y,cin,sum,cout);
      input x,y,cin;
      output sum, cout;
      wire s1,c1,c2,c3;
      xor(s1,x,y);
      xor(sum,cin,s1);
      and(c1,x,y);
      and(c2,y,cin);
      and(c3,x,cin);
      or(cout,c1,c2,c3);
endmodule
```

- **Dataflow modeling:**
  - Mainly used to describe combinational circuits. The basic mechanism used is the continuous assignment.

    `assign` [delay] LHS_net = RHS_expression;

```
module full_adder(x,y,cin,sum,cout);
      input x,y,cin;
      output sum, cout;

      assign {cout, sum} = x + y + cin;

endmodule
```

- **Behavioral modeling**:
  - Used to describe complex circuits (primarily sequential). Mechanisms: **`initial`** and **`always`** statements.

```
module full_adder(x,y,cin,sum,cout);
      input x,y,cin;
      output reg sum, cout;

    always @ (x or y or cin) begin
        {cout, sum} = a + b + cin;
    end
endmodule
```

Getting Started

With Verilog

Describing a digital system as a set of modules

- Modules can have interfaces to other modules (*instantiation* = creating a copy).
- Modules are connected using *nets*.

What can we do?

We'll be doing this.

- Simulation to verify the system (test benches)
- Synthesis to map to hardware (low-level primitives, ASIC, FPGA)

Development Process

Verilog Language Features

Operators

Arithmetic Operators:

+	unary (sign) plus
–	unary (sign) minus
+	binary plus (add)
–	binary minus (subtract)
*	multiply
/	divide
%	modulus
**	exponentiation

Examples:

– (b + c)

(a – b) + (c * d)

(a + b) / (a – b)

a % b

a ** 3

Verilog Language Features

Operators

Logical Operators:

!	logical negation
&&	logical AND
||	logical OR

&&

A	B	A && B
F	F	F
F	T	F
T	F	F
T	T	T

Evaluates to True or False

Examples:

```
(done && ack)
(a || b)
! (a && b)
((a > b) || (c ==0))
((a > b) && ! (b > c))
```

Verilog Language Features

Operators

Bitwise Operators:

~	bitwise NOT
&	bitwise AND
|	bitwise OR
^	bitwise exclusive-OR
~^	bitwise exclusive-NOR

Examples:

```
wire  a, b, c, d, f1, f2, f3, f4;
assign f1 = ~a | b;
assign f2 = (a & b) | (b & c) | (c & a);
assign f3 = a ^ b ^ c;
assign f4 = (a & ~b) | (b & c & ~d);
```

Operate on bits, return a value that is also a bit.

Verilog Language Features

Operators: Logical vs. Bitwise

```
1'b1 && 1'b1        = 1        1'b1 & 1'b1        = 1
1'b1 && 1'b0        = 0        1'b1 & 1'b0        = 0
! 1'b1              = 0        ~ 1'b1             = 0
! 3'b000            = 1        ~ 3'b000           = 111
3'b111 && 3'b111    = 1        3'b111 & 3'b111    = 111
3'b100 && 3'b111    = 0        3'b000 & 3'b111    = 000
```

Verilog Language Features

Operators

Relational Operators:

!=	not equal
==	equal
>=	greater or equal
<=	less or equal
>	greater
<	less

Examples:

```
(a != b)
((a + b) == (c – d))
((a > b) && (c < d))
(count <= 0)
```

Operate on numbers, return True or False

Verilog Language Features

Operators

Shift Operators:

>>	shift right
<<	shift left
>>>	arithmetic shift right

Examples:

```
wire [15:0] data, target;
assign target = data >> 3;
assign target = data >>> 2;
```

>>> :: 2's complement number system

Shift right: ÷ 2
Shift left : * 2

Reduction, conditional, concatenation, and replication operators also available.

Verilog Language Features

Data Values

Verilog supports 4 value levels:

Value Level	Represents
0	Logic 0 state
1	Logic 1 state
x	Unknown logic state
z	High impedance state

- All unconnected nets are set to ‘z’.
- All register variables are set to ‘x’.

Verilog Language Features

Module - the basic unit of hardware in Verilog

- Cannot contain definitions of other modules,
- Can be ***instantiated*** within another module - **hierarchy of modules**.

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

Temporary connections (wires)

Verilog Language Features

Module example: **A simple AND function**

```
// A simple AND function
module simple_AND(t1, a, b);
    input a, b;
    output t1;
    assign t1 = a & b;
endmodule
```

Is this a structural or behavioral description?

Assign statement:

```
assign [delay] LHS_net = RHS_expression;
```

- used **typically** for combinational circuits
- continuous assignment, delay optional
- LHS must be “net” type var (usually “wire”)
- RHS can be both “register” or “net” type

Not synced with clock!

Verilog Language Features

Module example 2: **A 2-level combinational circuit**

```
// A 2-level combinational circuit
module two_level(a, b, c, d, f);
    input a, b, c, d;
    output f;
    wire t1, t2; // intermediate lines
    assign t1 = a & b;
    assign t2 = ~(c | d);
    assign f = ~(t1 & t2);
endmodule
```

Verilog Language Features **Data Types**

```
wire sbar;
assign sbar = ~S;
```

A variable can be:

A. Net **wire, wor, wand, tri, supply0, supply1, etc.**

- Must be continuously driven,
- Cannot be used to store a value,
- Models connections between continuous assignments and instantiations,
- 1-bit values by default, unless declared as vectors explicitly.
- Default value of a *net* is “Z” - high impedance state.

B. Register **reg, integer, real, time**

- Retains the last value assigned to it,
- Usually used to represent storage elements (sometimes in combinational circuits),
- May or may not map to a HW register during synthesis.
- Default value of a *reg* data type is “X”.

Verilog Language Features

Data Types

net example:

```
module use_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
```

Is this a valid design?

Verilog Language Features

Data Types

net example:

