UMM AL-QURA UNIVERSITY

College of Computer and Information Systems

Computer Engineering Department

1403371

Advanced Logic Design

Lab Manual

Student Name: _____

Student ID: _____

Section: _____ Group: _____

Session (Fall / Spring / Summer) _____

This page is intentionally left blank

TABLE OF CONTENTS

Table of Contents……i

Laboratory Safety Guidelines……ii

Experiment No. 1. Introduction to Verilog HDL…….…….….1

Experiment No. 2. Half Adder Implementation using (Gate Level Modeling)……...16

Experiment No. 3. 4-to-1 MUX Implementation using (Gate Level Modeling)…….....18

Experiment No. 4. 2-to-4 Decoder using (Gate Level Modeling)……....21

Experiment No. 5. 4-bit Full Adder using (Gate Level Modeling)…….....24

Experiment No. 6. 4-to-1 MUX using (Dataflow Modeling)……..…….…….....27

Experiment No. 7. 32-bit Adder / Subtractor (Dataflow Modeling)…….…...….29

Experiment No. 8. 32-bit Magnitude Comparator using (Dataflow Modeling)……..….31

Experiment No. 9. Ripple Carry Counter Simulation (Behavioral Modeling)……...….33

Experiment No. 10. Up-down Circular Counter using Asynchronous Reset……...36

Experiment No. 11. State Machine …….……..……..39

Experiment No. 12. Speed Machine……...……..43

Prepared By:	Engr. Muhammad Saqib	2012
Revised By:	Engr. Muhammad Farhan	2013
Reviewed By:	_____	_____
Approved By:	_____	_____

Umm Al-Qura University

Computer Engineering Department

LABORATORY SAFETY GUIDELINES

A. General Laboratory Safety Rules

1. Personal Safety

- Be familiar with the electrical and fire hazards associated with your workplace.
- Be as careful for the safety of others as for yourself. Think before you act.
- Be tidy and systematic.
- Avoid bulky, loose or trailing clothes. Avoid long loose hair.
- No one is allowed to enter in the lab area bare foot due to increased risk of electric shock.
- Remove metal bracelets, rings or watchstraps when working in the laboratories.
- Avoid working with wet hands and clothing.

2. Food, Beverages and Smoking

- Due to the increased risk of electric shock, no drinking of beverages, consumption or storage of any kind of food is allowed in the laboratory.
- Smoking is prohibited in all laboratories in all timings.

3. Soldering

- No one is allowed to do soldering in any of the computer engineering laboratories except the graduation project design laboratory.
- Anyone doing soldering in the graduation project design laboratory must wear appropriate apparel, socks, gloves, covered shoes and safety goggles to prevent the possibility of severe burns resulting from the splashing or dripping of hot liquefied solder into the face and eyes or on to the exposed skin on the chest, hands, legs, and feet.
- Students who are not so properly attired for these tasks will NOT be allowed to perform any type of soldering in the graduation project design laboratory.

4. Laboratory Operating Hours

- Students are never allowed to work alone in any lab area other than scheduled laboratory operating hours unless either a Lab T/A or Course Instructor is present inside that lab area.
- The laboratory operating hours for students are posted on the entrance doorway and on the notice board of computer engineering department.

5. Power Supply Related Safety

- Voltages above 50-VAC or 120-VDC are always dangerous.
- Extra precautions should be considered as voltage levels are increased.
- Never make any changes to circuits or mechanical layout without first isolating the circuit by switching off and removing connections to power supplies.

6. Laboratory Equipment

- Lab equipment may not be removed from the Computer Engineering lab areas without the permission of the Laboratory Supervisors.
- Laboratory bench equipment (except for some lab bench computers) must be turned off before closing down the lab area for the day.
- Never open (remove cover) of any equipment in the laboratories.
- Never "jump," disable, bypass or otherwise disengage any safety device or feature of any equipment in the laboratories.
- Laboratories shall be locked when unoccupied.

7. Waste Management Safety

- Know the correct handling, storage and disposal procedures for batteries, cell, capacitors, inductors and other high energy-storage devices.

8. Equipment Safety

- Before equipment is energized ensure, circuit connections and layout have been checked by a Teaching Assistant (TA) and all colleagues in your group has given their consent.
- Experiments left unattended should be isolated from the power supplies. If for a special reason, it must be left on, a barrier and a warning notice are required.
- Equipment found to be faulty in any way should be reported to the lab supervisor and taken out of service until inspected and declared safe.

9. Equipment Accessories

- Use extension cords only when necessary and only on a temporary basis.
- Request new outlets if your work requires equipment in an area without an outlet.
- Discard damaged cords, cords that become hot, or cords with exposed wiring.

B. Electrical and Fire Emergency Responses

1. Police, Fire or Medical Emergency

- Use the telephone located in the laboratory area and press 0-996 to notify police, fire, and ambulance for emergency help.
- Everyone present in the laboratory area shall be familiar with the locations and operation of safety and emergency equipment, including but not limited to, fire extinguishers, first aid kits, emergency power off system, fire alarm pull stations, and emergency telephones.

2. Electric Shock

- When someone suffers serious electrical shock, he may be knocked unconscious.
- If the victim is still in contact with electrical current, immediately turn off the electrical power source.
- If you cannot disconnect the power source, depress the Emergency Power Off switch.
- Do not touch a victim that is still in contact with a live power source; you could be electrocuted! Have someone call for emergency medical assistance immediately. Administer first-aid, as appropriate.

3. Electrical Fire

- If an electrical fire occurs, try to disconnect the electrical power source, if possible.
- If the fire is small and you are not in immediate danger; and if you have been properly trained in fighting fires, use the correct type of fire extinguisher to extinguish the fire.
- When in doubt, push in the Emergency Power Off button.
- NEVER use water to extinguish an electrical fire.

4. Emergency Power Off

- Every lab is equipped with an Emergency Power off System.
- When this switch is depressed, electrical power to the lab will shut off, except for lights.
- Only authorized personnel are permitted to reset power once the Emergency Power Off system has been engaged.

5. Building Evacuation in Emergency

- Everyone present in the laboratory should be familiar to emergency exits & way out plans.
- Use the nearest exit doorway from lab area closest to the stairwell to exit the building.
- Follow the Emergency Exit Signs posted in the hallways. Do not use elevators.
- Lab Teaching Assistants (T/As) or Instructor shall make sure all persons are out of the laboratory area and follow the directions posted at each doorway to the laboratory area.

The above general laboratory safety rules are designed to safeguard you and your co-workers, fellow students and colleagues and are a minimum requirement for individuals working in the computer engineering laboratories at Umm Al-Qura University, Makkah Al-Mukarramah. Specialized training and rules may apply depending on type and scope of activities involved.

This page is intentionally left blank

Lab 1: Introduction to Verilog HDL

In this lab we will learn about

- ✓ Basic building blocks of Hardware Description language
  - Introduction to Hardware Description Language
  - Basic Verilog HDL constructs with examples
  - Levels of abstractions in Verilog HDL
  - Comparisons of different abstraction level with examples

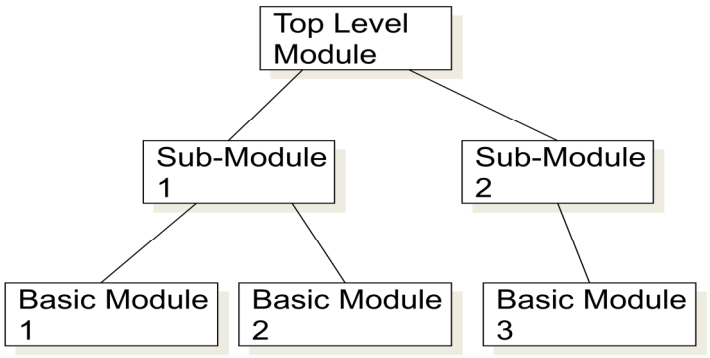
Hardware Description Language

HDL (Hardware Description Language) is a language that describes the hardware of digital systems in a textual form. It resembles a programming language, but is specifically oriented to describing hardware structures and behaviors. The main difference with the traditional programming languages is HDL's representation of extensive parallel operations whereas traditional ones represent mostly serial operations.

