|  |
| --- |
| **DIGITAL SYSTEM DESIGN LABORATORY** |
| **REPORT LAB 5** |

**SINGLE CYCLE MICROPROCESSOR DESIGN**

**Instruction Datapath**

**Part 1**

Phạm Quang Vinh ITITIU21347

### I. LAB OBJECTIVES

### This Lab experiments are intended to design and test a Single Cycle Microprocessor

### II. DESCRIPTION

### Single Cycle Microprocessor datapath to be implemented is in figure 2.1.

### 

### Figure 2.1: Single Cycle Microprocessor DataPath

### III. LAB PROCEDURE

### III.1 EXPERIMENT NO. 1

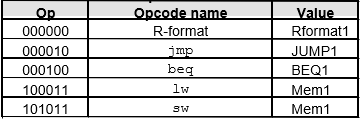
##### III.1.1 AIM: To understand and write the assembly code using MIPS Instruction set

##### register number of MIPS compiler conventions

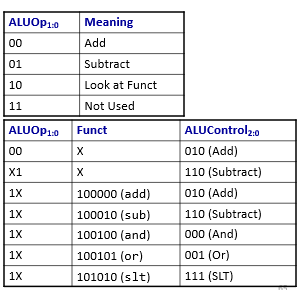
##### 

##### MIPS Assembly Language Sumarize:

##### Operation code (Op code) sumarize:

****

**ALU opcode and Function**

****

##### Instruction Formats

**Timeline

Description automatically generated**

**III.1.2 CODE**

**a) Assembly Code sample 1:**

Instruction Meaning

addi $s0, $zero, 33 load immediate value 33 to register $s0

addi $s1, $zero, 66 load immediate value 66 to register $s1

add $s2, $s0, $s1 $s2 = $s0 + $s1  
sub $s3, $s1, $s0 $s1 = $s1 – $s0  
sw $s3, 10($s2) Memory[$s2+10] = $s3

lw $s1, 10($s2) $s1 = Memory[$s2+10]

**b) Assembly Code sample 2:**

Assume the code start from address PC=0x00000000, one instruction is store in one memory location.

Instruction Meaning

addi $s2, $zero, 55 load immediate value 55 to register $S2

addi $s3, $zero, 22 load immediate value 22 to register $S3

addi $s5, $zero, 33 load immediate value 55 to register $S3

add $s4,$s2,$s3 $s4 = $s2 + $s3  
sub $s1,$s2,$s3 $s1 = $s2 – $s3  
sw $s1,100($s2) Memory[$s2+100] = $s1

lw $s1,100($s2) $s1 = Memory[$s2+100]   
bne $s1,$s5,End Next instr. is at End if $s4 !=$s5

addi $s6, $zero, 10 load immediate value 10 to register $s6  
beq $s4,$s5, End Next instr. is at End if $s4 = $s5

addi $s6, $zero, 20 load immediate value 20 to register $s6

End: j End jump Here

**III.1.3 LAB ASSIGNMENT**a) Compile the Assembly **Assembly Code sample 1** into machine code (decimal code and binary code)

|  |  |
| --- | --- |
| **Instruction** | **Meaning** |
| addi $s0, $zero, 33 | load immediate value 33 to register $s0 |
| addi $s1, $zero, 66 | load immediate value 66 to register $s1 |
| add $s2, $s0, $s1 | $s2 = $s0 + $s1 |
| sub $s3, $s1, $s0 | $s3 = $s1 – $s0 |
| sw $s3, 10($s2) | Memory[$s2+10] = $s3 |
| lw $s1, 10($s2) | $s1 = Memory[$s2+10] |

b) Explain briefly the meaning of **Assembly Code sample 1**

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Decimal (Hex) Code** | **Binary Code** |
| addi $s0, $zero, 33 | 20100021 (0x20100021) | 001000 00000 10000 0000 0000 0010 0001 |
| addi $s1, $zero, 66 | 20110042 (0x20110042) | 001000 00000 10001 0000 0000 0100 0010 |
| add $s2, $s0, $s1 | 02119020 (0x02119020) | 000000 10000 10001 10010 00000 100000 |
| sub $s3, $s1, $s0 | 02308022 (0x02308022) | 000000 10001 10000 10011 00000 100010 |
| sw $s3, 10($s2) | AE43000A (0xAE43000A) | 101011 10010 10011 0000 0000 0000 1010 |
| lw $s1, 10($s2) | 8E51000A (0x8E51000A) | 100011 10010 10001 0000 0000 0000 1010 |