```
module use_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
```

For these inputs, f will be indeterminate!

Wrong design!

Verilog Language Features

Data Types

net example - correct design:

```
// A 2-level combinational circuit
module using_wand(a, b, c, d, f);
    input a, b, c, d;
    output f;
    wand f; // net f declared as wand
    assign f = a & b;
    assign f = c | d;
endmodule
```

Here, function realized will be
f = (A & B) & (C | D)

Verilog Language Features

Data Types

net example 2:

```
module using_supply_wire(a, b, c, f);
    input a, b, c;
    output f;
    wire t1, t2;
    supply0 gnd;
    supply1 vcc;
    nand G1 (t1, vcc, a, b);
    xor G2 (t2, c, gnd);
    and G3 (f, t1, t2);
endmodule
```

Is this a structural or behavioral description?

Verilog Language Features

Data Types

reg example:

- Declaration explicitly specifies the size (default is 1-bit):
  - `reg x, y;  // 1-bit register variables`
  - `reg [7:0] bus; // An 8-bit bus`
- Treated as an unsigned number in arithmetic expressions.

For 2's comp. signed use *integer* data type

- **MUST** be used when modeling actual ***sequential*** HW, e.g. counters, shift registers, etc.
- Two types of assignments possible:
  - `A = B + C;`
  - `A <= B + C;`

A must be a *reg* type var

Verilog Language Features

Data Types

Come back to these reg example slides after you learn about sequential circuits!

reg example - 32-bit counter with synchronous reset:

```
module simple_counter(clk, rst, count);
    input clk, rst;
    output [31:0] count;
    reg [31:0] count;

    always @(posedge clk)
    begin
        if(rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```

Because we must have a *reg* type var at LHS

Otherwise: compiler error

If *rst* is high, reset occurs at the positive edge of the next clock.

How to fix this?

Any variable assigned within the *always* block must be of type *reg*.

Verilog Language Features

Data Types

reg example - solution: 32-bit counter with asynchronous reset:

```
module simple_counter(clk, rst, count);
    input clk, rst;
    output [31:0] count;
    reg [31:0] count;

    always @(posedge clk or posedge rst)
    begin
        if(rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```

Reset occurs whenever *rst* goes high.

Verilog Language Features

Data Types

Integer example:

- General purpose register data type,
- 2's complement signed integer in arithmetic expressions,
- Default size is 32 bits.

```
wire [15:0] X, Y;
integer C;

C = X + Y;
```

Synthesis tool deduces that C is 17 bits (16 bits + a carry)

Other register data types: *real*, *time*.

Verilog Language Features

Vectors

- Both **`Net`** or **`reg`** type variables can be declared as vectors: **multiple bit widths**
- Specifying width with: **[MSB:LSB]**

```
wire x, y, z;     // single bit variables
wire[7:0] sum;    // MSB is sum[7], LSB is sum[0]
reg [1:10] data; // MSB is data[1], LSB is data[10]
reg clock;
```

sum[0]
sum[1]
⋮
sum[7]

Verilog Language Features

Vectors

- Parts of a vector can be addressed and used in an expression:

```
reg [31:0] IR;
reg [5:0] opcode;
reg [4:0] reg1, reg2, reg3;
reg [10:0] offset;
```

```
opcode = IR[31:26];
reg1 = IR[25:21];
reg2 = IR[20:16];
reg3 = IR[15:11];
offset = IR[10:0];
```

SUM = IR[25:21] + IR[20:16]

Multi-dimensional arrays and memories also possible.

Verilog Language Features

Constant Values

- **Sized** or **unsized** form,
- Syntax:
  - `<size>'<base><number>`
- Examples:

```
4'b0101        // 4-bit binary number 0101
1'b0           // Logic 0 (1-bit)
12'hB3C        // 12-bit number 1011 0011 1100
12'h8xF        // 12-bit number 1000 xxxx 1111
25             // signed number, in 32 bits (size not specified)
```

Verilog Language Features

Parameters

- Constants with a given name,
- Size deduced from the constant value itself:

```
parameter HI = 5, LO = 0;
parameter up = 2'b00, down = 2'b01, steady = 2'b10;
```

```
// Parameterized design:
// an N-bit counter
module counter(clk, rst, count);
    parameter N = 31;
    input clk, rst;
    output [0:N] count;
    reg [0:N] count;

    always @(negedge clk)
    begin
        if (rst)
            count = 0;
        else
            count = count + 1;
    end
endmodule
```

Verilog Language Features

Predefined Logic Gates

- Can be instantiated within a module to create a structural (gate-level) design.

2-input AND	2-input OR	2-input EXOR
0 & 0 = 0	0 | 0 = 0	0 ^ 0 = 0
0 & 1 = 0	0 | 1 = 1	0 ^ 1 = 1
1 & 1 = 1	1 | 1 = 1	1 ^ 1 = 0
1 & x = x	1 | x = 1	1 ^ x = x
0 & x = 0	0 | x = x	0 ^ x = x
1 & z = x	1 | z = x	1 ^ z = x
z & x = x	z | x = x	z ^x = x

Remember that Verilog supports 4 value levels.

Verilog Language Features

There are also other gates with tristate control

List of Some Primitive Gates

```
and Gate1 (out, in1, in2);
nand Gate2 (out, in1, in2);
or Gate3 (out, in1, in2);
nor Gate4 (out, in1, in2);
xor Gate5 (out, in1, in2);
xnor Gate6 (out, in1, in2);
not Gate7 (out, in1);
```

Number of inputs can be arbitrary.

- Output ports must be connected to a *net*
- Input ports may be either *net* or *reg* type vars

```
and G (a, b, c, d, e, f);
```

output

inputs

A 5-input AND gate

Verilog Language Features

Two Ways to Specify Connectivity During Module Instantiation

Connectivity of the signal lines between two modules can be specified in 2 ways:

- **Positional Association (Implicit)**
  - Parameters listed **in the same order** as in the original module description.
- **Explicit Association**
  - Parameters **explicitly** listed **in arbitrary order**.

Verilog Language Features

- **Positional Association Example - Full Adder Using Half Adder Module**

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Verilog Language Features

- **Positional Association Example - Full Adder Using Half Adder Module**