The most common use of a HDL is to provide an alternative to schematics. When a language is used for the above purpose (i.e. to provide an alternative to schematics), it is referred to as a structural description in which the language describes an interconnection of components. Such structural description can be used as input to logic simulation just as a schematic is used. Models for each of the primitive components are required. If an HDL is used, then these models can also be written in the HDL providing a more uniform, portable representation for simulation input. HDL can be used to represent logic diagrams, Boolean expressions, and other more complex digital circuits. There two levels of approaches for hardware designing are followed.

i. Top down approach
ii. Bottom up approach

In top down design, a very high-level description of an entire system can be precisely specified using an HDL. This high-level description can then be refined and partitioned into lower-level descriptions as a part of the design process. While in bottom up approach, design process starts from lowest possible description of hardware and goes upward in hierarchy toward to top level.



As a documentation language, HDL is used to represent and document digital systems in a form that can be read by both humans and computers and is suitable as an exchange language between designers. The language content can be stored and retrieved easily and processed by computer software in an efficient manner.

There are two applications of HDL processing:

i. Simulation
ii. Synthesis

A simulator interprets the HDL description and produces a readable output, such as a timing diagram, that predicts how the hardware will behave before it is actually fabricated. Simulation allows the detection of functional errors in a design without having to physically create the circuit. Logic synthesis is the process of converting a high-level description of design into an optimized gate-level representation. Logic synthesis uses a standard cell library which have simple cells, such as basic logic gates like and, or, and nor, or macro cells, such as adder, muxes, memory, and flip-flops. Standard cells put together are called technology library. Normally the technology library is known by the transistor size (0.18u, 90nm).

There are two standard HDL's that are supported by IEEE.

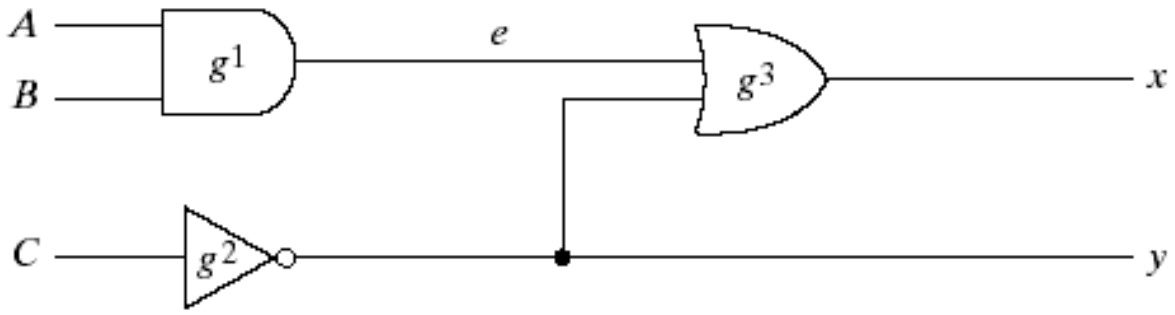
i. VHDL (Very-High-Speed Integrated Circuits Hardware Description Language) - Sometimes referred to as VHSIC HDL, this was developed from an initiative by US. Dept. of Defense.

ii. Verilog HDL – developed by Cadence Data systems and later transferred to a consortium called Open Verilog International (OVI).

Verilog is both a behavioral and a structural language. Internals of each module can be defined at four levels of abstraction, depending on the needs of the design. The levels are defined below.

1. Basic Verilog Constructs with example

Verilog HDL has a syntax that describes precisely the legal constructs that can be used in the language. It uses about 100 keywords pre-defined, lowercase, identifiers that define the language constructs. Example of keywords: *module, endmodule, input, output wire, and, or, not* , etc., Any text between two slashes (//) and the end of line is interpreted as a comment. Blank spaces are ignored and names are case sensitive. A module is the building block in Verilog. It is declared by the keyword *module* and is always terminated by the keyword *endmodule*. Each statement is terminated with a semicolon, but there is no semi-colon after *endmodule*. Example is shown in Fig 2.



//Simple example of verilog HDL design block without delay

module smpl_circuit(A,B,C,x,y);

input A,B,C;

```
output x,y;
wire        e;
and       g1(e,A,B);
not       g2(y,C);
or        g3(x,e,y);
endmodule
```

1.1 Gate Delays

Sometimes it is necessary to specify the amount of delay from the input to the output of gates. In Verilog, the delay is specified in terms of time units and the symbol #.The association of a time unit with physical time is made using timescale compiler directive. Compiler directive starts with the "backquote (`)" symbol. E.g.`timescale 1ns/100ps .The first number specifies the unit of measurement for time delays. The second number specifies the precision for which the delays are rounded off, in this case to 0.1ns.

```
// Description of circuit with delay
module circuit_with_delay(A,B,C,x,y);
input  A,B,C;
output x,y;
wire   e;
and #(30) g1(e,A,B);
or  #(20) g3(x,e,y);
not #(10) g2(y,C);
endmodule
```

1.2 Testbench or Stimulus

A stimulus module is an HDL program that has the following form.

module testname

Declare local reg and wire identifiers

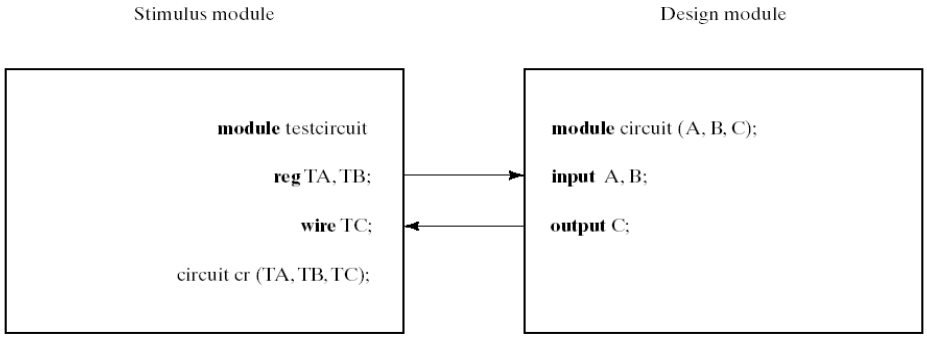
Instantiate the design module under test.

Generate stimulus using initial and always statements

Display the output response.

endmodule

A test module typically has no inputs or outputs. The signals that are applied as inputs to the design module for simulation are declared in the stimulus module as local **reg** data type. The outputs of the design module that are displayed for testing are declared in the stimulus model as local **wire** data type. The module under test is then instantiated using the local identifiers.



In order to simulate a circuit with HDL, it is necessary to apply inputs to the circuit for the simulator to generate an output response. An HDL description that provides the stimulus to a design is called a test bench. The *initial* statement specifies inputs between the keyword *begin* and *end*.

The always statement executes repeatedly in a loop. The initial statement executes only once starting from simulation time=0 and may continue with any operations that are delayed by a given number of units as specified by the symbol #. Initially ABC=000 (A,B and C are each set to 1'b0 (one binary digit with a value 0). $finish is a *system task*.

//Testbench for simple circuit

module stimcrct;

reg A,B,C;

```
wire x,y;
circuit_with_delay cwd(A,B,C,x,y);
initial
  begin
     A = 1'b0; B = 1'b0; C = 1'b0;
   #100
     A = 1'b1; B = 1'b1; C = 1'b1;
   #100  $finish;
  end
endmodule
```

The response to the stimulus generated by the initial and always blocks will appear at the output of the simulator as timing diagrams. It is also possible to display numerical outputs using Verilog *system tasks*.

$display – display one-time value of variables or strings with end-of-line return,

$write – same $display but without going to next line.

$monitor – display variables whenever a value changes during simulation run.

$time – displays simulation time

$finish – terminates the simulation

The syntax for $display,$write and $monitor is of the form

Task-name (format-specification, argument list);

E.g. $display(%d %b %b, C,A,B);

$display("time = %0d A = %b B=%b",$time,A,B)

2. Basic Concepts

2.1 Operators

Operators are of three types: unary, binary, and ternary. Unary operators precede the operand. Binary operators appear between two operands. Ternary operators have two separate operators that separate three operands. Examples

a = ~ b; // ~ is a unary operator. b is the operand

a = b && c; // && is a binary operator. b and c are operands

a = b ? c : d; // ?: is a ternary operator. b, c and d are operands

2.2 Number Specification

There are two types of number specification in Verilog: sized and unsized

Sized numbers are represented as <size> '<base format> <number>.