c) Compile the Assembly **Assembly Code sample 2** into machine code (decimal code and binary code)

|  |  |
| --- | --- |
| **Instruction** | **Meaning** |
| addi $s2, $zero, 55 | load immediate value 55 to register $s2 |
| addi $s3, $zero, 22 | load immediate value 22 to register $s3 |
| addi $s5, $zero, 33 | load immediate value 33 to register $s5 |
| add $s4, $s2, $s3 | $s4 = $s2 + $s3 |
| sub $s1, $s2, $s3 | $s1 = $s2 – $s3 |
| sw $s1, 100($s2) | Memory[$s2+100] = $s1 |
| lw $s1, 100($s2) | $s1 = Memory[$s2+100] |
| bne $s1, $s5, End | Next instruction is at End if $s1 != $s5 |
| addi $s6, $zero, 10 | load immediate value 10 to register $s6 |
| beq $s4, $s5, End | Next instruction is at End if $s4 == $s5 |
| addi $s6, $zero, 20 | load immediate value 20 to register $s6 |
| End: j End | Jump to End |

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Decimal (Hex) Code** | **Binary Code** |
| addi $s2, $zero, 55 | 20120037 (0x20120037) | 001000 00000 10010 0000 0000 0011 0111 |
| addi $s3, $zero, 22 | 20130016 (0x20130016) | 001000 00000 10011 0000 0000 0001 0110 |
| addi $s5, $zero, 33 | 20150021 (0x20150021) | 001000 00000 10101 0000 0000 0010 0001 |
| add $s4, $s2, $s3 | 02432020 (0x02432020) | 000000 10010 10011 10100 00000 100000 |
| sub $s1, $s2, $s3 | 02431822 (0x02431822) | 000000 10010 10011 10001 00000 100010 |
| sw $s1, 100($s2) | AE510064 (0xAE510064) | 101011 10010 10001 0000 0000 0110 0100 |
| lw $s1, 100($s2) | 8E510064 (0x8E510064) | 100011 10010 10001 0000 0000 0110 0100 |
| bne $s1, $s5, End | 16150002 (0x16150002) | 000101 10001 10101 0000 0000 0000 0010 |
| addi $s6, $zero, 10 | 2016000A (0x2016000A) | 001000 00000 10110 0000 0000 0000 1010 |
| beq $s4, $s5, End | 12150001 (0x12150001) | 000100 10100 10101 0000 0000 0000 0001 |
| addi $s6, $zero, 20 | 20160014 (0x20160014) | 001000 00000 10110 0000 0000 0001 0100 |
| j End | 0800000C (0x0800000C) | 000010 0000 0000 0000 0000 0000 1100 |

d) Explain briefly the meaning of **Assembly Code sample 2**

1. addi $s2, $zero, 55 - Load the immediate value 55 into register $s2.
2. addi $s3, $zero, 22 - Load the immediate value 22 into register $s3.
3. addi $s5, $zero, 33 - Load the immediate value 33 into register $s5.
4. add $s4, $s2, $s3 - Add the values in registers $s2 and $s3, and store the result in $s4.
5. sub $s1, $s2, $s3 - Subtract the value in $s3 from the value in $s2, and store the result in $s1.
6. sw $s1, 100($s2) - Store the value in $s1 to the memory location at the address $s2 + 100.
7. lw $s1, 100($s2) - Load the value from the memory location at the address $s2 + 100 into register $s1.
8. bne $s1, $s5, End - Branch to End if the values in registers $s1 and `$