```
module half_adder (Sum, Carry, A, B);
    input A, B;
    output Carry, Sum;
    //structural description
    xor G1(Sum, A, B);
    and G2(Carry, A, B);
endmodule
```

What is the equivalent behavioral design?

Verilog Language Features

- **Positional Association Example - Full Adder Using Half Adder Module**

```
module half_adder (Sum, Carry, A, B);
    input A, B;
    output Carry, Sum;
    //structural description
    xor G1(Sum, A, B);
    and G2(Carry, A, B);
endmodule
```

=

```
//behavioral description
assign Sum = A ^ B;
assign Carry = A & B;
```

Verilog Language Features

- **Positional Association Example - Full Adder Using Half Adder Module**

Note the port order

```
module half_adder (Sum, Carry, A, B);
```

```
module full_adder (Sum, Cout, A, B, Cin);
    input A, B, Cin;
    output Cout, Sum;
    wire w1, w2, w3;

endmodule
```

How do we use the half_adder module to implement a full adder?

Verilog Language Features

- **Positional Association Example - Full Adder Using Half Adder Module**

Note the port order

```
module half_adder (Sum, Carry, A, B);
```

```
module full_adder (Sum, Cout, A, B, Cin);
    input A, B, Cin;
    output Cout, Sum;
    wire w1, w2, w3;
    half_adder HA1 (w1, w2, A, B);
    half_adder HA2 (Sum, w3, Cin, w1);
    or (Cout, w2, w3);
endmodule
```

Verilog Language Features

- **Explicit Association Example - Full Adder Using Half Adder Module**

```
module half_adder (Sum, Carry, A, B);
```

Ports are explicitly specified - order is not important

```
module full_adder (Sum, Cout, A, B, Cin);
    input A, B, Cin;
    output Cout, Sum;
    wire w1, w2, w3;
    half_adder HA1 (.A(A), .B(B), .Sum(w1), .Carry(w2));
    half_adder HA2 (.Sum(Sum), .Carry(w3), .B(Cin), .A(w1));
    or (Cout, w2, w3);
endmodule
```

Less chance for errors

Verilog Language Features

Combinational vs. Sequential Circuits

Combinational: The output only depends on the present input.

Sequential: The output depends on both the present input and the previous output(s) (the state of the circuit).

Verilog Language Features

Combinational vs. Sequential Circuits

Sequential logic: **Blocks that have memory elements**: Flip-Flops, Latches, Finite State Machines.

- Triggered by a ‘clock’ event.
  - Latches are sensitive to level of the signal.
  - Flip-flops are sensitive to the transitioning of clock

Combinational constructs are not sufficient. We need new constructs:

- `always`
- `initial`

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

Verilog Language Features

Sequential Circuits

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

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

Remember our counter example

```
module simple_counter(clk, rst, count);
    input clk, rst;
    output [31:0] count;
    reg [31:0] count;

    always @(posedge clk)
    begin
        if(rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```

Verilog Language Features

Sequential Circuits

- Sequential statements are within an **`'always'`** block,
- The sequential block is triggered with a change in the sensitivity list,
- Signals assigned within an **`always`** block must be declared as **`reg`**,
  - The values are preserved (memorized) when no change in the sensitivity list.
- We do not use **`'assign'`** within the **`always`** block.

- **Always blocks allow powerful statements**
  - `if .. then .. else`
  - `case`

Difference between **Synchronous** and **Asynchronous** Sequential Circuits

SYNCHRONOUS CIRCUIT	ASYNCHRONOUS CIRCUIT
All the **State Variable** changes are synchronized with a universal clock signal.	The **State Variables** are not synchronized to change simulteneously and may change at anytime irrespective of each other to achieve the next **Steady Internal State**
Since all the Internal State changes are in the strict control of a master clock source they are less prone to failure or to a race condition and hence are more reliable.	Since there is no such universal clock source, the internal state changes as soon as any of the inputs change and hence are more prone to a race condition.
Timings of the internal state changes are in our control.	The changes in the internal state of an asynchronous circuit are not in our control.

Verilog Language Features

Sequential Circuits

32-bit counter with *synchronous* reset:

```
module simple_counter(clk, rst, count);
    input clk, rst;
    output [31:0] count;
    reg [31:0] count;

    always @(posedge clk)
    begin
        if(rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```

Because we must have a *reg* type var at LHS

Otherwise: compiler error

How to make this async.?

If *rst* is high, reset occurs at the positive edge of the next clock.

Any variable assigned within the *always* block must be of type *reg*.

Verilog Language Features

Sequential Circuits

Solution: 32-bit counter with *asynchronous* reset:

```
module simple_counter(clk, rst, count);
    input clk, rst;
    output [31:0] count;
    reg [31:0] count;

    always @(posedge clk or posedge rst)
    begin
        if(rst)
            count = 32'b0;
        else
            count = count + 1;
    end
endmodule
```

Reset occurs whenever *rst* goes high.

Non-blocking and Blocking Statements

Non-blocking

```
always @ (a)
begin
   a <= 2'b01;
   b <= a;
// all assignments are made here
// b is not (yet) 2'b01
end
```

- Values are assigned at the end of the block.
- All assignments are made in parallel, process flow is not-blocked.

Blocking

```
always @ (a)
begin
   a = 2'b01;
// a is 2'b01
   b = a;
// b is now 2'b01 as well
end
```

- Value is assigned immediately.
- Process waits until the first assignment is complete, it blocks progress.

Blocking statements allow sequential descriptions

How to Simulate Verilog Module(s)

Testbench: provides stimulus to Unit-Under-Test (UUT) to verify its functionality, captures and analyzes the outputs.

Requirements:
Inputs and outputs need to be connected to the test bench

How to Simulate Verilog Module(s) Example

Suppose we want to design and simulate this circuit.

We can choose either *behavioral* or *structural* design.

How to Simulate Verilog Module(s) Example

Let's choose structural gate-level design:

```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description

endmodule
```

Which gates do we need, and what will the connections be?

How to Simulate Verilog Module(s) Example

Let's choose structural gate-level design:

```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);



endmodule
```

How to Simulate Verilog Module(s) Example

Let's choose structural gate-level design:

```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);


endmodule
```

How to Simulate Verilog Module(s) Example

Let's choose structural gate-level design:

```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);

endmodule
```

How to Simulate Verilog Module(s) **Example**

Let's choose structural gate-level design:

```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);
    nand G4(Y, t1, t2, t3);
endmodule
```

Now we need to provide stimulus and monitor the outputs - TESTBENCH

How to Simulate Verilog Module(s) Example

Saved as function_Y_testbench.v

TESTBENCH

Saved as function_Y.v

Unit Under Test

```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);
    nand G4(Y, t1, t2, t3);
endmodule
```

```
module function_Y_testbench;

endmodule
```

Note there are no ports in a testbench!

How to Simulate Verilog Module(s) Example

Saved as function_Y_testbench.v

TESTBENCH

Saved as function_Y.v

Unit Under Test

```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);
    nand G4(Y, t1, t2, t3);
endmodule
```

```
module function_Y_testbench;
    reg A, B, C, D, E, F;
    wire Y;

endmodule
```

Vars MUST be declared as *reg*
Output as wire

How to Simulate Verilog Module(s) Example

Saved as function_Y.v

Unit Under Test

```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);
    nand G4(Y, t1, t2, t3);
endmodule
```

Saved as function_Y_testbench.v

TESTBENCH

```
module function_Y_testbench;
    reg A, B, C, D, E, F;
    wire Y;

    function_Y UUT(A, B, C, D, E, F, Y);

endmodule
```

Initialize Unit Under Test (UUT)

How to Simulate Verilog Module(s) Example

Saved as function_Y.v

Unit Under Test

```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);
    nand G4(Y, t1, t2, t3);
endmodule
```

Saved as function_Y_testbench.v

TESTBENCH

```
`include "function_Y.v"
module function_Y_testbench;
    reg A, B, C, D, E, F;
    wire Y;

    function_Y UUT(A, B, C, D, E, F, Y);

    initial
        begin



        end
endmodule
```

initial block - gets executed once

How to Simulate Verilog Module(s) Example

Saved as function_Y.v

Unit Under Test

```
module function_Y (A, B, C, D, E, F, Y);
    input A, B, C, D, E, F;
    output Y;
    wire t1, t2, t3, Y;
    //structural description
    nand G1(t1, A, B);
    and G2(t2, C, ~B, D);
    nor G3(t3, E, F);
    nand G4(Y, t1, t2, t3);
endmodule
```

Saved as function_Y_testbench.v

TESTBENCH

```
`include "function_Y.v"
module function_Y_testbench;
    reg A, B, C, D, E, F;
    wire Y;

    function_Y UUT(A, B, C, D, E, F, Y);

    initial
        begin
        #10 A = 0; B = 0; C = 0; D = 0; E = 0; F = 0;
        #10 A = 1; B = 0; C = 1; D = 1; E = 0; F = 0;
        #10 A = 0; B = 1;
        #10 F = 1;
        #10 $finish;
        end
endmodule
```

Stimulus

How to Simulate Verilog Module(s) Example

Results can be viewed as waveforms:

How to Simulate Verilog Module(s) **Example**

We can also monitor the changes and print them to the console using *$monitor*:

```
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);
    #10 A = 0; B = 0; C = 0; D = 0;
    #10 A = 1; B = 0; C = 1; D = 1;
    #10 A = 0; B = 1;
    #10 F = 1;
    #10 $finish;
    end
```

Tcl Console | Messages | Log

```
# run 1000ns
                    0A = x, B = x, C = x, D = x, E = x, F = x, Y = x
                   10A = 0, B = 0, C = 0, D = 0, E = 0, F = 0, Y = 1
                   20A = 1, B = 0, C = 1, D = 1, E = 0, F = 0, Y = 0
                   30A = 0, B = 1, C = 1, D = 1, E = 0, F = 0, Y = 1
                   40A = 0, B = 1, C = 1, D = 1, E = 0, F = 1, Y = 1
$finish called at time : 50 ns : File "C:/Users/selma/Google Drive/xilinx/starter_proje
INFO: [USF-XSim-96] XSim completed. Design snapshot 'helper_module_testbench_behav' loa
INFO: [USF-XSim-97] XSim simulation ran for 1000ns
launch_simulation: Time (s): cpu = 00:00:04 ; elapsed = 00:00:16 . Memory (MB): peak =
```

Type a Tcl command here

We can also use ***$dumpfile*** to dump variable changes to a file.

Short Practice Example 1

Implement a **the circuit given below** using **structural gate-level** design approach.

Short Practice Example 1

Implement a **the circuit given below** using **structural gate-level** design approach.

```
module Simple Circuit (A, B, C, D, E);

endmodule
```

Short Practice Example 1

Implement a **the circuit given below** using **structural gate-level** design approach.

```
module  Simple_Circuit (A, B, C, D, E);
 output        D, E;
 input         A, B, C;
 wire          w1;



endmodule
```

Short Practice Example 1

Implement a **the circuit given below** using **structural gate-level** design approach.

```
module  Simple_Circuit (A, B, C, D, E);
 output        D, E;
 input         A, B, C;
 wire          w1;

 and           G1 (w1, A, B); // Optional gate instance name


endmodule
```

Short Practice Example 1

Implement a **the circuit given below** using **structural gate-level** design approach.

```
module  Simple_Circuit (A, B, C, D, E);
 output        D, E;
 input         A, B, C;
 wire          w1;

 and           G1 (w1, A, B); // Optional gate instance name
 not           G2 (E, C);

endmodule
```

Short Practice Example 1

Implement a **the circuit given below** using **structural gate-level** design approach.