Examples

4'b1111 // This is a 4-bit binary number

12'habc // This is a 12-bit hexadecimal number

16'd255 // This is a 16-bit decimal number.

Numbers that are specified without a <base format> specification are decimal numbers by default. Numbers that are written without a <size> specification have a default number of bits that is simulator- and machine-specific (must be at least 32).

23456 // This is a 32-bit decimal number by default

'hc3 // This is a 32-bit hexadecimal number

'o21 // This is a 32-bit octal number

2.3 Data Types

Integer, real, and time register data types are supported in Verilog. An integer is a general purpose register data type used for manipulating quantities. Integers are declared by the keyword integer. Example

integer counter; // general purpose variable used as a counter.

2.3.1 Real

Real number constants and real register data types are declared with the keyword real. They can be specified in decimal notation (e.g., 3.14) or in scientific notation (e.g., 3e6, which is 3 x 106).

2.3.2 Time

Verilog simulation is done with respect to simulation time. A special time register data type is used in Verilog to store simulation time. A time variable is declared with the keyword time. Example

time save_sim_time; // Define a time variable save_sim_time

2.3.3 Nets

Nets represent connections between hardware elements. Just as in real circuits, nets have values continuously driven on them by the outputs of devices that they are connected to. Nets are declared primarily with the keyword wire. Nets are one-bit values by default unless they are declared explicitly as vectors. The terms wire and net are often used interchangeably. The default value of a net is z (except the trireg net, which defaults to x). Nets get the output value of their drivers. If a net has no driver, it gets the value z. Example

wire a; // Declare net a for the above circuit

wire b,c; // Declare two wires b,c for the above circuit

wire d = 1'b0; // Net d is fixed to logic value 0 at declaration.

Note that net is not a keyword but represents a class of data types such as wire, wand,wor, tri, triand, trior, trireg, etc. The wire declaration is used most frequently.

2.3.4 Registers

Registers represent data storage elements. Registers retain value until another value is placed onto them. Do not confuse the term registers in Verilog with hardware registers built from edge-triggered flip-flops in real circuits. In Verilog, the term register merely means a variable that can hold a value. Unlike a net, a register does not need a driver. Verilog registers do not need a clock as hardware registers do. Values of registers can be changed anytime in a simulation by assigning a new value to the register. Register data types are commonly declared by the keyword reg. The default value for a reg data type is x.

```
reg reset; // declare a variable reset that can hold its value
initial // this construct will be discussed later
begin
reset = 1'b1; //initialize reset to 1 to reset the digital circuit.
#100 reset = 1'b0; // after 100 time units reset is deasserted.
End
```

Registers can also be declared as signed variables. Such registers can be used for signed arithmetic. Nets or reg data types can be declared as vectors (multiple bit widths). If bit width is not specified, the default is scalar (1-bit). Examples are shown below.

```
wire a; // scalar net variable, default
wire [7:0] bus; // 8-bit bus
wire [31:0] busA,busB,busC; // 3 buses of 32-bit width.
reg clock; // scalar register, default
```

reg [0:40] virtual_addr; // Vector register, virtual address 41 bits wide

2.3.5 Parameter

Verilog allows constants to be defined in a module by the keyword parameter. Parameters cannot be used as variables. Parameter values for each module instance can be overridden individually at compile time. This allows the module instances to be customized. This aspect is discussed later. Parameter types and sizes can also be defined. Example

parameter port_id = 5; // Defines a constant port_id

2.4 Condition in hardware circuits

Verilog supports four different types of values

Value Level	Condition in Hardware Circuits
0	Logic zero, false condition
1	Logic one, true condition
x	Unknown logic value
z	High impedance, floating state

3. Levels of Abstractions

3. 1 Behavioral or algorithmic level modeling

This is the highest level of abstraction provided by Verilog HDL. A module can be implemented in terms of the desired design algorithm without concern for the hardware implementation details. Designing at this level is very similar to C programming.

- Behavioral modeling represents digital circuits at a functional and algorithmic level.
- It is used mostly to describe sequential circuits, but can also be used to describe combinational circuits.
- Behavioral descriptions use the keyword **always** followed by a list of procedural assignment statements.
- The target output of procedural assignment statements must be of the **reg** data type.
- A **reg** data type retains its value until a new value is assigned.
- The procedural assignment statements inside the **always** block are executed every time there is a change in any of the variable listed after the @ symbol. (Note that there is no ";" at the end of **always** statement). Example is shown below.

```
//Behavioral description of 2-to-1-line multiplexer
module mux 2X1_bh(A,B,Select,OUT);
input A,B,Select;
output OUT;
reg OUT;
always @(Select or A or B)
if (Select==1)
OUT=A;
else
 OUT=B;
endmodule
```

3.2 Dataflow level modeling

At this level, the module is designed by specifying the data flow. The designer is aware of how data flows between hardware registers and how the data is processed in the design.

- Dataflow modeling uses a number of operators that act on operands to produce desired results.

- Verilog HDL provides about 30 operator types.
- Dataflow modeling uses continuous assignments and the keyword **assign**.
- A continuous assignment is a statement that assigns a value to a net.
- The value assigned to the net is specified by an expression that uses operands and operators.
- Dataflow Modeling provides the means of describing combinational circuits by their function rather than by their gate structure.
- **Conditional operator (?:)**
- condition ? true-expression : false-expression;
- A 2-to-1 line multiplexer
- **assign** OUT = select ? A : B;

  Example is shown below

```
//Dataflow description of 2-to-1-line multiplexer
module mux 2X1_df(A,B,Select,OUT);
input A,B,Select;
output OUT;
assign OUT=select?A:B;
endmodule
```

3.3 Gate level modeling

The module is implemented in terms of logic gates and interconnections between these gates. Design at this level is similar to describing a design in terms of a gate-level logic diagram.

```
//Gate level description of 2-to-1-line multiplexer
module mux2X1_GL(A, B, Select, OUT);
input  A, B, Select;
output  OUT;
```

```
wire Select_n,a1,b1;
and g1(a1,A,Select_n);
not g2(Select_ n, Select);
and g3(b1,Select,B);
or  g4(OUT,a1,b1);
endmodule
```

3.4 Switch level modeling

This is the lowest level of abstraction provided by Verilog. A module can be implemented in terms of switches, storage nodes, and the interconnections between them.

3.5 Different level of abstraction using Examples.

```
// Module 4-to-1 multiplexer using gate level modeling.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);
ouput out;
input i0, i1, i2, i3,s1,s0;
// Internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3;
// Gate instantiations
// Create s1n and s0n signals.
not  n1(s1n, s1);
not n2(s0n, s0);
// 3-input and gates instantiated
and a1(y0, i0, s1n, s0n);
and a2(y1, i1, s1n, s0);
and a3 (y2, i2, s1, s0n);
```

```
and a4 (y3, i3, s1, s0);
// 4-input or gate instantiated
or o1 (out, y0, y1, y2, y3);
endmodule

// Module 4-to-1 multiplexer using data-flow modeling.
module mux4_to_1 (
output  out,
 input  i0, i1, i2, i3,
input  s1, s0);
//Logic equation for out
assign out =    (~s1 & ~s0 & i0)|
              (~s1 & s0 & i1) |
              (s1 & ~s0 & i2) |
              (s1 & s0 & i3) ;
endmodule

// Module 4-to-1 multiplexer using behavioral modeling.
module mux4_to_1 ( output  out,
input  i0, i1, i2, i3,
      input  s1, s0);
always @(s1 or s0 or i0 or i1 or i2 or i3)
begin
  case ({s1, s0})
  2'b00: out = i0;
  2'b01: out = i1;
  2'b10: out = i2;
  2'b11: out = i3;
  default: out = 1'bx;
```

```
  endcase
end
endmodule
```

Verilog allows the designer to mix and match all four levels of abstractions in a design. In the digital design community, the term register transfer level (RTL) is frequently used for a Verilog description that uses a combination of behavioral and dataflow constructs and is acceptable to logic synthesis tools.

