FPGA Programming using Verilog HDL Language

Conducted by:
BHUBANESWAR INSTITUTE OF TECHNOLOGY
Infovalley, Harapur, Bhubaneswar, Orissa - 752054, India
Ph - +91-674-2113498, Fax - +91-674-2113497, Email: info@bit.edu.in
Website: www.bit.edu.in

Course Objective

• Create and implement designs by using the ISE software design environment and Basys-2 Spartan3E FPGA board.
• Verilog code for synthesis
• Functions and Tasks
• Creating Finite State Machine (FSM) by using Verilog
• Verilog test fixtures for simulation
• Introduction to FPGA
• Project Work
PHASE-I  DAY 1 (3rd Aug, 2011)

AGENDA:

• Introduction to VLSI and its importance
• Getting Started with ISE 10.1 and Basys-2 Spartan 3E Kit
• Lab work

Introduction to VLSI and its importance

• VLSI stands for "Very Large Scale Integration".
• This is the field which involves packing more and more logic devices into smaller and smaller areas.
• VLSI circuits are everywhere ... your computer, your car, your brand new state-of-the-art digital camera, the cell-phones...
Introduction to VLSI and its importance

Moore’s Law:

• Gordon Moore: co-founder of Intel.
• Predicted that number of transistors per chip would grow exponentially (double every 18 months).

IC and number of logic gates:

• SSI: Small-scale Integration, Gates< 10
• MSI: Medium-scale Integration, 10< Gates< 1000
• LSI: Large-scale Integration, Gates> 1000
• VLSI: Very Large-scale Integration, Gates>100000
**Introduction to VLSI and its importance**

**Why VLSI?**

Integration improves the design:
- higher speed
- lower power
- physically smaller
- integration reduces manufacturing cost
- higher reliability
- more functionality

---

**Getting Started with ISE 10.1 and Basys-2**

**Spartan 3E Kit**

- Starting the ISE Software
- Creating a new project
  - Device Properties
  - Create an HDL source
- Writing of Program Code
- Checking the Syntax of the Module
- Design simulation
  - Initialize Timing
  - Simulating Design Functionality:
Getting Started with ISE 10.1 and Basys-2 Spartan 3E Kit

- Create timing constraints
- Implement design and verify constraints
  All Constraints Met Report
- Assigning Pin Location Constraints
  Reimplement design and verify pin locations
- Pin Report
- Download design to the sparta-3e basys-2 board
  To program a device
- Testing the FPGA board

LAB WORK

LAB WORK:

- Half Adder
- Full Adder
  Programming and implementation on FPGA
Verilog Programming for Half Adder and Full Adder:

<table>
<thead>
<tr>
<th>Half_Adder</th>
<th>Full_Adder</th>
</tr>
</thead>
<tbody>
<tr>
<td>module Half_Adder(</td>
<td>module Full_Adder(</td>
</tr>
<tr>
<td>input a,</td>
<td>input a,</td>
</tr>
<tr>
<td>input b,</td>
<td>input b,</td>
</tr>
<tr>
<td>output sum,</td>
<td>input cin,</td>
</tr>
<tr>
<td>output carry</td>
<td>output sum,</td>
</tr>
<tr>
<td>);</td>
<td>output cout</td>
</tr>
<tr>
<td>assign sum = a^b; // sum bit</td>
<td>assign sum = (a^b)^cin; // sum bit</td>
</tr>
<tr>
<td>assign carry = (a&amp;b); //carry bit</td>
<td>assign carry = (a&amp;b)</td>
</tr>
<tr>
<td>endmodule</td>
<td>(b&amp;cin)</td>
</tr>
<tr>
<td></td>
<td>(cin&amp;a); //carry bit</td>
</tr>
<tr>
<td></td>
<td>endmodule</td>
</tr>
</tbody>
</table>

LAB WORK


AGENDA:

- Needs of Verilog HDL
- Verilog Module and Ports
- One Language, Many Coding Styles
- Module instantiation
Needs of Verilog HDL

What is HDL?

• HDL 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.

• A primary use of HDLs is the simulation of designs before the designer commit to fabrication.

• Two major Hardware Description Languages (HDL) used by hardware designers in industry and academia are:
  - Verilog
  - VHDL

What is Verilog?

• Verilog HDL has a syntax that describes precisely the legal constructs that can be used in the language.

• Verilog was introduced in 1985 by Gateway Design System Corporation, now a part of Cadence Design Systems.
Verilog Module and Ports

Verilog Module:

• module is the basic building block in Verilog

Basic Elements of Module:

- Module Definition
  - module
  - endmodule

- Interface

- Add_on

- Module body

Example:

module Half_Adder(
  input a,
  input b,
  output sum,
  output carry
);
assign sum = a^b; // sum bit
assign carry = (a&b); // carry bit
endmodule
Verilog Module and Ports

The module name:
module names must follow these rules:
- It can be composed of letters, digits, dollar sign ($), and underscore characters (_) only.
- It must start with a letter or underscore.
- No spaces are allowed inside an identifier.
- Upper and lower case characters are distinguished (Verilog is case sensitive)
- Reserved keywords cannot be used as identifiers.

Ex: Counter_4Bit, ALU, Receiver, UART_Transmit

Ports:
Ports provide a means for a module to communicate through input and output.

3 types of ports in Verilog
- Input
- output
- inout
Internals of each module can be defined at four levels of abstraction, depending on the needs of the design.

four levels of abstractions are:
• Behavioral or algorithmic level
• Dataflow level
• Gate level or structural level
• Switch level

Behavioral or algorithmic level:

• 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.

Let’s design a 4:1 MUX in Behavioral Model
One Language, Many Coding Styles

4:1 MUX in Behavioral Model:

```verilog
module mux_4_to_1 (O, I0, I1, I2, I3, S1, S0);
output O;
input I0, I1, I2, I3, S0, S1;
reg O;
always @(S1 or S0 or I0 or I1 or I2 or I3)
begin
    case ({S1, S0})
        2'b00 : O = I0;
        2'b01 : O = I1;
        2'b10 : O = I2;
        2'b11 : O = I3;
        default : O = 1'bx;
    endcase
end
endmodule
```

Dataflow level:

- 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.

Let's design the same 4:1 MUX using Dataflow Model
One Language, Many Coding Styles

4:1 MUX using Dataflow Model:

module mux_4_to_1 (O, I0, I1, I2, I3, S1, S0);
output O;
input I0, I1, I2, I3, S0, S1;
assign O = (~S1 & ~S0 & I0) | (~S1 & S0 & I1) | (S1 & ~S0 & I2) | (S1 & S0 & I3);
endmodule

One Language, Many Coding Styles

Gate level or Structural level:

• The module is implemented in terms of logic gates and interconnections between these gates.

• It resembles a schematic drawing with components connected with signals.

• A change in the value of any input signal of a component activates the component. If two or more components are activated concurrently, they will perform their actions concurrently as well.

Let’s design a 4:1 MUX in Gate level Model
4:1 MUX in Gate level Model:

module mux_4_to_1 (O, I0, I1, I2, I3, S1, S0);
output O;
input I0, I1, I2, I3, S0, S1;
wire NS0, NS1;
wire Y0, Y1, Y2, Y3;
not N1(NS0, S0);
not N2(NS1, S1);
and A1(Y0, I0, NS1, NS0);
and A2(Y1, I1, NS1, S0);
and A3(Y2, I2, S1, NS0);
and A4(Y3, I3, S1, S0);
or O1(O, Y0, Y1, Y2, Y3);
endmodule

Switch level:

- This is the lowest level of abstraction provided by Verilog.
- A module can be implemented in terms of transistors, switches, storage nodes, and the interconnections between them.
- Design at this level requires knowledge of switch-level implementation details.
Module instantiation

- The process of creating objects from a module template is called \textit{instantiation}, and the objects are called \textit{instances}.