```
module  Simple_Circuit (A, B, C, D, E);
  output        D, E;
  input         A, B, C;
  wire          w1;

  and           G1 (w1, A, B); // Optional gate instance name
  not           G2 (E, C);
  or            G3 (D, w1, E);
endmodule
```

Short Practice Example 1

Implement a **testbench** for the Simple_Circuit.

```
module  Simple_Circuit (A, B, C, D, E);
 output        D, E;
 input         A, B, C;
 wire          w1;

 and           G1 (w1, A, B); // Optional gate instance name
 not           G2 (E, C);
 or            G3 (D, w1, E);
endmodule
```

```
// Test bench for Simple_Circuit

module  t_Simple_Circuit;



endmodule
```

Short Practice Example 1

Implement a **testbench** for the Simple_Circuit.

```
module  Simple_Circuit (A, B, C, D, E);
 output        D, E;
 input         A, B, C;
 wire          w1;

 and           G1 (w1, A, B); // Optional gate instance name
 not           G2 (E, C);
 or            G3 (D, w1, E);
endmodule
```

```
// Test bench for Simple_Circuit

module  t_Simple_Circuit;
 wire     D, E;
 reg      A, B, C;



endmodule
```

Short Practice Example 1

Implement a **testbench** for the Simple_Circuit.

```
module  Simple_Circuit (A, B, C, D, E);
 output        D, E;
 input         A, B, C;
 wire          w1;

 and           G1 (w1, A, B); // Optional gate instance name
 not           G2 (E, C);
 or            G3 (D, w1, E);
endmodule
```

```
// Test bench for Simple_Circuit

module  t_Simple_Circuit;
 wire    D, E;
 reg     A, B, C;

Simple_Circuit M1 (A, B, C, D, E); // Instance name required



endmodule
```

Short Practice Example 1

Implement a **testbench** for the Simple_Circuit.

```
module  Simple_Circuit (A, B, C, D, E);
 output        D, E;
 input         A, B, C;
 wire          w1;

 and           G1 (w1, A, B); // Optional gate instance name
 not           G2 (E, C);
 or            G3 (D, w1, E);
endmodule
```

```
// Test bench for Simple_Circuit

module  t_Simple_Circuit;
 wire    D, E;
 reg     A, B, C;

Simple_Circuit M1 (A, B, C, D, E); // Instance name required

initial
 begin


 end


endmodule
```

Short Practice Example 1

Implement a **testbench** for the Simple_Circuit.

```
module  Simple_Circuit (A, B, C, D, E);
 output        D, E;
 input         A, B, C;
 wire          w1;

 and           G1 (w1, A, B); // Optional gate instance name
 not           G2 (E, C);
 or            G3 (D, w1, E);
endmodule
```

```
// Test bench for Simple_Circuit

module  t_Simple_Circuit;
 wire    D, E;
 reg     A, B, C;

Simple_Circuit M1 (A, B, C, D, E); // Instance name required

initial
  begin
    A = 1'b0; B = 1'b0; C = 1'b0;
    #100 A = 1'b1; B = 1'b1; C = 1'b1;
  end


endmodule
```

Short Practice Example 1

Implement a **testbench** for the Simple_Circuit.

```
module  Simple_Circuit (A, B, C, D, E);
 output        D, E;
 input         A, B, C;
 wire          w1;

 and           G1 (w1, A, B); // Optional gate instance name
 not           G2 (E, C);
 or            G3 (D, w1, E);
endmodule
```

```
// Test bench for Simple_Circuit

module  t_Simple_Circuit;
 wire     D, E;
 reg      A, B, C;

Simple_Circuit M1 (A, B, C, D, E); // Instance name required

initial
  begin
    A = 1'b0; B = 1'b0; C = 1'b0;
    #100 A = 1'b1; B = 1'b1; C = 1'b1;
  end

 initial #200 $finish;
endmodule
```

Short Practice Example 1

Implement a **testbench** for the Simple_Circuit.

```
module  Simple_Circuit (A, B, C, D, E);
 output        D, E;
 input         A, B, C;
 wire          w1;

 and           G1 (w1, A, B); // Optional gate instance name
 not           G2 (E, C);
 or            G3 (D, w1, E);
endmodule
```

```
// Test bench for Simple_Circuit

module  t_Simple_Circuit;
 wire    D, E;
 reg     A, B, C;

Simple_Circuit M1 (A, B, C, D, E); // Instance name required

initial
  begin
    A = 1'b0; B = 1'b0; C = 1'b0;
    #100 A = 1'b1; B = 1'b1; C = 1'b1;
  end

 initial #200 $finish;
endmodule
```

Is the delay in ms, ns...?

Short Practice Example 1

Implement a **testbench** for the Simple_Circuit.

```
module  Simple_Circuit (A, B, C, D, E);
 output        D, E;
 input         A, B, C;
 wire          w1;

 and           G1 (w1, A, B); // Optional gate instance name
 not           G2 (E, C);
 or            G3 (D, w1, E);
endmodule
```

```
`timescale 1ns/100ps
module  t_Simple_Circuit;
 wire    D, E;
 reg     A, B, C;

Simple_Circuit M1 (A, B, C, D, E); // Instance name required

initial
  begin
    A = 1'b0; B = 1'b0; C = 1'b0;
    #100 A = 1'b1; B = 1'b1; C = 1'b1;
  end

 initial #200 $finish;
endmodule
```

You should specify the timescale

Short Practice Example 2

Implement the circuit specified by **the given Boolean equations** using **dataflow** design approach.

$$E = A + BC + B'D$$

$$F = B'C + BC'D'$$

Short Practice Example 2

Implement the circuit specified by **the given Boolean equations** using **dataflow** design approach.

$$E = A + BC + B'D$$
$$F = B'C + BC'D'$$