Lab 2: Half Adder Implementation using (Gate Level Modeling)

In this lab we will learn about

- ✓ Basic building blocks of Verilog HDL
  - Implementation using Modelsim
  - Design block of hardware
  - Test bench or Stimulus for hardware design

Half adder design block

Half adder consist of two inputs A and B, applied to 'and' and 'xor' gate. The output of 'and' is carry while the output of 'xor' gate is sum.

```
//Design block for half adder using gate level modeling
module half_adder(C, S, A, B);
        input A, B;
        output C, S;

        and a1(C, A, B);
        xor x1(S, A, B);
endmodule
```

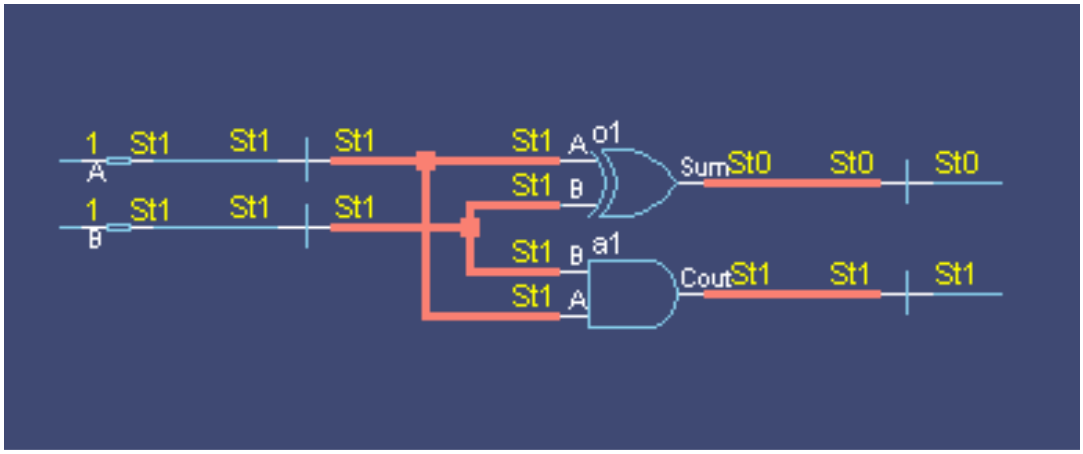
Once a design block is completed, it must be tested. The functionality of the design block can be tested by applying stimulus and checking results. Such a block is called the stimulus block. It is good practice to keep the stimulus and design blocks separate. The stimulus block is also commonly called a test bench. In test bench design must be instantiated to check the design.

Half adder Stimulus block or Test bench

```
// Test bench for half adder by instantiating half adder design block
module test_practical_02;
        reg A, B;
wire Cout, Sum;

        half_adder h1(Cout, Sum, A, B);

        initial
                begin
```

```
        {A, B} = 0;
        repeat(3)
                #1 {A, B} = {A, B} + 1;
        end

    initial $monitor("%b + %b = %b", A, B, {Cout, Sum});
endmodule
```

Dataflow diagram



Sample output

```
#           0     A = 0,     B = 0,     CS = 00
#           1     A = 0,     B = 1,     CS = 01
#           2     A = 1,     B = 0,     CS = 01
#           3     A = 1,     B = 1,     CS = 10
```

Tasks

1. Design full adder using Gate level modeling, also write a test bench.
2. Design full adder using two half adders and OR gate, also design a test bench.

Lab 3: 4-to-1 MUX Implementation using (Gate Level Modeling)

In this lab we will learn about

- ✓ Gate level modeling of multiplexer with Verilog HDL
  - Gate level modeling
  - Vectors in Verilog
  - Design block of hardware
  - Test bench or Stimulus for hardware design

4-to-1 multiplexer design block

Multiplexer have four inputs and two selection inputs while one output.

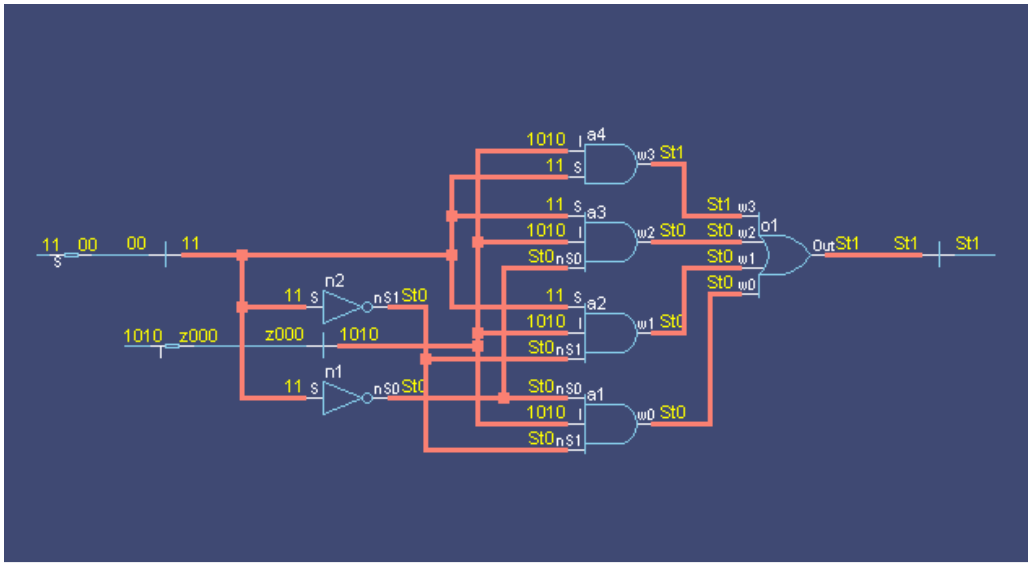
```
//Design block for 4-to-1 multiplexer using gate level modeling
module mux_4x1(Out, I, S);
        input [3:0]I;
        input [1:0]S;
        output Out;
        wire nS0, nS1;
        wire w0, w1, w2, w3;

        not n1(nS0, S[0]);
        not n2(nS1, S[1]);

        and a1(w0, I[0],  nS1,  nS0);
        and a2(w1, I[1],  nS1, S[0]);
        and a3(w2, I[2], S[1],   nS0);
        and a4(w3, I[3], S[1],  S[0]);

        or  o1(Out, w0, w1, w2, w3);
endmodule
```

Once a design block is completed, it must be tested. The functionality of the design block can be tested by applying stimulus and checking results. Such a block is called the stimulus block. It is good practice to keep the stimulus and design blocks separate. The stimulus block is also commonly called a test bench. In test bench design must be instantiated to check the design.

Test bench for 4-to-1 multiplexer

```
// Test bench for 4-to-1 multiplexer by instantiating 4x1 design block
module test_practical_03;
        reg [3:0]I;
        reg [1:0]S;
        wire Out;
        mux_4x1 m1(Out, I, S);

        initial
                begin
                I = 4'b1010;
                S = 2'b00;
                repeat(3)
                        #1      S = S + 1;
                end

        initial $monitor($time,"      S = %b      Out = %b      (I3 - I0 = %b)",S, Out, I);
endmodule
```

Dataflow diagram



Sample output of 4-to-1 multiplexer

```
#               0       S = 00          Out = 0         (I3 - I0 = 1010)
#               1       S = 01          Out = 1         (I3 - I0 = 1010)
#               2       S = 10          Out = 0         (I3 - I0 = 1010)
#               3       S = 11          Out = 1         (I3 - I0 = 1010)
```

Task

Design 2-to-1 multiplexer using Gate level modeling, also design a test bench.