- A module can be instantiated in another module thus creating hierarchy.

**Syntax:**

\texttt{Module\_name \ Instance\_name (Port\_Association\_List)}

- Port\_association\_list shows how ports are mapped. Port mapping can be done in two different ways i.e. “Port mapping by order” and “Port mapping by name”.

**Example for Module Port Connection:**

**Verilog Programming for DFF**

```verilog
dmodule DFF (Q, D, CLK);
input D, CLK;
output Q;
reg Q;
always @ (posedge CLK)
Q <= D;
endmodule
```

**Diagram:**

- [Diagram of DFF modules with connections]
Module instantiation

Module port connection BY ORDER:

module SYNCHRO(ASYNC,SYNC,CLOCK);
input ASYNC;
input CLOCK;
output SYNC;
wire C1_ASYNC;
DFF DFF1 (C1_ASYNC, ASYNC, CLOCK);
DFF DFF2 (SYNC, C1_ASYNC, CLOCK);

Endmodule

• If the number of ports increased, then it is very difficult to do “module ports connection by order”.

Module instantiation

Module Port Connection BY NAME:

Module SYNCHRO (ASYNC, SYNC, CLOCK);
input ASYNC;
input CLOCK;
output SYNC;
wire C1_ASYNC;
DFF DFF1 (.D (ASYNC), .CLK (CLOCK), .Q (C1_ASYNC));
DFF DFF2 (.D (C1_ASYNC), .Q (SYNC), .CLK (CLOCK));
endmodule;

• Advantage of using “port connection by name” is, it is easy to port map for large number of ports in a design.

Note: Always connect ports by name.
Module instantiation

• Inside the same module, instance names of particular module should be
different. In the same way, inside the same module, instance name of
different modules should be different.

For example:

```plaintext
module A();
  DFF DFF1();
  DFFE DFF1();
endmodule
```

IS NOT ALLOWED

LAB WORK

LAB WORK:

• Implementation of Full Adder using the instantiation of Half
  Adder
  Note: design the Half Adder using dataflow method and implement it in Full
  Adder using both Module port connection by order and module port
  connection by name.

• Implementation of XOR gate using each programming models.
AGENDA:

- Comments in Verilog
- Data Types in Verilog
- Expressions, Operands And Operators in Verilog
- Timing control in Verilog

Comments in Verilog

- Comments can be inserted in the code for readability and documentation.

There are two ways to write comments.
- A one line comment starts with "//". Verilog skips from that point to the end of line.
- A multiple-line comment block comment starts with "/*" and ends with "*/".
- Multiple-line comments cannot be nested. However, one-line comments can be embedded in multiple-line comments.

```verilog
a = b && c;  // This is a one-line comment
/* This is a multiple line
comment */
/* This is /* an illegal */ comment */
/* This is //a legal comment */
```
Data Types in Verilog

Data values:

<table>
<thead>
<tr>
<th>Value Level</th>
<th>Condition in Hardware Circuits (Logic State)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Logic Zero/ False Condition/ Ground</td>
</tr>
<tr>
<td>1</td>
<td>Logic One/ True Condition/ Power</td>
</tr>
<tr>
<td>x</td>
<td>Unknown/ Uninitialized</td>
</tr>
<tr>
<td>z</td>
<td>High Impedance/ Floating</td>
</tr>
</tbody>
</table>

Values ‘x’ and ‘z’ are case insensitive.

Nets:

Keywords: wire, supply0, supply1
Default value: z
Default size: 1 bit

- Nets represent the continuous updating of outputs with respect to their changing inputs.
- Supply0 and supply1 define wires tied to logic 0 (ground) and logic 1 (power), respectively.

```verbatim
supply0 my_gnd; // equivalent to a wire assigned 0
supply1 a, b;
```
Data Types in Verilog

Registers:

Keywords: reg
Default value: x
Default size: 1 bit

• Registers represent data storage elements. Registers retain value until another value is placed onto them.

```verilog
reg a; // single 1-bit register variable
reg [7:0] b; // an 8-bit vector; a bank of 8 registers.
```

An example of how registers are used is shown.

```verilog
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
```
Data Types in Verilog

Comparing reg versus wire:

```verilog
module MUX21 (input A, input B, input SEL, output wire OUT1);
assign OUT1 = (A & SEL) | (B & ~SEL);
endmodule

OUT1 is a wire by default

module MUX21 (input A, input B, input SEL, output reg OUT1);
always @ (A, B, SEL)
    if (SEL)
        OUT1 = A;
    else
        OUT1 = B;
endmodule

OUT1 must be declared as reg
```

Vectors:

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).


**Vector Part Select:**

busA[7] // bit # 7 of vector busA

**Multi-dimensional arrays:**

reg [4:0] port_id[0:7]; // Array of 8 port_ids; each port_id is 5 bits wide
Data Types in Verilog

Memories:

• Memories are modeled in Verilog simply as an one-dimensional array of registers.

• Each element of the array is known as an element or word and is addressed by a single array index.

• It is important to differentiate between n 1-bit registers and one n-bit register.

```verilog
reg mem1bit[0:1023]; // Memory mem1bit with 1K 1-bit words
reg [7:0] membyte[0:1023]; // Memory membyte with 1K 8-bit words(bytes)
membyte[511] // Fetches 1 byte word whose address is 511.
```

Expressions, Operands And Operators

Expressions, operators, and operands form the basis of dataflow modeling.

Expressions:

Expressions are constructs that combine operators and operands to produce a result.

```verilog
// Examples of expressions. Combines operands and operators
a ^ b
in1 | in2
```
Expressions, Operands And Operators

Operands:

• Operands can be any one of the data types.

integer count, final_count;
final_count = count + 1; //count is an integer operand

Operators:

• Operators act on the operands to produce desired results.

<table>
<thead>
<tr>
<th>Operator Type</th>
<th>Operator Symbol</th>
<th>Operation Performed</th>
<th>Number of Operands</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>*</td>
<td>multiply</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>divide</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>add</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>subtract</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>modulus</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>**</td>
<td>power (exponent)</td>
<td>two</td>
</tr>
<tr>
<td>Logical</td>
<td>!</td>
<td>logical negation</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>logical and</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>logical or</td>
</tr>
<tr>
<td>Relational</td>
<td>&gt;</td>
<td>greater than</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>less than</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>less than or equal</td>
<td>two</td>
</tr>
<tr>
<td>Equality</td>
<td>==</td>
<td>equality</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>inequality</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>===</td>
<td>case equality</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td>!==</td>
<td>case inequality</td>
<td>two</td>
</tr>
</tbody>
</table>
### Expressions, Operands And Operators

#### Arithmetic Operators:
There are two types of arithmetic operators: binary and unary.

**Unary operators:**

-4 // Negative 4  
+5 // Positive 5

**Binary operators:**

D / E // Divide D by E. Truncates any fractional part.

- If any operand bit has a value x, then the result of the entire expression is x.  
- Modulus operators produce the *remainder* from the division of two numbers.