```
// Verilog model: Circuit with Boolean expressions
module Circuit_Boolean_CA (E, F, A, B, C, D);



endmodule
```

Short Practice Example 2

Implement the circuit specified by **the given Boolean equations** using **dataflow** design approach.

$$E = A + BC + B'D$$
$$F = B'C + BC'D'$$

```
// Verilog model: Circuit with Boolean expressions

module Circuit_Boolean_CA (E, F, A, B, C, D);
  output      E, F;
  input       A, B, C, D;



endmodule
```

Short Practice Example 2

Implement the circuit specified by **the given Boolean equations** using **dataflow** design approach.

$$E = A + BC + B'D$$
$$F = B'C + BC'D'$$

```
// Verilog model: Circuit with Boolean expressions

module Circuit_Boolean_CA (E, F, A, B, C, D);
  output      E, F;
  input       A, B, C, D;

  assign E = A || (B && C) || ((!B) && D);
  assign F = ((!B) && C) || (B && (!C) && (!D));
endmodule
```

Short Practice Example 2

Implement the circuit specified by **the given Boolean equations** using **dataflow** design approach.

$$E = A + BC + B'D$$
$$F = B'C + BC'D'$$

```
// Verilog model: Circuit with Boolean expressions

module Circuit_Boolean_CA (E, F, A, B, C, D);
  output      E, F;
  input       A, B, C, D;

  assign E = A || (B && C) || ((!B) && D);
  assign F = ((!B) && C) || (B && (!C) && (!D));
endmodule
```

Think about how a TESTBENCH would look like

Questions?

Lab Example

Implement a 4-Bit Ripple Carry Adder in Verilog in the following steps:

1. Implement a **1-Bit Full Adder** using **behavioral dataflow** design approach. Fill in the truth table, find the corresponding functions for *Sum* and *Carry_out*, write a Verilog module, test it by writing a testbench for all possible cases.

2. Implement a **4-Bit Ripple Carry Adder** by instantiating your 1-Bit Full Adder module as many times as necessary. Use **structural** design approach and **explicit association**. Test it by writing an appropriate testbench.

Lab Example

Solution:

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

Full_Adder.v module

```
module Full_Adder(
    input A,
    input B,
    input Cin,
    output S,
    output Cout
    );
    assign S = (A^B)^Cin;
    assign Cout = (A&B)|(B&Cin)|(Cin&A);
endmodule
```

Note the behavioral description

```
sum = (A^B)^C
Cout = AB+BC+AC
```

Full_Adder_Testbench.v module

```
module Full_Adder_Testbench;
    //inputs
    reg A, B, Cin;
    //outputs
    wire S, Cout;
    // Instantiate the Unit Under Test (UUT)
    Full_Adder UUT(.A(A), .B(B), .Cin(Cin), .S(S), .Cout(Cout));
    //Provide stimulus
   initial begin
    // Initialize Inputs
    A = 0; B = 0; Cin = 0;
    #10 A = 0; B = 0; Cin = 1;
    #10 A = 0; B = 1; Cin = 0;
    #10 A = 0; B = 1; Cin = 1;
    #10 A = 1; B = 0; Cin = 0;
    #10 A = 1; B = 0; Cin = 1;
    #10 A = 1; B = 1; Cin = 0;
    #10 A = 1; B = 1; Cin = 1;
    #10 $finish;
   end
```

Simulation results:

Four_Bit_RCA.v module

```
`include "Full_Adder.v"
module Four_Bit_RCA(
    input [3:0] A,
    input [3:0] B,
    input Cin,
    output [3:0] S,
    output Cout
    );
    wire [2:0] Carries;
    Full_Adder A1(.A(A[0]),.B(B[0]),.Cin(Cin),.S(S[0]),.Cout(Carries[0]));
    Full_Adder A2(.A(A[1]),.B(B[1]),.Cin(Carries[0]),.S(S[1]),.Cout(Carries[1]));
    Full_Adder A3(.A(A[2]),.B(B[2]),.Cin(Carries[1]),.S(S[2]),.Cout(Carries[2]));
    Full_Adder A4(.A(A[3]),.B(B[3]),.Cin(Carries[2]),.S(S[3]),.Cout(Cout));
endmodule
```

Note the structural description and explicit association

Four_Bit_RCA_Testbench.v module

```
module Four_Bit_RCA_Testbench;
    //inputs
    reg [3:0] A, B;
    reg Cin;
    //outputs
    wire [3:0] S;
    wire Cout;
    // Instantiate the Unit Under Test (UUT)
    Four_Bit_RCA UUT(.A(A), .B(B), .Cin(Cin), .S(S), .Cout(Cout));
    //Provide stimulus
   initial begin
    // Initialize Inputs
    A = 4'b0000; B = 4'b0000; Cin = 0;
    #10 A = 4'b0000; B = 4'b0000; Cin = 1;
    #10 A = 4'b1100; B = 4'b0011; Cin = 0;
    #10 A = 4'b1100; B = 4'b0011; Cin = 1;
    #10 A = 4'b1110; B = 4'b0001; Cin = 1;
    #10 A = 4'b1100; B = 4'b0011; Cin = 1;
    #10 A = 4'b1111; B = 4'b0001; Cin = 0;
    #10 A = 4'b1111; B = 4'b1111; Cin = 1;
    #10 $finish;
   end
```

Only some test cases. Why not all?

How can we automate the process?

Lab Example 2 - Sequential Circuits

Figure: Sequential Circuit

Lab Example 2 - Sequential Circuits

Storage Elements

Latches: level-sensitive

Flip-flops: edge-sensitive

Lab Example 2 - Sequential Circuits

A flip flop is a basic building block of sequential logic circuits. It is a circuit that has two stable states and can store one bit of state information. The output changes state by signals applied to one or more control inputs.

Edge-Triggered D Flip Flop

Characteristic Equation
Q(t+1) = D

Inputs	Present Output	Next Output
D	Q_n	Q_{n+1}
0	0	0
0	1	0
1	0	1
1	1	1

A basic D Flip Flop has a D (data) input, a clock (CLK) input and outputs Q and Q' (the inverse of Q). Optionally it may also include the PR (Preset) and CLR (Clear) control inputs.

Lab Example 2 - Sequential Circuits

Verilog codes for different implementations of D flip-flop:

Verilog code for Rising Edge D Flip Flop:

```
module RisingEdge_DFlipFlop(D,clk,Q);
input D; // Data input
input clk; // clock input
output reg Q; // output Q
always @(posedge clk)
begin
 Q <= D;
end
endmodule
```

Verilog code for Falling Edge D Flip Flop:

```
module FallingEdge_DFlipFlop(D,clk,Q);
input D; // Data input
input clk; // clock input
output reg Q; // output Q
always @(negedge clk)
begin
 Q <= D;
end
endmodule
```

Lab Example 2 - Sequential Circuits

Verilog codes for different implementations of D flip-flop:

Verilog code for Rising Edge D Flip Flop with Synchronous Reset:

```
module RisingEdge_DFlipFlop_SyncReset(D,clk,sync_reset,Q);
input D; // Data input
input clk; // clock input
input sync_reset; // synchronous reset
output reg Q; // output Q
always @(posedge clk)
begin
 if(sync_reset==1'b1)
  Q <= 1'b0;
 else
  Q <= D;
end
endmodule
```

Lab Example 2 - Sequential Circuits

Sequential Circuit Design (With an Example)

Here we state the steps of designing a sequential circuit with an example circuit specification:

111 Sequence Detector:
Design a circuit that outputs 1 when a sequence three consecutive 1's is applied to input, and 0 otherwise.

Input	X	0 1 1 1 0 0 0 1 1 1 1 1 0 0 0
Output	Y	0 0 0 1 0 0 0 0 0 1 1 1 0 0 0

Lab Example 2 - Sequential Circuits

To design a sequential circuit, start with the problem definition and use the following steps:

- **Step-1**: **Create a state transition diagram from the description. Reduce the number of states if necessary, and assign binary values to the states.**

State diagram for 111 Sequence Detector with assigned binary values to the states.

Lab Example 2 - Sequential Circuits

- Step-2: Convert the state transition diagram into a state transition table (binary coded state table).

State Table for Sequence Detector

Present State		Input	Next State		Output
A	*B*	*x*	*A*	*B*	*y*
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

Lab Example 2 - Sequential Circuits

- **Step-3: Determine the number of flip-flops needed and choose flip-flop types. Derive their excitation tables (if designing a sequential circuit with flip-flops other than the D type), and derive input and output equations from the state table. Minimize the functions for the flip-flop inputs (e.g. using Karnaugh Maps).**

We choose D-type flip-flops. Since there are four states, we need two D flip-flops, and we label their outputs as A and B. The characteristic equation of the D flip-flop is `Q(t + 1) = D`, which means that the next-state values in the state table specify the D input condition for the flip-flop.

Lab Example 2 - Sequential Circuits

State Table for Sequence Detector

Present State		Input	Next State		Output
A	B	x	A	B	y
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

Characteristic Equation
Q(t+1) = D

From the state table above we obtain the following input equations:

$$D_A = Ax + Bx$$

$$D_B = Ax + B'x$$

$$y = AB$$

Lab Example 2 - Sequential Circuits

- Step-4: Use simplified functions for D1 and D2 to design sequential circuit.
- Step-5: Finally determine the combinatorial circuit to represent the output (if any).

$$D_A = Ax + Bx$$

$$D_B = Ax + B'x$$

$$y = AB$$

Lab Example 2 - Sequential Circuits

Verilog Code for the 111 Sequence Detector

Structural design using flip-flops:

Let's choose rising-edge synchronous reset D flip-flop

```
module RisingEdge_DFlipFlop_SyncReset(D,clk,sync_reset,Q);
input D; // Data input
input clk; // clock input
input sync_reset; // synchronous reset
output reg Q; // output Q
always @(posedge clk)
begin
 if(sync_reset==1'b1)
  Q <= 1'b0;
 else
  Q <= D;
end
endmodule
```

Verilog Code for the 111 Sequence Detector

$D_A = Ax + Bx$

$D_B = Ax + B'x$

$y = AB$

```
module three_1s_seq_detector(
    input x_in,
    input clock,
    input reset,
    output y);

    wire [1:0] state;

    D_ff_reset DA(
        .D((state[1] & x_in) | (state[0] & x_in)),
        .clk(clock),
        .Q(state[1]),
        .reset(reset)
    );
    D_ff_reset DB(
        .D((state[1] & x_in) | (~state[0] & x_in)),
        .clk(clock),
        .Q(state[0]),
        .reset(reset)
    );

    assign y = state[1] & state[0];
endmodule
```

```
module D_ff_reset(D,clk,reset,Q);
    input D; // Data input
    input clk; // clock input
    input reset; // synchronous reset
    output reg Q; // output Q

    always @(posedge clk)
    begin
        if(reset==1'b1)
        Q <= 1'b0;
        else
        Q <= D;
    end
endmodule
```

Lab Example 2 - Sequential Circuits

Verilog Code for the 111 Sequence Detector

Alternative structural design using D Flip-flops without reset:

```
module D_ff(D, clk, Q); // Rising-edge D flip flop
    input D; // Data input
    input clk; // clock input
    output reg Q; // output Q

    always @(posedge clk)
    begin
        Q <= D;
    end
endmodule
```

Let's choose rising-edge D flip-flop

Verilog Code for the 111 Sequence Detector

Remember the equations:

$$D_A = Ax + Bx$$

$$D_B = Ax + B'x$$

$$y = AB$$

```
`include "D_ff.v"
module three_1s_seq_detector(
    input x_in,
    input clock,
    input reset,
    output y);

    reg [1:0] present_state = 2'b00;
    wire [1:0] next_state;