Lab 4: 2-to-4 Decoder using (Gate Level Modeling)

In this lab we will learn about

- ✓ Decoder implementation using Verilog HDL
  - Gate level modeling
  - Design block of 2-to-4 Decoder
  - Test bench for 2-to-4 Decoder

2-to-4 Decoder design block

2-to-4 consist of two inputs and one output.

```
//Design block for 2-to-4 decoder using gate level modeling
module decoder_2x4(Out, I);
        input [1:0]I;
        output [3:0]Out;
        wire nI0, nI1;


        not n1(nI0, I[0]);
        not n2(nI1, I[1]);

        and a1(Out[0],  nI1,  nI0);
        and a2(Out[1],  nI1, I[0]);
        and a3(Out[2], I[1],   nI0);
        and a4(Out[3], I[1],  I[0]);
endmodule
```

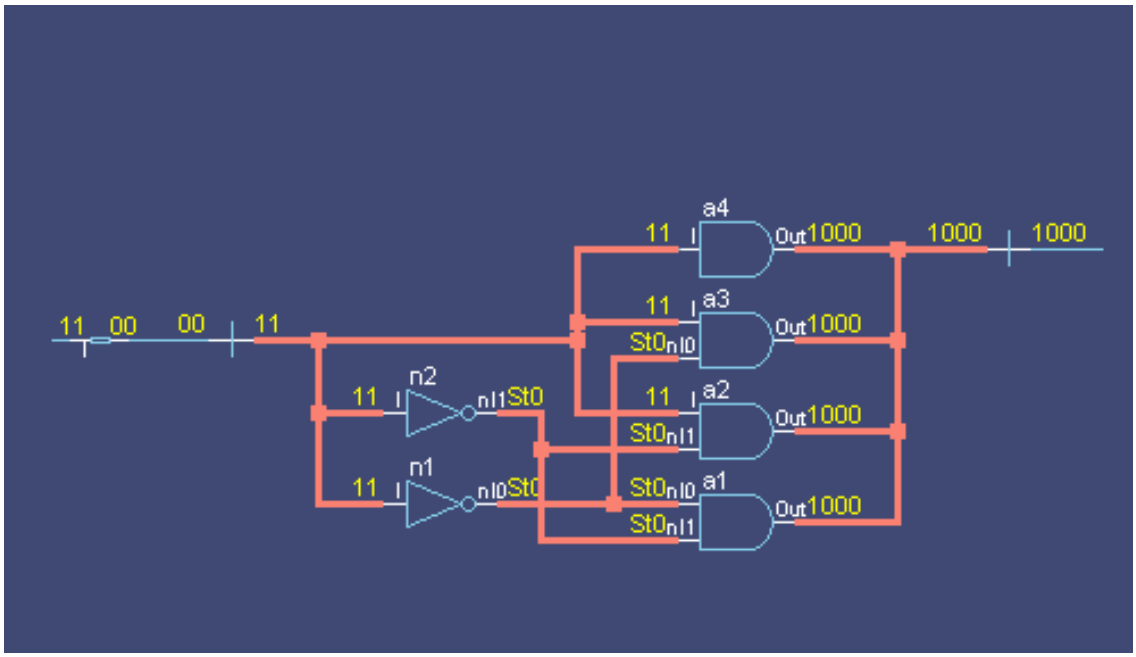
The stimulus block is also commonly called a test bench. In test bench design must be instantiated to check the design.

2-to-4 Decoder Test bench

```
// Test bench for 2-to-4 decoder by instantiating decoder design block
        module test_practical_04;
        reg [1:0]I;
```

```
        wire [3:0]Out;
        decoder_2x4 d1(Out, I);

        initial
                begin
                I = 2'b00;
                repeat(3)
                        #1      I = I + 1;
                end

        initial $monitor($time,"      I = %b      Out = %b",I, Out);
endmodule
```

Dataflow diagram



Sample output

```
#                    0        I = 00          Out = 0001

#                    1        I = 01          Out = 0010

#                    2        I = 10          Out = 0100

#                    3        I = 11          Out = 1000
```

Tasks

3. Design full adder using Gate level modeling, also write a test bench.
4. Design full adder using two half adders and OR gate, also design a test bench.

Lab 5: 4-bit Full Adder using (Gate Level Modeling)

In this lab we will learn about

- ✓ Gate level modeling of Full adder with Verilog HDL
  - Gate level modeling
  - Design block of Full adder
  - Test bench or Stimulus for Full adder design

4-bit adder using full adder design block

Half adder consist of two inputs A and B, applied to 'and' and 'xor' gate. The output of 'and' is carry while the output of 'xor' gate is sum.

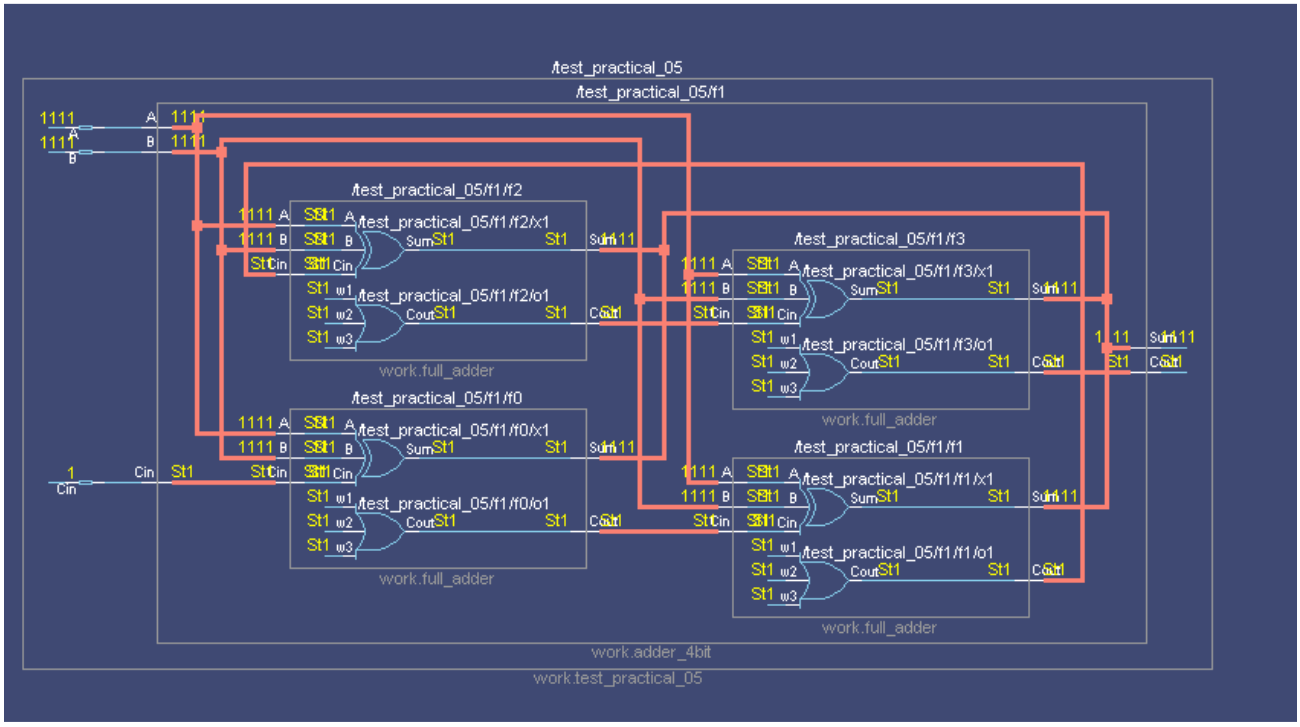
```
//Design block for 4-bit full adder using gate level modeling
module full_adder(Cout, Sum, A, B, Cin);
        input A, B, Cin;
        output Cout, Sum;

        xor x1(Sum, A, B, Cin);
        and a1(w1, A, B);
        and a2(w2, B, Cin);
        and a3(w3, Cin, A);
        or  o1(Cout, w1, w2, w3);
endmodule

module adder_4bit(Cout, Sum, A, B, Cin);
        input [3:0]A, B;
        input Cin;
        output [3:0]Sum;
        output Cout;
        wire w1, w2, w3;

        full_adder f0(w1, Sum[0], A[0], B[0], Cin);
        full_adder f1(w2, Sum[1], A[1], B[1], w1);
        full_adder f2(w3, Sum[2], A[2], B[2], w2);
        full_adder f3(Cout, Sum[3], A[3], B[3], w3);
endmodule
```

The stimulus block is also commonly called a test bench. In test bench design must be instantiated to check the design.

Test bench for 4-bit adder

```
// Test bench for 4-bit by instantiating 4-bit adder design block
module test_practical_06;
        reg [3:0]A, B;
        reg Cin;
        wire [3:0]Sum;
        wire Cout;
        adder_4bit f1(Cout, Sum, A, B, Cin);

        initial
                begin
                {A, B, Cin} = 0;
                #1              Cin = 0;
                #1          A = 10; B = 05;
                #1          A = 05; B = 07;
                #1          A = 00; B = 12; Cin = 1;
                #1      A = 11; B = 08;
                #1      A = 15; B = 15; Cin = 0;
                #1      A = 15; B = 15; Cin = 1;
                end

        initial $monitor($time,"        %d +\t %d +\t %d \t= \t%b-%d",A,B,Cin,Cout,Sum);
endmodule
```

Dataflow diagram



Sample output of 4-bit adder

```
#                 0          0 +    0 +  0     =      0- 0
#                 2         10 +    5 +  0     =      0-15
#                 3          5 +    7 +  0     =      0-12
#                 4          0 +   12 +  1     =      0-13
#                 5         11 +    8 +  1     =      1- 4
#                 6         15 +   15 +  0     =      1-14
#                 7         15 +   15 +  1     =      1-15
```

Task

Design 2-to-1 multiplexer using Gate level modeling, also design a test bench.