### III.2 EXPERIMENT NO. 2

##### III.2.1 AIM: To implement R-Type Datapath

##### 

##### Given the assembly code

addi $s1, $zero, 33 load immediate value 33 to register $s0

addi $s2, $zero, 66 load immediate value 66 to register $s1

##### add $s0, $s1, $s2 $s0 = $s1 + $s2

##### Translate into Macchine code:

##### 

##### 

##### Modify the code register file:

##### Assign initial value of the register 17=33 ( $s1=33)

##### Assign initial value of the register 18=33 ( $s2=66)

##### 

##### 

##### 

**III.2.2 CODE**

module R\_Type\_Add(

input [4:0] rs, // Source register 1

input [4:0] rt, // Source register 2

input [4:0] rd, // Destination register

input [2:0] ALUop, // ALU operation (for future expansion if needed)

input Zero, // Zero flag (for future use, not used in ADD)

output reg [31:0] ALU\_out // ALU result output

);

always @(\*) begin

case(ALUop)

// For now, assuming ALUop is not used and only ADD operation is performed

3'b000: ALU\_out = rs + rt; // ADD operation

default: ALU\_out = 0; // Default to zero (shouldn't happen in a real ALU)

endcase

end

endmodule

**III.2.3 LAB ASSIGNMENT**1) Write Verilog code to implement R\_Type\_Add module

module R\_Type\_Add(

input [4:0] rs, // Source register 1

input [4:0] rt, // Source register 2

input [4:0] rd, // Destination register

input [2:0] ALUop, // ALU operation (for future expansion if needed)

input Zero, // Zero flag (for future use, not used in ADD)

output reg [31:0] ALU\_out // ALU result output

);

always @(\*) begin

case(ALUop)

// For now, assuming ALUop is not used and only ADD operation is performed

3'b000: ALU\_out = rs + rt; // ADD operation

default: ALU\_out = 0; // Default to zero (shouldn't happen in a real ALU)

endcase

end

endmodule

2) Write testbenches to verify R\_Type\_Add module, simulate and check the simulation output data.

module tb\_R\_Type\_Add;

// Inputs

reg [4:0] rs, rt, rd;

reg [2:0] ALUop;

reg Zero;

// Outputs

wire [31:0] ALU\_out;

// Instantiate R\_Type\_Add module

R\_Type\_Add uut (

.rs(rs),

.rt(rt),

.rd(rd),

.ALUop(ALUop),

.Zero(Zero),

.ALU\_out(ALU\_out)

);

// Initialize inputs

initial begin

rs = 17; // $s1 = 33

rt = 18; // $s2 = 66

rd = 16; // $s0

ALUop = 3'b000; // ADD operation

Zero = 0;

// Stimulus

#10; // Wait for 10 time units

// Add more stimulus here if needed

// Display result

$display("ALU Result (ADD) = %d", ALU\_out);

// Add assertions if required

// $assert(condition, message);

// End simulation

$finish;

end

endmodule

3) Write Top level Verilog code to implement R\_Type\_Add module in FPGA Kit

module top\_level\_module (

// Define your inputs and outputs here

input wire clk, // Clock input

input wire reset, // Reset input

// Other inputs and outputs as needed

);

// Define your signals and instances here

reg [4:0] rs, rt, rd;

reg [2:0] ALUop;

reg Zero;

wire [31:0] ALU\_out;

// Instantiate R\_Type\_Add module

R\_Type\_Add uut (

.rs(rs),

.rt(rt),

.rd(rd),

.ALUop(ALUop),

.Zero(Zero),

.ALU\_out(ALU\_out)

);

// Add connections to other modules in your datapath

// Example:

// module\_A u\_module\_A (.input1(signal1), .output1(signal2));

// module\_B u\_module\_B (.input2(signal2), .output2(signal3));

// Define clock and reset logic if necessary

endmodule

### III.3 EXPERIMENT NO. 3

##### III.3.1 AIM: To implement SW I-Type Instruction Datapath

##### 

**III.3.2 CODE**

module SW\_datapath(

input [31:0] rs, // Register rs input

input [31:0] rt, // Register rt input

input [15:0] offset, // Offset for memory access

input [1:0] ALUOp, // ALU operation code

input MemWrite, // Memory write control signal

input MemRead, // Memory read control signal

output reg [31:0] Mem\_out // Memory data output

);

reg [31:0] ALU\_result; // ALU result

reg [31:0] addr; // Memory address

// ALU operation based on ALUOp

always @\* begin

case (ALUOp)