13 % 3 // Evaluates to 1

### Bitwise

<table>
<thead>
<tr>
<th>Bitwise</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>bitwise negation</td>
<td>one</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise and</td>
<td>two</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise xor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise xnor</td>
</tr>
</tbody>
</table>

### Reduction

<table>
<thead>
<tr>
<th>Reduction</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>reduction and</td>
<td>one</td>
</tr>
<tr>
<td>!&amp;</td>
<td>reduction nand</td>
<td>one</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reduction or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reduction xor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>reduction xor</td>
</tr>
</tbody>
</table>

### Shift

<table>
<thead>
<tr>
<th>Shift</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;&gt;</td>
<td>Right-shift</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Left-shift</td>
</tr>
<tr>
<td>&gt;&gt;&gt;</td>
<td>Arithmetic right-shift</td>
</tr>
<tr>
<td>&lt;&lt;&lt;</td>
<td>Arithmetic left-shift</td>
</tr>
</tbody>
</table>

### Concatenation

<table>
<thead>
<tr>
<th>CONCATENATION</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{ }</td>
<td>Concatenation</td>
</tr>
</tbody>
</table>

### Replication

<table>
<thead>
<tr>
<th>REPLICATION</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>{ { } }</td>
<td>Replication</td>
</tr>
</tbody>
</table>

### Conditional

<table>
<thead>
<tr>
<th>CONDITIONAL</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>Conditional</td>
</tr>
</tbody>
</table>

---

23-08-2011
Logical Operators:

- Logical operators are *logical-and* (&&), *logical-or* (||) and *logical-not* (!). Operators && and || are binary operators. Operator ! is a unary operator.
- Logical operators always evaluate to a 1-bit value, 0 (false), 1 (true), or x (ambiguous).

```
// Logical operations
A = 2'b10; B = 1'b0;
A && B // Evaluates to 0. Equivalent to (logical-1 && logical-0)
```

Relational Operators:

- Relational operators are *greater-than* (>), *less-than* (<), *greater-than-or-equal-to* (>=), and *less-than-or-equal-to* (<=).

```
// X = 4'b1010, Y = 4'b1101,
Y >= X // Evaluates to a logical 1
```
Expressions, Operands And Operators

Equality Operators:

- Equality operators are logical equality (==), logical inequality (!=), case equality (===), and case inequality (!==).

// X = 4'b1010, Y = 4'b1101
// Z = 4'b1xxz, M = 4'b1xxz
X != Y // Results in logical 1
X == Z // Results in x
Z === M // Results in logical 1 (all bits match, including x and z)

Expressions, Operands And Operators

Bitwise Operators:

- Bitwise operators are negation (~), and (&&), or (||), xor (^), xnor (^~), ^~.
- Bitwise operators perform a bit-by-bit operation on two operands.
- The exception is the unary negation operator (~), which takes only one operand and operates on the bits of the single operand.

// X = 4'b1010, Y = 4'b1101, Z = 4'b10x1
~X // Negation. Result is 4'b0101
X | Y // Bitwise or. Result is 4'b1111
Expressions, Operands And Operators

Bitwise Operators cont...:

• It is important to distinguish bitwise operators ~, &, and | from logical operators !, &&, ||.
• Logical operators always yield a logical value 0, 1, x, whereas bitwise operators yield a bit-by-bit value. Logical operators perform a logical operation, not a bit-by-bit operation.

```plaintext
// X = 4'b1010, Y = 4'b0000
X | Y // bitwise operation. Result is 4'b1010
X || Y // logical operation. Equivalent to 1 || 0. Result is 1.
```

Expressions, Operands And Operators

Reduction Operators:

• Reduction operators are and (&), nand (~&), or (|), nor (~|), xor (^), and xnor (~^, ^~).
• Reduction operators take only one operand. Reduction operators perform a bitwise operation on a single vector operand and yield a 1-bit result.

```plaintext
// X = 4'b1010
&X // Equivalent to 1 & 0 & 1 & 0. Results in 1'b0
^X // Equivalent to 1 ^ 0 ^ 1 ^ 0. Results in 1'b0
```
Expressions, Operands And Operators

Shift Operators:

• Shift operators are *right shift* (>>), *left shift* (<<).
• Shift operators shift a vector operand to the right or the left by a specified number of bits.
• When the bits are shifted, the vacant bit positions are filled with zeros.

    // X = 4'b1100
    Y = X >> 1;  // Y is 4'b0110. Shift right 1 bit. 0 filled in MSB position.
    Y = X << 2; // Y is 4'b0000. Shift left 2 bits.

Concatenation Operator:

• The *concatenation* operator ( {, } ) provides a mechanism to append multiple operands.

    // A = 1'b1, B = 2'b00, C = 2'b10, D = 3'b110
    Y = {B, C} // Result Y is 4'b0010
    Y = {A, B[0], C[1]} // Result Y is 3'b101
Expressions, Operands And Operators

Replication Operator:

• Repetitive concatenation of the same number can be expressed by using a replication constant.
• A replication constant specifies how many times to replicate the number inside the brackets ( {} ).

A = 1'b1; B = 2'b00; C = 2'b10;
Y = { 4{A} } // Result Y is 4'b1111
Y = { 4{A} , 2{B} , C } // Result Y is 8'b1111000010

Expressions, Operands And Operators

Conditional Operator:

The conditional operator (?) takes three operands.
Usage: condition_expr ? true_expr : false_expr ;

The condition expression (condition_expr) is first evaluated. If the result is true (logical 1), then the true_expr is evaluated. If the result is false (logical 0), then the false_expr is evaluated.

Y=x?a:b; // if x is ‘1’ then Y=a, if x is ‘0’ then Y=b
Delay in Assignment (#):

- It delays execution for a specific amount of time, ‘delay’.

**Syntax:** #delay

Two types of delay assignments:

- **Delayed assignment:**
  \[ \text{variable} = \text{expression}; \]

- **Intra-assignment delay:**
  \[ \text{variable} = \# \Delta t \text{ expression}; \]

---

Example:

```verilog
module delay_test(
    input a,  
    input b, 
    output reg c, 
    output reg d 
);
always begin 
    c = #20 (a&b); // a&b gets evaluated at time 0 but gets assigned to ‘c’ after 20ns 
    #50 d = (a|b); 
    /* a|b gets evaluated after 50ns and gets assigned to ‘d’ immediately.*/
end
endmodule
```
**Timing control in Verilog**

**Event Control (@):**
- It delays execution until there is a transition on any one of the signals in the sensitivity list.

Syntax: @(edge signal or signal)

```verilog
module delay_test1(
    input a,
    input b,
    output reg c);
always @(a or b)
    c = (a&b);
endmodule
/*Here the statement within the always block gets evaluated whenever there is a transition on 'a' or 'b'.*/
```

**Wait:**
- It delays execution until the condition evaluates as true.

Syntax: wait(condition)

Module delay_test2(
    input a,
    input b,
    output reg c);
initial
begin
    c=0;
    wait(a==1)
    c=b;
end
endmodule
LAB WORK

LAB WORK:
• Design a 2- to – 1 multiplexer using dataflow modeling (logic equation), and Using Conditional Operators.

Verilog program for different operators
• Delay Problem

---

LAB WORK

<table>
<thead>
<tr>
<th>2-to-1 Multiplexer, Using Logic Equations</th>
<th>2-to-1 Multiplexer, Using Conditional Operators</th>
</tr>
</thead>
</table>
| /* Module 2-to-1 multiplexer using data flow. Logic equation*/  
  module mux2_to_1 (out, i0, i1, s);  
  // Port declarations from the I/O diagram  
  output out;  
  input i0, i1;  
  input s;  
  //Logic equation for out  
  assign out = (~s & i0) | (s & i1);  
  endmodule | /* Module 2-to-1 multiplexer using data flow. Conditional operator.*/  
  module multiplexer2_to_1 (out, i0, i1, s);  
  // Port declarations from the I/O diagram  
  output out;  
  input i0, i1;  
  input s;  
  assign out = s ? i1 : i0;  
  endmodule |
LAB WORK

Verilog program for different operators:

Write down a Verilog program which can show the use of following operations
- logical-or (||)
- Relational operator greater-than (>)
- logical inequality (!=)
- case equality (===)
- Bitwise operator negation (~)
- Reduction operator xor (^)
- right shift ( >> )
- concatenation operator ( {, } )
- Replication Operator

Delay Problem:

Write down a Verilog program to implement both intra-assignment delay and delayed assignment. Also implement a wait statement.