Mid Term Project

Design project 1. ***Implement 4x16 decoder using 4x1 decoders***

Lab 6: 4-to-1 MUX using (Dataflow Modeling)

In this lab we will learn about

- ✓ Data flow level modeling of multiplexer with Verilog HDL
  - Data flow level modeling
  - Use of assign statement and conditional operator
  - Design block of 4-to-1 mux and comparison with gate level modelling
  - Test bench or Stimulus for hardware design

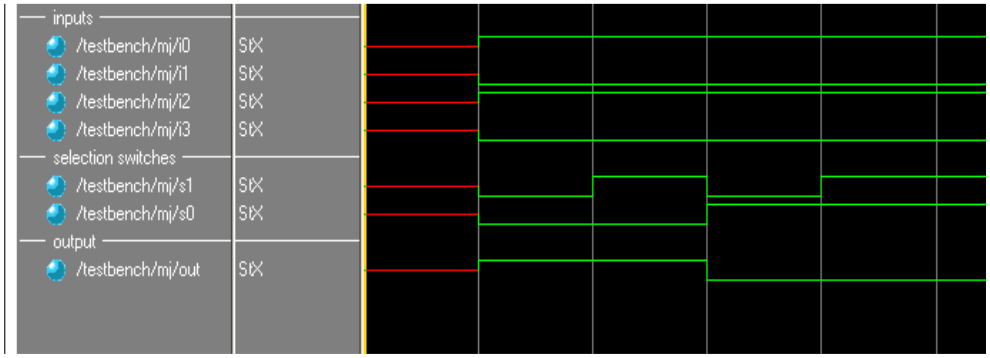
4-to-1 Mux design block

```
//Design block for 4-bit full adder using gate level modeling
Module MUX4(OUT,I0,I1,I2,I3,S1,S0);

Output out;

Input i0,i1,i2,i3,s0,s1;

Assign out=( s1 & s0 & i0)|
                    ( s1 & s0 & i1)|
                    ( s1 & s0 & i2)|
                    ( s1 & s0 & i3);
endmodule
```

4-to-1 Mux design block using Conditional operators

```
// Compare to gate-level model
module multiplexer4_to_1 (out, i0, i1, i2, i3, s1, s0);
// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3;
input s1, s0;
// Use nested conditional operator
assign out = s1 ? ( s0 ? i3 : i2) : (s0 ? i1 : i0) ;
endmodule
```

Dataflow diagram



Task

Design 2-to-1 multiplexer using Date flow level modeling, also design a test bench. Compare results with gate level implementation of 2-to-multiplexer

Lab 7: 32-bit Adder / Subtractor (Dataflow Modeling)

In this lab we will learn about

- ✓ Data flow level modeling of adder/subtractor with Verilog HDL
  - More complex designs using data flow modeling
  - Design block of 32-bit adder/subtractor
  - Test bench or Stimulus for hardware design

32-bit adder/subtractor

```
//Design block for 32-bit  adder/subtractor using data flow level modeling
        module adder_subtractor(Cout_Borrow, Sum_Difference, A, B, Cin, Command);
        input [31:0] A, B;
        input Cin, Command;
        output [31:0] Sum_Difference;
        output Cout_Borrow;

        assign {Cout_Borrow, Sum_Difference} = (Command)
                                                ? A + B + Cin
                                                : A - B - Cin;
endmodule
```

The stimulus block is also commonly called a test bench. In test bench design must be instantiated to check the design.

Test bench for 32-bit adder/subtractor

```
// Test bench for 32-bit adder/subtractor
module test_practical_09;
        reg [31:0]A, B;
        reg Cin, Command;
        wire [31:0]Sum_Difference;
        wire Cout_Borrow;
        adder_subtractor as1(Cout_Borrow, Sum_Difference, A, B, Cin, Command);

        initial
```

```
begin
{A, B, Cin, Command} = 0; //Command = 0 means subtraction
#1      A = 10; B = 2;
#1      A = 2;  B = 10;
#1      A = 52; B = 52;

#1      A = 0; B = 0; Command = 1;          // Command = 1 means
addition
#1      A = 10; B = 2;
#1      A = 2;  B = 10;
#1      A = 52; B = 52; Cin = 1;
end

      initial $monitor($time,"      %d  %c  %d = (%b) %d",A, (Command)?"+":"-", B,
Cout_Borrow, Sum_Difference);
endmodule
```

Sample output of 32-bit adder/subtractor

```
#      0      0      -      0      = (0) 0

#      1      10     -      2      = (0) 8

#      2      2      -      10     = (1) 4294967288

#      3      52     -      52     = (0) 0

#      4      0      +      0      = (0) 0

#      5      10     +      2      = (0) 12

#      6      2      +      10     = (0) 12

#      7      52     +      52     = (0) 105
```

Home Task

Write analysis report for the experiment performed in the lab.

Lab 8: 32-bit Magnitude Comparator using (Dataflow Modeling)

In this lab we will learn about

- ✓ Data flow level modeling of 32-bit magnitude Comparator with Verilog HDL
  - Data flow level modeling
  - Design block of hardware
  - Test bench or Stimulus for hardware design

32-bit magnitude comparator Design block

```
//Design block for 32-bit  comparator using data flow level modeling
module comparator_32bit(Eq, Gt, Lt, A, B);
        input [31:0] A, B;
        output Eq, Gt, Lt;

        assign   Eq = (A==B),
                 Lt = (A<B) ,
                 Gt = (A>B) ;
endmodule
```

The stimulus block is also commonly called a test bench. In test bench design must be instantiated to check the design.

Test bench for 32-bit magnitude comparator

```
// Test bench for 32-bit magnitude comparator
module test_practical_10;
        reg [31:0]A, B;
        wire Eq, Gt, Lt;
        comparator_32bit c1(Eq, Gt, Lt, A, B);

        initial
                begin
                {A, B} = 0;
                #1      A = 10; B = 2;
                #1      A = 2;  B = 10;
                #1      A = 52; B = 52;
```

```
        end

        initial $monitor($time,"    %d  %s  %d",A, (Eq)?"==":(Lt)?"<":">", B);
endmodule
```

Sample output of 32-bit magnitude comparator

```
#               0               0       ==      0
#               1               10      >       2
#               2               2       <       10
#               3               52      ==      52
```

Home Tasks

1. Write analysis report for the experiment performed in the lab.

2. Design BCD adder using data flow level modeling, also design a test bench.

Lab 9: Ripple Carry Counter Simulation (Behavioral Modeling)

In this lab we will learn about

- ✓ Behavioral modeling of D flip flop with Verilog HDL
  - Behavioral modeling
  - Sequential circuit design in verilog
  - Design block of hardware
  - Test bench or Stimulus for hardware design

Ripple carry counter top level design block

```
module ripple_carry_counter(q, clk, reset);
output [3:0] q;
input clk, reset;
//4 instances of the module T_FF are created.
T_FF tff0(q[0],clk, reset);
T_FF tff1(q[1],q[0], reset);
T_FF tff2(q[2],q[1], reset);
T_FF tff3(q[3],q[2], reset);
Endmodule
```

T-flip flop design block

```
module T_FF(q, clk, reset);
output q;
input clk, reset;
wire d;
D_FF dff0(q, d, clk, reset);
not n1(d, q); // not is a Verilog-provided primitive. case sensitive
endmodule
```

D-flip flop design block

```
//Design block for D flip flop using data behavioral level modeling
module en(clk,d,q,reset);

output q;
input d,clk,reset;

reg q;

always@(negedge clk or posedge reset)

if(reset)
q=0;
else
q=d;

endmodule
```

The stimulus block is also commonly called a test bench. In test bench design must be instantiated to check the design.