2'b00: ALU\_result = rs + rt; // ALUOp = 00, addition

2'b01: ALU\_result = rs - rt; // ALUOp = 01, subtraction

2'b10: ALU\_result = rs & rt; // ALUOp = 10, bitwise AND

2'b11: ALU\_result = rs | rt; // ALUOp = 11, bitwise OR

default: ALU\_result = 0; // Default case

endcase

end

// Memory address calculation

always @\* begin

addr = rs + offset;

end

// Memory access logic

always @\* begin

if (MemWrite) begin

// Perform memory write operation

// Assuming some memory write operation here

// Replace with actual memory write logic

end else if (MemRead) begin

// Perform memory read operation

// Assuming some memory read operation here

// Replace with actual memory read logic

// For now, just assign a value to Mem\_out

Mem\_out = addr; // Dummy assignment, replace with actual memory data read

end

end

endmodule

**III.3.3 LAB ASSIGNMENT**1) Write Verilog code to implement SW\_datapath module

Above !!!

2) Write testbenches to verify SW\_datapath module, simulate and verify the output data.

module tb\_SW\_datapath;

// Inputs

reg [31:0] rs, rt;

reg [15:0] offset;

reg [1:0] ALUOp;

reg MemWrite, MemRead;

// Outputs

wire [31:0] Mem\_out;

// Instantiate the module

SW\_datapath dut (

.rs(rs),

.rt(rt),

.offset(offset),

.ALUOp(ALUOp),

.MemWrite(MemWrite),

.MemRead(MemRead),

.Mem\_out(Mem\_out)

);

// Clock generation

reg clk = 0;

always #5 clk = ~clk;

// Test stimulus

initial begin

// Initialize inputs

rs = 32'hABCDEFF0;

rt = 32'h12345678;

offset = 16'h1234;

ALUOp = 2'b00; // Addition

MemWrite = 0;

MemRead = 1;

// Apply stimulus and observe outputs

#10; // Wait for initial setup

$display("rs = %h, rt = %h, offset = %h, ALUOp = %b, MemWrite = %b, MemRead = %b",

rs, rt, offset, ALUOp, MemWrite, MemRead);

#10; // Wait a few cycles

// Add more test cases or assertions as needed

// End simulation

$finish;

end

endmodule

3) Write Top level Verilog code to implement SW\_datapath module in FPGA Kit

module tb\_SW\_datapath;

// Inputs

reg [31:0] rs, rt;

reg [15:0] offset;

reg [1:0] ALUOp;

reg MemWrite, MemRead;

// Outputs

wire [31:0] Mem\_out;

// Instantiate the module

SW\_datapath dut (

.rs(rs),

.rt(rt),

.offset(offset),

.ALUOp(ALUOp),

.MemWrite(MemWrite),

.MemRead(MemRead),

.Mem\_out(Mem\_out)

);

// Clock generation

reg clk = 0;

always #5 clk = ~clk;

// Test stimulus

initial begin

// Initialize inputs

rs = 32'hABCDEFF0;

rt = 32'h12345678;

offset = 16'h1234;

ALUOp = 2'b00; // Addition

MemWrite = 0;

MemRead = 1;

// Apply stimulus and observe outputs

#10; // Wait for initial setup

$display("rs = %h, rt = %h, offset = %h, ALUOp = %b, MemWrite = %b, MemRead = %b",

rs, rt, offset, ALUOp, MemWrite, MemRead);

#10; // Wait a few cycles

// Add more test cases or assertions as needed

// End simulation

$finish;

end

endmodule

### III.4 EXPERIMENT NO. 4

##### III.4.1 AIM: To implement LW I-Type Instruction Datapath

##### 

**III.4.2 CODE**

module Datapath\_LW\_Type(

input clk, // Clock input

input reset, // Reset signal

input RegWrite, // Register write enable

input MEM\_read, // Memory read enable

input MEM\_write, // Memory write enable

input [2:0] ALU\_op, // ALU operation code

input [4:0] rs, // Source register rs

input [4:0] rt, // Source/destination register rt

input [1:0] A\_ckeck, // Some other input

input [15:0] offset, // Offset for load/store

output reg Zero, // Zero flag output

output reg [31:0] LW\_out // Load word output

);

reg [31:0] rs\_data; // Data read from rs register

reg [31:0] rt\_data; // Data read from rt register

reg [31:0] address; // Calculated memory address

reg [31:0] mem\_data; // Data read from memory

reg [31:0] ALU\_result; // Result of ALU operation

reg [4:0] rs\_index; // Index for rs register

reg [4:0] rt\_index; // Index for rt register

// Register file

reg [31:0] registers [31:0];