Weekend Assignment

• Designing of a 4:1 mux instantiating 2:1 mux only.

• Designing a three bit parallel adder instantiating full adder within that. The full adder should be instantiated with the half adders.

• Design a 3 bit parallel adder in behavioral modeling.
AGENDA:

• Behavioral Modeling in Verilog
  - Basic Blocks
  - Procedural Assignments
  - Procedural Assignment Groups
  - Various Programming Statements used in Verilog

Types of assignments in Verilog:

Continuous assignments:

• Continuous assignments can only be made to nets. The operands can be of any data type. If one of the operands on the right hand side (RHS) of the assignment change, as the name suggests, the net on the left hand side (LHS) of the assignment is updated.

Procedural assignments:

• Procedural assignments are made to reg, integer, real or time, they need updating constantly to reflect any change in the value of the operands on the RHS.
Behavioral Modeling in Verilog

Procedural Basic Blocks:

• Two Types of Procedural Basic Blocks
  - Initial Block
  - Always Block

Initial Block:

Keywords: initial

• An initial block consists of a statement or a group of statements enclosed in begin... end which will be executed only once at simulation time 0.

• If there is more than one block they execute concurrently and independently.

• The initial block is normally used for initialization, monitoring, generating wave forms (e.g., clock pulses) and processes which are executed once in a simulation.
Behavioral Modeling in Verilog

Example:
module initial_test(
    output reg clock,
    output reg alpha
);

initial
clock = 1'b0; // variable initialization
initial
begin // multiple statements have to be grouped
    alpha = 0;
    #10 alpha = 1; // waveform generation
    #20 alpha = 0;
    #5 alpha = 1;
end;
endmodule

Behavioral Modeling in Verilog

Always Block:

Keywords: always

• An always block is similar to the initial block, but the statements inside an always block will repeated continuously, in a looping fashion, until stopped by $finish.

One way to simulate a clock pulse:

reg clock;
initial clock = 1'b0; // start the clock at 0
always #10 clock = ~clock; // toggle every 10 time units
initial #5000 $finish // end the simulation after 5000 time units
endmodule
Behavioral Modeling in Verilog

Procedural Assignments:

Two types of Procedural Assignments:

- Blocking Assignments
- Nonblocking Assignments

Blocking Assignments:

- Procedural (blocking) assignments (=) are done sequentially in the order the statements are written.
- A second assignment is not started until the preceding one is complete.

**Syntax**

variable = expression;

initial
begin
a=0; b=0; c=1;
#50 a = b | c; // wait for 50 units, and execute a = b | c =1.
d = a; // Time continues from last line, d=1 = b | c at t=50.
a = #50 b & c; // Time continues from last line, b & c = 0 at t = 50, a=0 = b & c at t=100.
d = a; // Time continues from last line, d=0 = b & c at t=100.
end
Behavioral Modeling in Verilog

Non Blocking Assignments:

- Nonblocking assignments (<=), which follow each other in the code, are started in parallel.

- The right hand side of nonblocking assignments is evaluated starting from the completion of the last blocking assignment or if none, the start of the procedure.

- The transfer to the left hand side is made according to the delays. An intra-assignment delay in a non-blocking statement will not delay the start of any subsequent statement blocking or non-blocking. However normal delays will are cumulative and will delay the output.

Syntax
variable <= expression;

Example:

initial
begin
a=0; b=0; c=1;
#50 a <= b|c; // b|c execute at t=0 then wait for 50 units, and execute a=1.
d <= a; // Time continues from last line, d=0 at t=50.
a <= #50 b&c;
/* b&c=0 execute at t=0. For assignment time continues from last line, a=0 = b&c at t=100. */
d <= a; // d=1 at t=50 for the delayed statement.
end
The following example shows interactions between blocking and non-blocking for simulation only (not for synthesis).

```
initial
begin
a=1; b=0; c=1; x=0;
#50 a = b & c; // wait for 50 units, then grab b,c and execute a = b&c = 0.
d = a; // Time continues from last line, d=0 = b&c at t=50.
x <= #60 b | c;
  // grab b|c now at t=50, don’t stop, make x=1 = b|c at t= 50+60 =110.
c <= #20 a; /* grab a at t=50 (end of last blocking statement).
    Deliver c=0 at t= 50+20 = 70. Previous x is unaffected by c change. */
y <= #10 b | c; // grab b|c at t=50, don’t stop, make y=1 at t=50+10=60.
#30 z = b ^~ c;
  // grab b~^c at t=80 (#50+#30), make z=1 (c =0 at t=70 and b = 0) at t=80.
w <= x; // make w=1 at t=80. Starting at last blocking assign.
end
```

For synthesis

- One must not mix “<=” or “=” in the same procedure.
- “<=” best mimics what physical flip-flops do; use it for “always @ (posedge clk ..) type procedures.
- “=” best corresponds to what c/c++ code would do; use it for combinational procedures.
Behavioral Modeling in Verilog

Procedural Assignment Groups:

- If a procedure block contains more than one statement, those statements must be enclosed within.

Sequential `begin - end` block
Parallel `fork - join` block

---

initial
begin
#1 clk = 0;
#5 reset = 0;
#5 enable = 0;
#2 data = 0;
end

clk gets 0 after 1 time unit, reset gets 0 after 6 time units, enable after 11 time units, data after 13 units. All the statements are executed sequentially.

initial
fork
#1 clk = 0;
#5 reset = 0;
#5 enable = 0;
#2 data = 0;
join

clk gets value after 1 time unit, reset after 5 time units, enable after 5 time units, data after 2 time units. All the statements are executed in parallel.
Behavioral Modeling in Verilog

Various Programming Statements used in Verilog:

If statement:
• It executes the statement or statement_group if the condition evaluates as true. If we need more than one statement (i.e. statement_group) then we need to use begin-end or a fork-join block.
Syntax
if(condition)
statement or statement_group

If-else statement:
• It executes the first statement or statement_group if the condition evaluates as true and executes the second statement or statement_group if the condition evaluates as false.
Syntax
if(condition)
statement or statement_group
else
statement or statement_group

Example:
if (alu_func == 2'b00)
aluout = a + b;
else if (alu_func == 2'b01)
aluout = a - b;
else if (alu_func == 2'b10)
aluout = a & b;
else // alu_func == 2'b11
aluout = a | b;

if (a == b)
begin
x = 1;
ot = 4'b1111;
end
**Behavioral Modeling in Verilog**

**Case statement:**

- It compares the *expression* with each of the *case_item*’s and executes the *statement* or *statement_group* associated with the first matching *case_item*.
- It executes the *default* if none of the *case_item*’s match. Here the default case is optional.

**Syntax:**

```
case(expression)
    case_item1: statement or statement_group
    case_item2: statement or statement_group
    ...
    case_itemN: statement or statement_group
    default: statement or statement_group
endcase
```

**Example:**

```
case (alu_ctr)
    2'b00: aluout = a + b;
    2'b01: aluout = a - b;
    2'b10: aluout = a & b;
    default: aluout = 1'bx; /* Treated as don’t cares for minimum logic generation. */
endcase

case (x, y, z)
    2'b00: aluout = a + b; /*case if x or y or z is 2'b00.*/
    2'b01: aluout = a - b;
    2'b10: aluout = a & b;
    default: aluout = a | b;
endcase
```
Behavioral Modeling in Verilog

Forever Loop:

• It is an infinite loop that continuously executes the statement or statement_group.

Syntax:
forever
statement or statement_group

Example:
always
forever begin
@(posedge clk);
a = a + 1;
end

Behavioral Modeling in Verilog

Repeat Loop:

• Like forever it is a loop that executes the statement or statement_group a fixed number of times based on the expression.