Test bench for Ripple carry counter

```
// Test bench for Ripple carry counter

module stimulus;

reg clk;

reg reset;

wire[3:0] q;

// instantiate the design block

ripple_carry_counter r1(q, clk, reset);

// Control the clk signal that drives the design block. Cycle time = 10

initial

clk = 1'b0; //set clk to 0

always

#5 clk = ~clk; //toggle clk every 5 time units
```

```
// Control the reset signal that drives the design block
// reset is asserted from 0 to 20 and from 200 to 220.
initial
begin
reset = 1'b1;
#15 reset = 1'b0;
#180 reset = 1'b1;
#10 reset = 1'b0;
#20 $finish; //terminate the simulation
end
// Monitor the outputs
initial
$monitor($time, " Output q = %d", q);
endmodule
```

Sample output for Ripple carry counter

```
0 Output q = 0
20 Output q = 1
30 Output q = 2
40 Output q = 3
50 Output q = 4
```

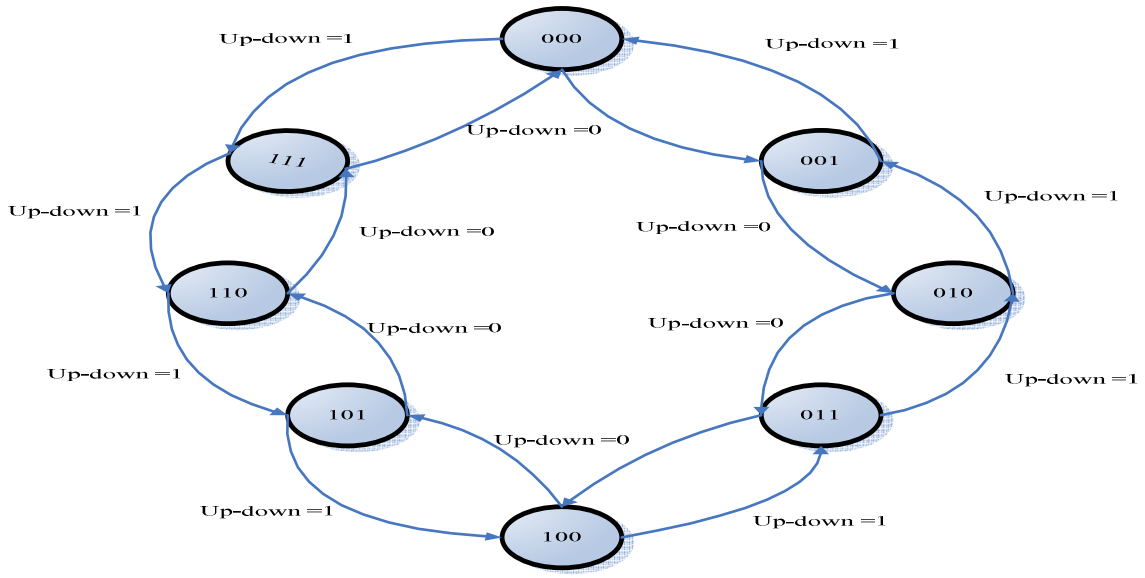
Lab 10: Up-down Circular Counter using Asynchronous Reset

In this lab we will learn about

- ✓ To understand 3-bit up-down circular counter with asynchronous reset
  - Sequential circuit design using Verilog HDL
  - Design block of hardware
  - Test bench or Stimulus for hardware design

Up-down circular counter and its design block

An up-down circular counter is one that counts up when the up-down bit is zero and count down when up-down bit is one and start from the beginning on getting to its maximum value. Asynchronous reset means that the counter is reset to zero whenever the reset bit is zero or one as desired irrespective of clock.




```
//Design block for D flip flop using data behavioral level modeling
module updown_counter (updown,count,clk,reset);
        input updown,clk,reset;
        output [2:0]count;
        reg [2:0]count ;
        always @ (posedge clk or negedge reset)
                if (~reset)
```

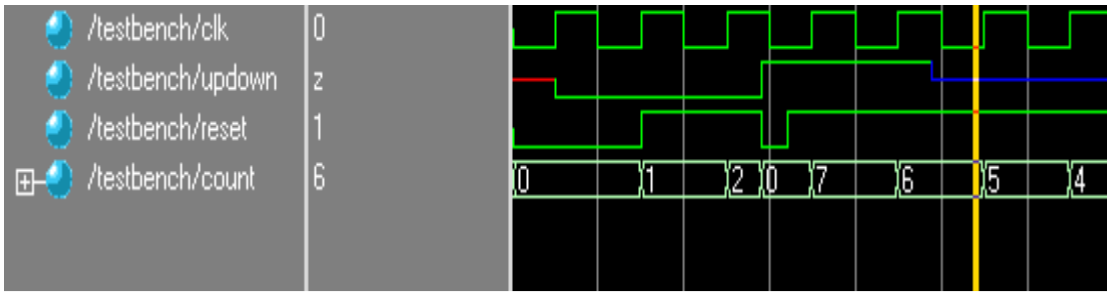
```
                                count= 3'd0;
else if(~updown)
                        count = count + 1'b1 %8;
                        else
                        count = count - 1'b1%8;
endmodule
```

The stimulus block is also commonly called a test bench. In test bench design must be instantiated to check the design.

Test bench for up-down circular counter

```
// Test bench for up-down circular counter
  module testbench;
            reg clk,updown,reset;
            wire [2:0] count;
            updown_counter test (updown,count,clk,reset);
            initial
            begin
                  clk=1'b0;
                        reset=1'b0;
      end
   always
   #5 clk=~clk;
             initial
            begin
                        updown = 1'bx;
                        #5 updown = 1'b0;
                        #10 reset = 1'b1;
                        #5 updown = 1'b0;
                  #9 updown = 1'b1; reset= 1'b0;
                        # 3 reset = 1'b1;
                        #7 updown = 1'b1;
                        # 10 updown = 1'bz;
                         #70$stop;
            end
endmodule
```

Dataflow diagram



Task

1. Write analysis report for the experiment performed in the lab. Explain the timing diagram

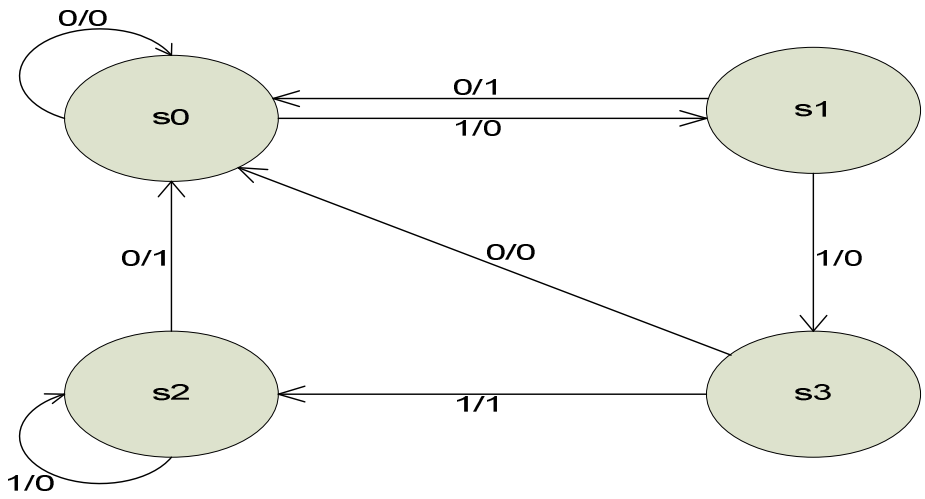
Lab 11: State Machine

In this lab we will learn about

- ✓ To understand state machine
  - State machine programming in Verilog HDL
  - Design block for state machine
  - Usage of case structures
  - Test bench or Stimulus for hardware design

State machine block diagram and design block

An up-down circular counter is one that counts up when the up-down bit is zero and count down when up-down bit is one and start from the beginning on getting to its maximum value. Asynchronous reset means that the counter is reset to zero whenever the reset bit is zero or one as desired irrespective of clock.




```
//Design block for state machine
module statemachine(y,prstate,x,clk,reset);

        input x,clk,reset;
        output y;
        output [1:0]prstate;
```