// ALU operations based on ALU\_op

always @(\*) begin

case (ALU\_op)

3'b000: ALU\_result = rs\_data + rt\_data; // Add

3'b001: ALU\_result = rs\_data - rt\_data; // Subtract

3'b010: ALU\_result = rs\_data & rt\_data; // Bitwise AND

3'b011: ALU\_result = rs\_data | rt\_data; // Bitwise OR

3'b100: ALU\_result = rs\_data ^ rt\_data; // Bitwise XOR

default: ALU\_result = 32'b0; // Default to zero

endcase

end

// Memory read and write logic

always @(posedge clk or posedge reset) begin

if (reset) begin

rs\_data <= 32'b0;

rt\_data <= 32'b0;

address <= 32'b0;

mem\_data <= 32'b0;

rs\_index <= 5'b0;

rt\_index <= 5'b0;

Zero <= 1'b0;

LW\_out <= 32'b0;

end else begin

// Register file access

rs\_data <= registers[rs];

rt\_data <= registers[rt];

// Calculate memory address for load/store

address <= rs\_data + offset;

// Handle ALU operations

case (ALU\_op)

3'b000: ALU\_result <= rs\_data + rt\_data; // Add

3'b001: ALU\_result <= rs\_data - rt\_data; // Subtract

3'b010: ALU\_result <= rs\_data & rt\_data; // Bitwise AND

3'b011: ALU\_result <= rs\_data | rt\_data; // Bitwise OR

3'b100: ALU\_result <= rs\_data ^ rt\_data; // Bitwise XOR

default: ALU\_result <= 32'b0; // Default to zero

endcase

// Memory read/write

if (MEM\_read) begin

mem\_data <= registers[address];

end

if (MEM\_write) begin

registers[address] <= rt\_data;

end

// Zero flag logic

if (ALU\_result == 32'b0) begin

Zero <= 1'b1;

end else begin

Zero <= 1'b0;

end

// Load word output

if (MEM\_read) begin

LW\_out <= mem\_data;

end

end

end

endmodule

**III.4.3 LAB ASSIGNMENT**1) Write Verilog code to implement LW\_datapath module

Above !!!

2) Write testbenches to verify LW\_datapath module, simulate and verify the output data.

`timescale 1ns / 1ps

module testbench();

// Inputs

reg clk;

reg reset;

reg RegWrite;

reg MEM\_read;

reg MEM\_write;

reg [2:0] ALU\_op;

reg [4:0] rs;

reg [4:0] rt;

reg [1:0] A\_ckeck;

reg [15:0] offset;

// Outputs

wire Zero;

wire [31:0] LW\_out;

// Instantiate LW\_datapath module

LW\_datapath dut (

.clk(clk),

.reset(reset),

.RegWrite(RegWrite),

.MEM\_read(MEM\_read),

.MEM\_write(MEM\_write),

.ALU\_op(ALU\_op),

.rs(rs),

.rt(rt),

.A\_ckeck(A\_ckeck),

.offset(offset),

.Zero(Zero),

.LW\_out(LW\_out)

);

// Clock generation

initial begin

clk = 0;

forever #5 clk = ~clk;

end

// Test stimulus

initial begin

// Reset sequence

reset = 1;

#10 reset = 0;

// Test case 1: Perform a load operation

RegWrite = 1;

MEM\_read = 1;

MEM\_write = 0;

ALU\_op = 3'b000; // Add

rs = 5'b00100; // Register index 20

rt = 5'b01000; // Register index 40

A\_ckeck = 2'b01;

offset = 16'd100; // Memory offset

// Wait for a few clock cycles

#100;