Syntax
repeat(expression)
statement or statement_group

Example:
always
repeat (2) begin
#50 a = 0;
#50 a = 1;
end
/* after 50, a = 0, after 100, a = 1, after 150, a = 0, after 200, a = 1 */
Behavioral Modeling in Verilog

While Loop:

• It executes the statement or statement_group as long as the condition evaluates as true.

Syntax
while(condition)
statement or statement_group

Example:
always
while (!a) begin
@ (posedge clk);
c = b | 1;
end

Behavioral Modeling in Verilog

For Loop:

• The for loop here uses three expressions separated by semicolons to control the loop.

Syntax
for(initial_value; condition; step)
statement or statement_group

• The first expression (initial_value) is executed once before entering the loop the first time.
• The second expression (condition) is evaluated to determine if the contents of the loop (i.e statement or statement_group) should be executed. If the loop condition expression is true, the loop is entered.
• The final expression (step) is evaluated at the end of the loop.
Behavioral Modeling in Verilog

Example:

```verilog
integer j;
always
for (j = 0; j <= 7; j = j + 1)
begin
    c[j] = a[j] & b[j];
    d[j] = a[j] | b[j];
end
```

LAB WORK:

• Generate a signal of 25MHz using initial and always block.
• Write a Verilog program to show the difference between begin-end and fork-join block.
• Write a verilog program with delays to show the difference between blocking and nonblocking assignment.
• Design a 4:1 MUX using if statement.
• Design a 3:8 Decoder using case statement.
• Generate a signal of 25MHz using forever loop.
• Design a 3:8 Decoder using while loop as well as for loop.
AGENDA:

- Function
- Task
- System Tasks
- Compiler Directives

Introduction to Functions and Tasks

Verilog provides Functions and Tasks to allow the behavioral description of a module to be broken into more manageable parts allowing better readability and manageability.

Functions and Tasks are useful for several reasons which are, they allow often used behavioral sequences to be written once and called when needed, they allow for a cleaner writing style and finally they allow data to be hidden from other parts of the design.
Functions

Functions are defined in the module in which they are used. It is possible to define function in separate file and use compile directive 'include to include the function in the file which instantiates the function. Function can call other functions, but cannot call task.

• Functions must contain at least one input argument and cannot drive more than one output.

• Functions cannot contain an inout or output declaration.

• Functions cannot contain time controlled statements (#, @, wait), i.e., it cannot contain delays.

Functions

• Function cannot include timing delays, like posedge, negedge, # delay. Which means that function should be executed in "zero" time delay.

• Functions must contain a statement that assigns the return value to the implicit function name register.

• The variables declared within the function are local to that function. The order of declaration within the function defines how the variables passed to the function by the caller are used.
Functions

Syntax of Function:

Function begins with keyword function and ends with keyword endfunction. Inputs are declared after the keyword function.

function [msb:lsb] function_name;
input [msb:lsb] input_arguments;
reg [msb:lsb] reg_variable_list;
parameter [msb:lsb] parameter_list;
integer [msb:lsb] integer_list;
... statements ...
endfunction

Example: Simple Function
function myfunction;
input a, b, c, d;
begin
myfunction = ((a | b) & (c | d));
end
endfunction

Calling a Function:
Let’s assume that function in above example is stored in a file called myfunction.v. Advantage of coding function in separate file is that, it can be used in multiple module's.
module func_test(a, b, c, d, e, f);
input a, b, c, d, e;
output f;
wire f;
`include "myfunction.v"
assign f = (myfunction (a,b,c,d)) ? e :0;
endmodule
Tasks

• Tasks are used in all programming languages, generally known as procedures or sub routines.

• A task is similar to a function, but unlike a function it has any number input and output ports. Therefore tasks do not return values.

• Included in the main body of code tasks can be called many times, reducing code repetition.

Tasks

• Tasks are defined in the module in which they are used. It is possible to define task in separate file and use compile directive 'include to include the task in the file which instantiates the task.

• Task can include timing delays, like posedge, negedge, # delay. The variables declared within the task are local to that task.

• The order of declaration within the task defines how the variables passed to the task by the caller are used. Task can call another task or function.
Tasks

Syntax of Task:

Task begins with keyword task and ends with keyword endtask. Input and output are declared after the keyword task. Local variables are declared after input and output declaration.

```vhd
task task_name;
input [msb:lsb] input_port_list;
output [msb:lsb] output_port_list;
reg [msb:lsb] reg_variable_list;
parameter [msb:lsb] parameter_list;
integer [msb:lsb] integer_list;
... statements ...
endtask
```

Example: Simple Task

```vhd
task convert;
input [3:0] temp_in;
output [4:0] temp_out;
begin
  temp_out = temp_in +5;
end
endtask
```
Tasks

Calling a Task:

Let’s assume that task in example “Simple Task” is stored in a file called mytask.v.
Advantage of coding task in separate file is that, it can be used in multiple modules.

module temp_cal (temp_a, temp_b, temp_c, temp_d);
  input [3:0] temp_a, temp_c;
  output [4:0] temp_b, temp_d;
  reg [4:0] temp_b, temp_d;
  `include "mytask.v"
  always @ (temp_a)
  convert (temp_a, temp_b);
  always @ (temp_c)
  convert (temp_c, temp_d);
endmodule

System Tasks

Verilog provides standard system tasks for certain routine operations. All system tasks appear in the form `$<keyword>`.

$display:

• Prints the formatted message once when the statement is executed during simulation.

Example:

initial $display("Hello World");

Prints the string between the quotation marks.
System Tasks

$strobe:

• It executes after all simulation events in the current time step have executed. Prints the formatted message once when executed. This task guarantees that the printed values for the signals/variables are the final values the signals/variables can have at that time step.

Example:
initial $strobe("Current values of A, B and C are A=%b, B=%b, C=%b", A, B, C);

Prints A, B and C and prints their value in binary format after all simulation events in the current time step have executed.

System Tasks

$monitor:

• Invokes a background process that continuously monitors the signals listed, and prints the formatted message whenever one of the signals changes. A newline is automatically added at the end of its output. Only one $monitor can be active at a time.

Example:
initial $monitor("Current values of A, B and C are A=%b, B=%b, C=%b", A, B, C);

Monitors A, B and C and prints their value in binary format whenever one of the signals (i.e. A or B or C) changes its value during simulation.
System Tasks

$time$

Returns the current simulation time as a 64-bit integer.

**Example:**

```verilog
initial $monitor("time = %d, A = %d", $time, A);
```

Prints the current simulation time as a 64-bit decimal whenever $monitor gets executed.


$finish$

Finishes a simulation and exits the simulation process.

**Example:**

```verilog
initial $finish;
```

Compiler Directives

• Compiler Directives direct the pre-processor part of Verilog Parser or Simulator. They are not bound by modules or files.

• When a Simulator encounters a compiler directive, the directive remains in effect until another compiler directive either modifies it or turns it off.

• All compiler directives are defined by using the `<keyword>` construct.
Compiler Directives

‘include:
File inclusion. The contents of another Verilog source file is inserted where the ‘include directive appears.

module func_test(a, b, c, d, e, f);
input a, b, c, d, e;
output f;
wire f;
#include "myfunction.v"
assign f = (myfunction(a,b,c,d)) ? e : 0;
endmodule

Here the contents of ‘myfunction.v’ are included within the ‘func_test’ module.

Compiler Directives

‘define:
Allows a text string to be defined as a macro name.

Example

‘define WIDTH 2
// Allows ‘WIDTH’ to be substituted by 2 where ever it gets used.
module adder (a,b,c);
input ['WIDTH-1:0] a;
input ['WIDTH-1:0] b;
output ['WIDTH:0] c;

assign c=a+b;
endmodule
Compiler Directives

Comparison between parameter and ‘define:

• ‘define is a macro that can be used to define any variable, function or expression under a name. You can use the macro for a given data in your code with 'identifier.