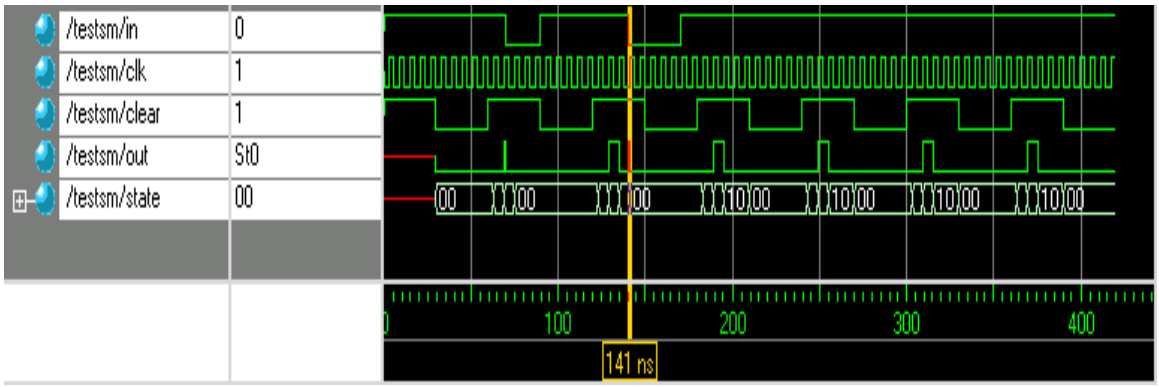
```
        reg y;

        reg [1:0] prstate,nxtstate;

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

        always@(posedge clk or negedge reset)
                if(~reset)
                        prstate=s0;
                else
                        prstate=nxtstate;
        always@(prstate or x)
                case(prstate)
                        s0:if(x) nxtstate=s1;
                          else nxtstate=s0;
                        s1:if(x) nxtstate=s3;
                          else nxtstate=s0;
                        s2:if(x) nxtstate=s2;
                          else nxtstate=s0;
                        s3:if(x) nxtstate=s2;
                          else nxtstate=s0;
                endcase
    always@(prstate or x)
                case(prstate)
                        s0:       y=0;
                        s1: if(x)
                                  y=1'b0;
                          else
                                  y=1'b1;
                        s2: if(x)
                                  y=1'b0;
                          else
                                  y=1'b1;
                        s3: if(x)
                                  y=1'b1;
                          else
                                  y=1'b0;
                endcase
endmodule
```

The stimulus block is also commonly called a test bench. In test bench design must be instantiated to check the design.

Test bench for state machine

```
// Test bench for state machine
module testsm;

reg in,clk,clear;
wire out;
wire [1:0]state;
        statemachine sm(out,state,in,clk,clear);

initial
        begin
        clk=1'b0;
        clear=1'b1;
        end
always
        #3 clk=~clk;

always
        #30 clear=~clear;

initial
        begin
          in=1'b1;
        #70 in=1'b0;
        #20 in=1'b1;
        #50 in=1'b0;
        #30 in=1'b1;
        #250 $stop;
        end

endmodule
```

Dataflow diagram



Task

1. Write analysis report for the experiment performed in the lab. Explain the timing diagram

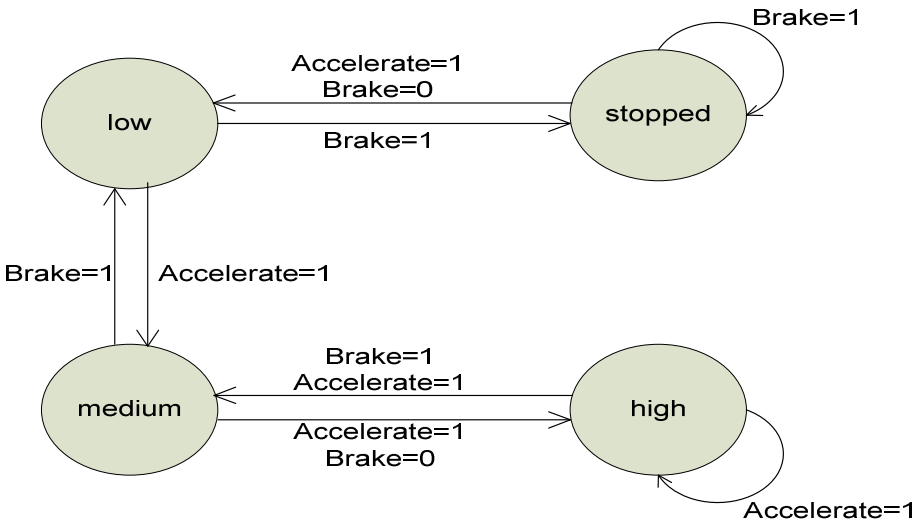
Lab 12: Speed Machine

In this lab we will learn about

- ✓ To understand state machine
  - State machine programming in Verilog HDL
  - Design block for state machine
  - Usage of case structures
  - Test bench or Stimulus for hardware design

Speed machine block diagram and design block

An up-down circular counter is one that counts up when the up-down bit is zero and count down when up-down bit is one and start from the beginning on getting to its maximum value. Asynchronous reset means that the counter is reset to zero whenever the reset bit is zero or one as desired irrespective of clock.




```
//Design block for speed machine
module speedmachine(speed,brake,accelerator,clk);
        input brake,accelerator,clk;
        output [1:0]speed;
        reg [1:0]speed;
        reg [1:0]state,nxtstate;
```

```
        parameter stopped=2'b00, s_low=2'b01,  s_medium=2'b10,  s_high=2'b11;
        always@(posedge clk)
                        begin
                        state=nxtstate;
                        speed=state;
                        end
        always@(state or accelerator or brake)
            if(brake)
            case(state)
                        stopped:nxtstate=stopped;
                         s_low: nxtstate=stopped;
                         s_medium:nxtstate=s_low;
                          s_high:nxtstate=s_medium;
                         default: nxtstate=stopped;
            endcase
            else if(accelerator)
                case(state)
                        stopped:nxtstate=s_low;
                         s_low: nxtstate=s_medium;
                         s_medium:nxtstate=s_high;
                          s_high:nxtstate=s_high;
                         default: nxtstate=stopped;
            endcase
endmodule

endmodule
```

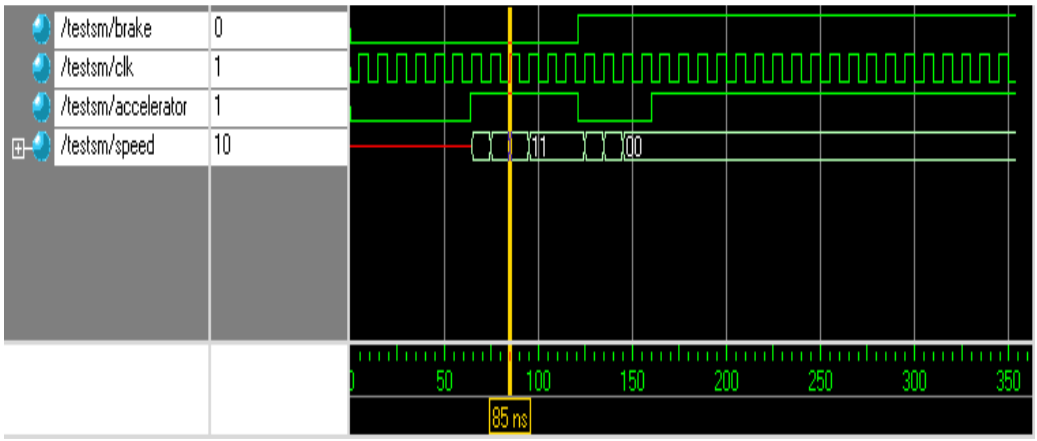
The stimulus block is also commonly called a test bench. In test bench design must be instantiated to check the design.

Test bench for speed machine

```
// Test bench for speed machine
module testsm;
reg brake,clk,accelerator;
wire [1:0]speed;
        speedmachine sm(speed,brake,accelerator,clk);
        initial
        clk=1'b0;
always
        #5 clk=~clk;
```

```
initial
        begin
         brake=1'b0;   accelerator=1'b0;
         #32 brake=1'b0;  accelerator=1'b0;
         #32 brake=1'b0;   accelerator=1'b1;
         #25 brake=1'b0;  accelerator=1'b1;
         #7 brake=1'b0;  accelerator=1'b1;
         #25 brake=1'b1;   accelerator=1'b0;
         #14 brake=1'b1;  accelerator=1'b0;
         #25 brake=1'b1;   accelerator=1'b1;
         #14 brake=1'b1;  accelerator=1'b1;
         #180 $stop;
        end
```

Dataflow diagram



Task

1. Write analysis report for the experiment performed in the lab. Explain the timing diagram