    D_ff D1 (
        .D((present_state[1]&x_in)|(present_state[0]&x_in)),
        .clk(clock),
        .Q(next_state[1])
    );

    D_ff D0 (
        .D((present_state[1]&x_in)|(~present_state[0]&x_in)),
        .clk(clock),
        .Q(next_state[0])
    );

    always @(reset or next_state) begin
      if (reset) begin present_state <= 2'b00; end
      else begin present_state <= next_state; end
    end
    assign y = present_state[1] & present_state[0];
endmodule
```

```
module D_ff(D, clk, Q); // Rising-edge D flip flop
    input D; // Data input
    input clk; // clock input
    output reg Q; // output Q

    always @(posedge clk)
    begin
        Q <= D;
    end
endmodule
```

Let's look at the code in parts...

Verilog Code for the 111 Sequence Detector

```
`include "D_ff.v"

module three_1s_seq_detector(
    input x_in,
    input clock,
    input reset,
    output y);

reg [1:0] present_state = 2'b00;


wire [1:0] next_state;
```

Include D_ff module

Declare I/O ports

present_state[1] to store A
present_state[0] to store B

next_state[1] to represent D_A
next_state[0] to represent D_B

Verilog Code for the 111 Sequence Detector

Instantiating D flip flops:

```
D_ff D1 (
        .D((present_state[1]&x_in)|(present_state[0]&x_in)),
        .clk(clock),
        .Q(next_state[1])
);
```

$$D_A = Ax + Bx$$

```
D_ff D0 (
        .D((present_state[1]&x_in)|(~present_state[0]&x_in)),
        .clk(clock),
        .Q(next_state[0])
);
```

$$D_B = Ax + B'x$$

Verilog Code for the 111 Sequence Detector

Handle reset and update present_state on change of next_state:

```
always @(reset or next_state) begin
  if (reset) begin present_state <= 2'b00; end
  else begin present_state <= next_state; end
end
```

How would you make output y implemented in structural design approach as well?

Combinational circuit to implement the output:

```
assign y = present_state[1] & present_state[0];
```

$$y = AB$$

Verilog Code for the 111 Sequence Detector

An alternative behavioral design in Verilog:

```
module three_1s_seq_detector_behavioral(
    input x_in,
    input clock,
    input reset,
    output y);
    parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;
    reg [1:0] state, next_state;

    always@(posedge clock, negedge reset)
        if (reset == 0) state <= s0;
        else state <= next_state;

    always@(state, x_in) begin

        case(state)
            s0: if (x_in == 0) next_state = s0;
            else if (x_in == 1) next_state = s1;
            s1: if (x_in == 0) next_state = s0;
            else if (x_in == 1) next_state = s2;
            s2: if (x_in == 0) next_state = s0;
            else if (x_in == 1) next_state = s3;
            s3: if (x_in == 0) next_state = s0;
            else if (x_in == 1) next_state = s3;
            default: next_state = s0;
        endcase
    end
    assign y = state[1] & state[0];
endmodule
```

Note different reset implementation

Note multiple always blocks

Solution No. 2 - Behavioral design

Verilog Code for the 111 Sequence Detector

An alternative behavioral design in Verilog:

```
parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10, s3 = 2'b11;
```

Use parameter to give constant names to your states

```
always@(state, x_in) begin
    case(state)
        s0: if (x_in == 0) next_state = s0;
        else if (x_in == 1) next_state = s1;
        s1: if (x_in == 0) next_state = s0;
        else if (x_in == 1) next_state = s2;
        s2: if (x_in == 0) next_state = s0;
        else if (x_in == 1) next_state = s3;
        s3: if (x_in == 0) next_state = s0;
        else if (x_in == 1) next_state = s3;
        default: next_state = s0;
    endcase
end
```

Whenever input x or state change

Lab Example 2 - Sequential Circuits

How would we design a testbench for a sequence detector?

The idea is to have an input bit stream (e.g. a 20-bit binary sequence),

```
input_data = 20'b00001110100111100000;
```

and send one input bit at a time at each clock cycle (e.g. using shifting).

```
always @ (posedge clk) begin
      x = input_data>>shift_amount; // get the next input bit
      shift_amount=shift_amount+1;
end
```

Entire sample testbench code:

```
module three_1s_seq_detector_tb;
    // Inputs
      reg x;
      reg clk;
      reg rst;
      // Outputs
      wire y;
      // Instantiate the Unit Under Test (UUT)
      three_1s_seq_detector uut (.x_in(x), .clock(clk), .reset(rst), .y(y));
      reg [19:0] input_data;
      integer shift_amount;
      initial begin
            // Initialize Inputs
            input_data = 20'b00001110100111100000;
            shift_amount=0;
            rst = 1; #50;
            rst = 0; #500; $finish;
      end
      initial begin // Generate clock
            clk = 0;
            forever begin
                  #10;
                  clk = ~clk; // change every 10ns
            end
      end
      always @ (posedge clk) begin
            x = input_data>>shift_amount; // get the next input bit
            shift_amount=shift_amount+1;
      end
endmodule
```

References

- Slides are mostly based on: NPTEL Online Certification Course on Hardware Modeling Using Verilog, by Prof. Indranil Sengupta, Department of Computer Science and Engineering, Indian Institute of Technology Kharagpur - Available online at: https://www.youtube.com/playlist?list=PLUtfVcb-iqn-EkuBs3arreilxa2UKIChl

- Digital Design, M. Morris Mano and Michael D. Ciletti, Prentice Hall.

- Verilog for Sequential Circuits, Design of Digital Circuits 2014, Srdjan Capkun, Frank K. Gürkaynak. Available online at: https://syssec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/system-security-group-dam/education/Digitaltechnik_14/09_Verilog_Sequential.pdf

- Digital VLSI Systems Design, A Design Manual for Implementation of Projects on FPGAs and ASICs Using Verilog, Dr. Seetharaman Ramachandran, Springer.