• Parameter is a data type in Verilog. It is used to declare constants which are not modified during runtime. Where as we can use defparam statement for updating the parameter.

• ‘define can be used anywhere within the module or outside the module.

• A parameter, on the other hand, will become a member of your module.

Example using parameter:

```verilog
module adder (a, b, c);
parameter WIDTH = 2; //default value
input [WIDTH-1:0] a;
input [WIDTH-1:0] b;
output [WIDTH:0] c;

assign c = a + b;
endmodule
```
Compiler Directives

`timestcale:

timescale <time_unit> / <time_precision>

• For specifying time unit and time precision of the modules that follow it.
• The time unit is the unit of measurement for time values such as the simulation time and delay values; the time precision specifies how simulation tools rounds time values.

For example:
	`timescale 1ns/100ps

test module
	assign #10.1678 out = in; // out is delayed by 10.2 ns from in.
endmodule

/* if `timescale 1ns/10ps; then out is delayed by 10.17 ns from in*/

LAB WORK

LAB WORK:

• Create a function which will take two 3 bit numbers and return the greater one. Use that function in the module to find out the greatest one among four 3 bit numbers.

• Create a Task which will be having 3 two bit inputs and 3 outputs among which one will hold the addition of 3 inputs, another will hold the greatest no. among them and the last one will hold the lowest no. among them. Use that task in a module which will be having 6 inputs and 6 outputs. First 3 outputs will hold the addition, greatest no. and lowest no. of first 3 inputs and last 3 outputs will hold the addition, greatest no. and lowest no. of last 3 inputs.
LAB WORK

Create a function which will take two 3 bit numbers and return the greater one. Use that function in the module to find out the greatest one among four 3 bit numbers.

```verilog
module func_t1 (a, b, c, d, e);

wire [2:0] f, g;

assign f = (myfunction(a, b));
assign g = (myfunction(c, d));
assign e = (myfunction(f, g));
endmodule
```

LAB WORK

Create a Task which will be having 3 two bit inputs and 3 outputs among which one will hold the addition of 3 inputs, another will hold the greatest no. among them and the last one will hold the lowest no. among them. Use that task in a module which will be having 6 inputs and 6 outputs. First 3 outputs will hold the addition, greatest no. and lowest no. of first 3 inputs and last 3 outputs will hold the addition, greatest no. and lowest no. of last 3 inputs.

```verilog
//Task Body “mytask.v”
task mytask;
input [1:0] a,b,c;
output [3:0] x;
output [1:0] y, z;
begin

// logic for addition
x = (a+b)+c;

// Rest of the task body refer next slide
end
endtask
```
LAB WORK

//logic for greast no.
if (a>=b && a>=c)
y = a;
else if (b>=c && b>=a)
y = b;
else if (c>=a && c>=b)
y = c;

//logic for lowest no.
if (a<=b && a<=c)
y = a;
else if (b<=c && b<=a)
y = b;
else if (c<=a && c<=b)
y = c;
end
endtask

LAB WORK

//Program calling task “mytask.v”
module task_t ( a,b,c,d,e,f,g,h,i,j,k,l);
input [1:0] a,b,c,d,e,f;
output reg [1:0] h,i,k,l;
output reg [3:0] g,j;

`include “mytask.v”

always @(a,b,c)
mytask (a,b,c,g,h,i);

always @(d,e,f)
mytask (d,e,f,j,k,l);
endmodule
AGENDA:

- Writing Testbenches
- Creating Testbenches and simulating for some simple designs

Writing Testbenches

- A test bench is a HDL program used for applying stimulus to a HDL design in order to test it and observe its response during simulation.

- In addition to the always statement, test benches use the initial statement to provide a stimulus to the circuit under test.
Writing Testbenches

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

```haskell
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.

The stimulus model generates inputs for the design module by declaring identifiers TA and TB as reg data type, and checks the output of the design unit with the wire identifier TC. The local identifiers are then used to instantiate the design module under test.
Writing Testbenches

• 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);
  $display(“time = %d A = %b B=%b”,$time,A,B);
  $finish;

Example 1: Description for the design

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

Example 1: Description for the testbench

//Stimulus for simple circuit
module stimcrct;
reg A,B,C;
wire x,y,z;
circuit_with_delay cwd(A,B,C,x,y,z);
initial
begin
  A = 1'b0; B = 1'b0; C = 1'b0;
  #100 A = 1'b1; B = 1'b0; C = 1'b1;
  #100 B = 1'b1;
  #100 $finish;
end
endmodule
Example 2: Description for the design
//Dataflow description of 2-to-1-line multiplexer
module mux21_df (A,B,select,OUT);
    input A,B,select;
    output OUT;
    assign OUT = select ? A : B;
endmodule

Example 2: Description for the testbench
//Stimulus for mux2x1_df
module testmux;
    reg TA,TB,TS; //inputs for mux
    wire Y; //output from mux
mux2x1_df mx (TA,TB,TS,Y); // instantiate mux
initial begin
    $display("MUX TEST");
    $monitor("select=%b A=%b B=%b OUT=%b T=%d",TS,TA,TB,Y,$time);
    TS = 1; TA = 0; TB = 1;
    #20 TA = 1;
    #20 TS = 0;
    #10 TB = 0;
end
endmodule
LAB WORK

LAB WORK:

• Create a testbench for the greatest number design using function you have created in the last class. Check all the possible stimulus for the design within the testbench. Use $monitor, $strobe, $time, $display, $finish; the system tasks within the testbench to check the outputs.

• Create a module having three inputs i0, i1, i2 each of 4 bits and having three outputs y0, y1, y2 each of 4 bits. Design the module in such a way that output y0, y1 and y2 will hold the value of inputs i0, i1, i2 in such a way that y2>y1>y0. Create a testbench for the above design and give all possible stimuli to the testbench to test the design.

AGENDA:

• Combinational Logic Implementation
• Creating Testbenches and simulating for the Combinational logic designed
Combinational Logic Implementation

- Modeling of 1x4 Demultiplexer using Verilog
- 3 bit parallel Adder/Subtractor using instantiation method
- Magnitude Comparator
- Modeling of Even Parity Generator using For Loop and using Unary operator
- Multi-Functional Combinational ALU
  - addition
  - subtraction
  - bitwise AND
  - bitwise XOR
  - magnitude comparison operation
  - parity generation operation

Modeling of 1x4 Demultiplexer using Verilog:

```verilog
module De_Mux_14 (y, i, s);
  input i;
  input [1:0] s;
  output [3:0] y;
  reg [3:0] y;
  always @(s, i)
  case (s)
    2'b00: y = {3'b000,i};
    2'b01: y = {2'b00,i,1'b0};
    2'b10: y = {1'b0,i,2'b00};
    2'b11: y = {i,3'b000};
    default: y = 4'b0000;
  endcase
endmodule
```

1x4 DeMUX

i

s[1:0]

y[3:0]
Combinational Logic Implementation

Magnitude Comparator

module mag_comp (aeb, alb, agb, a,b);
  output aeb, alb, agb;
  input [3:0] a,b;
  reg aeb, alb, agb;
always @(a,b)
    if (a==b) begin
      aeb = 1; alb = 0; agb = 0;
    end
    else if (a<b) begin
      aeb = 0; alb = 1; agb = 0;
    end
    else begin
      aeb = 0; alb = 0; agb = 0;
    end
endmodule

Combinational Logic Implementation

Modeling of Even Parity Generator using For Loop and using Unary operator

Using For loop:

module parity_gen_even (din, p_out);
parameter SIZE=4;
input [SIZE-1:0] din;
output p_out;
reg p_out;

reg temp;
integer i;
always @(din)
begin
    temp = 0;
    for (i=0; i<SIZE; i=i+1)
        temp = temp ^ din[i];

    p_out = temp;
end
endmodule

Combinational Logic Implementation

Using Unary Operator:

module parity_gen_even (din, p_out);
parameter SIZE=8;
input [SIZE-1:0] din;
output p_out;
wire p_out;

assign p_out = ^ din;

endmodule
Combinational Logic Implementation

Multi-Functional Combinational ALU
addition, subtraction, bitwise AND, bitwise XOR, magnitude comparison operation, parity generation operation

Problem Statement:
Sel = 00 : Addition
    = 01 : Subtraction
    = 10 : Bitwise AND
    = 11 : Bitwise XOR

aqb : a=b, alb : a<b, agb : a>b

oe : output enable signal

ov : holds the overflow at the time of arithmetic operation

p : generates the even parity bit for the output
Combinational Logic Implementation

Program:

module alu_mulfun(
    input [1:0] a,
    input [1:0] b,
    input [1:0] sel,
    input oe,
    output reg ov,
    output p,
    output [1:0] y,
    output reg aeb,
    output reg alb,
    output reg agb
);

Combinational Logic Implementation

reg [2:0] res;
always @(a or b or sel)
begin
    ov = 1'b0; res = 0;
    case (sel)
    2'b00:begin
        res = a+b;
        if(res>3'b011) ov = 1'b1;
    end
    2'b01:begin
        res = a-b;
        if(res>3'b011) ov = 1'b1;
    end
    2'b10:res = a&b;
    2'b11:res = a^b;
    default:res = 2'b00;
endcase
end
Combinational Logic Implementation

always @(a or b)
begin: Compare
if (a==b) {aeb, alb, agb} = 3'b100;
else if (a<b) {aeb, alb, agb} = 3'b010;
else {aeb, alb, agb} = 3'b001;
end

assign p = ^res;
assign y = oe?res:2'b11;
endmodule

AGENDA:

• Sequential Logic Implementation
• Creating Testbenches and simulating for the Sequential logic designed
Sequential Logic Implementation

Examples:

- ‘D’ Latch
- ‘D’ Latch with Reset
- ‘D’ Flip Flop
- ‘D’ Flip Flop with active high Asynchronous reset
- ‘D’ Flip Flop with active high Synchronous reset

---

‘D’ Latch:

```verilog
module Dlatch(q, latch_enb, data);
output q;
input data, latch_enb;
reg q;
always @(latch_enb or data)
begin
  if(latch_enb == 1)
    q = data;
end
endmodule
```

// latch_enb is a level sensitive signal. Since latch so level sensitive.
Sequential Logic Implementation

‘D’ Latch with reset:

module Dlatch(q, reset, latch_enb, data);
output q;
input data, latch_enb, reset;
reg q;
always @(reset or latch_enb or data)
begin
  if (reset == 1)
    q = 0;
  else if (latch_enb == 1)
    q = data;
end
endmodule

Sequential Logic Implementation

‘D’ Flip Flop:

module Dff(q, data, clk);
output q;
input data, clk;
reg q;
always @(posedge clk)
  q <= data;
endmodule

// clk is edge sensitive signal. Since Flipflop so edge sensitive.
Sequential Logic Implementation

- ‘D’ Flip Flop with active high Asynchronous reset:

```verilog
module Dff(q, data, rst, clk);
output q;
input data, rst, clk;
reg q;
always @(posedge clk or posedge rst)
begin
  if(rst == 1'b1)
    q <= 1'b0;
  else
    q <= data;
endmodule
// In case of asynchronous rst, it dominates over clk.
```

- ‘D’ Flip Flop with active high Synchronous reset:

```verilog
module Dff(q, data, rst, clk);
output q;
input data, rst, clk;
reg q;
always @(posedge clk)
begin
  if(rst == 1'b1)
    q <= 1'b0;
  else
    q <= data;
endmodule
// In case of synchronous rst, it synchronises with clk.
```
Sequential Logic Implementation

- SR Latch using both behavioral and instantiation method
- Modeling of 4bit Synchronous Up Counter with Reset
- Modeling of 4bit Asynchronous Up Down Counter with reset and load
- Shift Register – Scenario:1
- Shift Register – Scenario:2

Modeling of 4bit Synchronous Up Counter with Reset:

```verilog
module up_count_4b(q, rst, clk);
input clk, rst;
output [3:0] q;
reg [3:0] q;

always @(posedge clk)
begin
    if(rst == 0)
        q<=0;
    else
        q<=q+1;
end
endmodule
```
Sequential Logic Implementation

Modeling of 4bit Asynchronous Up Down Counter with reset and load:

Needs of the Design:

- \( d[3:0] \) = data in
- \( \text{rst} \) = clears the counter
- \( \text{load} \) = loads the \( 'd' \) value to \( 'q' \) till it will become \( '0' \)
- \( \text{ce} \) = enables counting
- \( \text{flag} \) = decides up counting or down counting
- \( \text{clk} \) = counting should happen at the positive edge of the clock

Decreasing Priority order of the signals:

```
module up_down_count_4b(q, d, load, ce, flag, rst, clk);
input [3:0] d;
input rst, load, ce, flag, clk;
output [3:0] q;
reg [3:0]q;

always @(posedge clk or posedge rst or posedge load)
begin
  if(rst == 1'b1)
    q<=0;
```

else
begin
  if(load == 1'b1)
    q<=d;
  else if (ce == 1'b1)
    begin
      if(flag == 1'b1)
        q<=q+1;
      else
        q<=q-1;
    end
  end
end
endmodule

**Shift Register – Scenario:1**

- **d[3:0]** = data input provides the input data which is required to be shifted
- **ld** = load the data input into the shift register
- **rst** = reset the shift register irrespective of the other inputs
- **clk** = shifting should happen at positive edge of clock signal
- **l_r** = direction flag, which indicates whether bits should be shifted in left/right direction
- **s_in** = fill the bit in empty place. This is done irrespective of the direction of shifting
Sequential Logic Implementation

module shift_reg (d, clk, ld, rst, l_r, s_in, q);
  output [3:0] q;
  input [3:0] d;
  input clk, ld, rst, l_r, s_in;
  reg [3:0] q;
always @(posedge clk)
begin
  if (rst)
    q <= 4’b0000;
  else if (ld)
    q <= d;
  else if (l_r)
    q <= {q[2:0], s_in};
  else
    q <= {s_in, q[3:1]};
end
endmodule

Sequential Logic Implementation

Shift Register – Scenario: 2

d[3:0] = data input provides the input data which is required to be shifted
rst = reset the shift register irrespective of the other inputs
clk = shifting should happen at positive edge of clock signal

If s1 and s0 are 0 and 1 respectively, then bits are shifted to right and vacated bits are filled with r_in and when s1 and s0 are 1 and 0 respectively, the bits are shifted to left and empty places are filled with l_in. When s1 and s0 are 1 and 1 respectively, then all the bits of d is loaded to corresponding bits of q and when s1 and s0 are 0 and 0, then q will be same as the previous value.
module uni_shift_reg (d, clk, rst, l_in, r_in, s1, s0, q);
    output [3:0] q;
    input [3:0] d;
    input clk, rst, l_in, r_in, s1, s0;
    reg [3:0] q;
always @(posedge clk)
begin
    if (rst)
        q <= 4'0000;
    else
        case ({s1, s0})
            2'b01: // shift_right
                q <= {r_in, q[3:1]};
            2'b10: // shift_left
                q <= {q[2:0], l_in};
            2'b11: // Parallel load
                q <= d;
            default: // Do nothing
                q <= q;
        endcase
    end
endmodule
Weekend Assignment

- Revision of combinational and sequential designs.
- Concept study for memory and State Machine.

PHASE-II  
DAY 11 & 12 (17\textsuperscript{th} & 18\textsuperscript{th} Aug, 2011)

AGENDA:

- Design of Finite State Machines
  - Moore machine Sequence Detector
  - Mealy machine Sequence detector
- Creating Testbenches and simulating for the FSM logic designed
Finite State Machines

Introduction to FSM:
A Finite State Machine (FSM) is a type of sequential circuit that is designed to sequence through specific patterns of finite states in a predetermined sequential manner.

An FSM consists of three parts
- State Memory
- Input Combinational logic
- Output Combinational Output Logic

State Memory:
The state memory is a set of n-bit flip-flops (state vector flip-flops) clocked by a single clock signal is used to hold the state vector (current state or simply state) of the FSM. A state vector with a length of n-bit has $2^n$ possible binary patterns known as state encoding. Often, not all $2^n$ patterns are needed, so the unused ones should be designed not to occur during normal operation.

Finite State Machines

Input Combinational Logic:
An FSM can only be in one state at any given time, and each active transition of the clock causes it to change from its current state to the next state, as defined by the next state logic. The next state is a function of the FSM’s input and its current state.

Combinational Output Logic:
Outputs are normally a function of the current state and possibly the FSM’s primary inputs (in the case of a Mealy FSM). Often in a Moore FSM, we may want to derive the outputs from the state memory and is independent of primary inputs.
Finite State Machines

Types of FSM:

2 Types of FSM
   - Mealy
   - Moore

Finite State Machine (FSM) is a means to model a digital system which has some finite states in it. To show the transition between states, state/transition diagram can be drawn. State/transition tables can be drawn which indicates what is the present state and to which state there will be a transition.

Moore machine output depends only on the state whereas, mealy machine output depends on the input as well as the state.

Moore machine is easy to model whereas, Mealy machine is complex because it depends on the input also.

Finite State Machines

The analysis of an FSM has three basic steps:

(i) Determine the next-state and output functions which depend on current state and input

(ii) Use next-state and output to construct a state/output table that completely specifies the next state and output of the circuit for every possible combination of current state and input.

(iii) Draw a state diagram that presents the information from the previous step in graphical form.
Finite State Machines

**Mealy FSM:**

A Mealy machine consists of three logic blocks, namely input combinational logic circuit, state memory and an output combinational logic circuit.

Here the output combinational logic circuit has two inputs, namely, the output of the sequential logic circuit as well as the external input signal. So output of Mealy FSM depends on the present state of the circuit and also it depends on the external inputs.

**Moore FSM:**

A Moore machine consists of three logic blocks, namely input combinational logic circuit, state memory and an output combinational logic circuit.

Here the output combinational logic circuit has one input, namely, the output of the sequential logic circuit (state memory). So Moore FSM depends on the present state of the circuit and is independent of the primary inputs.
Finite State Machines

Example State Diagram of Mealy and Moore Machine:

Moore Machine

Mealy Machine

Verilog Programming for FSM:

Encoding Style:

Binary Encoding:
S0 = 00
S1 = 01
S2 = 10
S3 = 11

Gray Encoding:
S0 = 00
S1 = 01
S2 = 11
S3 = 10
Finite State Machines

//Mealy state diagram for the circuit
module Mealy_mdl (x,y,CLK,RST);
input x,CLK,RST;
output y;
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 RST)
  if (~RST) Prstate <= S0; //Initialize to state S0
  else Prstate <= Nxtstate; //Clock operations
Finite State Machines

always @(Prstate or x)       //Determine next state
  case (Prstate)
    S0: if (x) Nxtstate = S1;
    S1: if (x) Nxtstate = S3;
    else Nxtstate = S0;
    S2: if (~x) Nxtstate = S0;
    S3: if (x) Nxtstate = S2;
    else Nxtstate = S0;
  endcase

always @(Prstate or x)       //Evaluate output
  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'b0; else y = 1'b1;
  endcase
endmodule

Moore state diagram for the circuit:

Fig. 5-18: Sequential Circuit with JK Flip-Flop
Finite State Machines

//Moore state diagram (Fig. 5-19)
module Moore_mdl (x,AB,CLK,RST);
  input x,CLK,RST;
  output [1:0]AB;
  reg [1:0] state;
  parameter S0=2'b00,S1=2'b01,S2=2'b10,S3=2'b11;
always @(posedge CLK or negedge RST)
  if (~RST) state = S0; //Initialize to state S0
  else
    case(state)
      S0: if (~x) state = S1;
      S1: if (x) state = S2; else state = S3;
      S2: if (~x) state = S3;
      S3: if (~x) state = S0;
    endcase
  assign AB = state; //Output of flip-flops
endmodule

Finite State Machines

Sequence Recognizer or Sequence Detector:

Identify the sequence 1101, regardless of where it occurs in a longer sequence.

Fig. 4-21 Construction of a State Diagram
Finite State Machines

**TABLE 4-5**
State Table for State Diagram in Figure 4-21

<table>
<thead>
<tr>
<th>Present State</th>
<th>Next State</th>
<th>Output Z</th>
<th>Output Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X = 0</td>
<td>X = 1</td>
<td>X = 0</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>B</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4-5  State Table for State Diagram in Figure 4-21

```
module seq_recognizer(CLK,RST,X,Z);
    input CLK,RST,X;
    output Z;
    reg [1:0] state, next_state;
    parameter A=2'b00,B=2'b01,C=2'b10,D=2'b11;
    reg Z;
    always@(posedge CLK or posedge RST) begin
        if(RST==1) state <= A;
        else state <= next_state;
    end

    always@(X or state) begin
        case(state)
            2'b00:
                Z <= 0;
            2'b01:
                Z <= 1;
            2'b10:
                Z <= 0;
            2'b11:
                Z <= 1;
        endcase
    end
endmodule
```
Finite State Machines

A: if(X) next_state <= B; else next_state <= A;
B: if(X) next_state <= C; else next_state <= A;
C: if(X) next_state <= C; else next_state <= D;
D: if(X) next_state <= B; else next_state <= A;
endcase
end
always @(X or state) begin
   case(state)
      A: Z<=0;
      B: Z<=0;
      C: Z<=0;
      D: Z<=X?1:0;
   endcase
end
endmodule

PHASE-III


AGENDA:

- Project Work
- Quiz Test
- Closing (Feedback) Meeting of Summer Training
PROJECT WORK

Design of Traffic Signal Controller:

- Main Highway
- Country Road

Problem Statements:

- The traffic signal for the main highway gets highest priority because cars are continuously present on the main highway. Thus the main highway signal remains green by default.
- Occasionally, cars from the country road arrive at the traffic signal. The traffic signal for the country road must turn green only long enough to let the cars on the country road go.
- As soon as there are no cars on the country road, the country road traffic signal turns yellow and then red and the traffic signal on the main highway turns green again.
- There is a sensor to detect cars waiting on the country road. The sensor sends a signal X as input to the controller. X=1 if there are cars on the country road; otherwise, X=0.
State Definitions for Traffic Signal Controller:

<table>
<thead>
<tr>
<th>State</th>
<th>Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hwy</td>
</tr>
<tr>
<td>S0</td>
<td>G</td>
</tr>
<tr>
<td>S1</td>
<td>Y</td>
</tr>
<tr>
<td>S2</td>
<td>R</td>
</tr>
<tr>
<td>S3</td>
<td>R</td>
</tr>
<tr>
<td>S4</td>
<td>R</td>
</tr>
</tbody>
</table>

There are delays on transitions from S1 to S2, from S2 to S3, and from S4 to S0. The delays must be controllable.
PROJECT WORK

ALL THE BEST for PROJECT WORK....

THE REAL LEARNING BEGINS FROM NOW....

Bhubaneswar Institute of Technology