// Test case 2: Perform another operation

// You can add more test cases here

// End simulation

$finish;

end

endmodule

3) Write Top level Verilog code to implement LW\_datapath module in FPGA Kit

module top\_module (

// Clock and reset

input wire clk,

input wire reset,

// Other inputs specific to your application

input wire [2:0] ALU\_op,

input wire [4:0] rs,

input wire [4:0] rt,

input wire [15:0] offset,

// Additional inputs as needed

// Outputs from LW\_datapath module

output reg Zero,

output reg [31:0] LW\_out

);

// Instantiate LW\_datapath module

LW\_datapath LW\_datapath\_inst (

.clk(clk),

.reset(reset),

.RegWrite(RegWrite),

.MEM\_read(MEM\_read),

.MEM\_write(MEM\_write),

.ALU\_op(ALU\_op),

.rs(rs),

.rt(rt),

.A\_ckeck(A\_ckeck),

.offset(offset),

.Zero(Zero),

.LW\_out(LW\_out)

);

// Additional logic for your application

endmodule

### III.5 EXPERIMENT NO. 5

##### III.5.1 AIM: To implement I-Type Instruction Beq datapath

##### 

**III.5.2 CODE**

module beq\_Datapath (

input [31:0] rs, // Register value of rs

input [31:0] rt, // Register value of rt

input [15:0] offset, // Offset for branch

input [1:0] ALUOp, // ALU operation control

input [31:0] PC\_plus\_4, // PC + 4

input Zero, // Zero signal from ALU

output [31:0] beq\_add // Branch address

);

// Internal signals

wire [31:0] branch\_target;

// Calculation of branch target address

assign branch\_target = PC\_plus\_4 + ({offset, 16'b0});

// Control logic for branch condition (BEQ)

always @\* begin

if (Zero && ALUOp == 2'b00) // ALUOp == 2'b00 indicates BEQ operation

beq\_add = branch\_target;

else

beq\_add = 32'b0; // No branch

end

endmodule

**III.5.3 LAB ASSIGNMENT**1) Write Verilog code to implement beq\_Datapath module

Above

2) Write testbenches to verify beq\_Datapath module, simulate and verify the output data.

`timescale 1ns / 1ps // Timescale directive

module testbench;

// Inputs to beq\_Datapath module

reg [31:0] rs, rt, PC\_plus\_4;

reg [15:0] offset;

reg [1:0] ALUOp;

reg Zero;

// Output from beq\_Datapath module

wire [31:0] beq\_add;

// Instantiate the beq\_Datapath module

beq\_Datapath dut (

.rs(rs),

.rt(rt),

.offset(offset),

.ALUOp(ALUOp),

.PC\_plus\_4(PC\_plus\_4),

.Zero(Zero),

.beq\_add(beq\_add)

);

// Initial stimulus

initial begin

// Test case 1: BEQ condition true

rs = 32'h00000010; // Example register values

rt = 32'h00000010;

offset = 16'h0001; // Example offset

ALUOp = 2'b00; // BEQ operation

PC\_plus\_4 = 32'h1000; // Example PC + 4 value

Zero = 1'b1; // BEQ condition met

// Test case 2: BEQ condition false

// Uncomment to test another case

// rs = 32'h00000020;

// rt = 32'h00000030;

// offset = 16'h0001;

// ALUOp = 2'b00;

// PC\_plus\_4 = 32'h2000;

// Zero = 1'b0;

// Add delay to observe outputs

#10;

// Display results

$display("Test Case 1: rs=%h, rt=%h, offset=%h, ALUOp=%b, PC\_plus\_4=%h, Zero=%b, beq\_add=%h", rs, rt, offset, ALUOp, PC\_plus\_4, Zero, beq\_add);

// Uncomment to display another case

// $display("Test Case 2: rs=%h, rt=%h, offset=%h, ALUOp=%b, PC\_plus\_4=%h, Zero=%b, beq\_add=%h", rs, rt, offset, ALUOp, PC\_plus\_4, Zero, beq\_add);

// End simulation

$finish;

end

endmodule

3) Write Top level Verilog code to implement beq\_Datapath module in FPGA Kit

`timescale 1ns / 1ps // Timescale directive

module top\_module (

// Add your FPGA kit-specific ports here

);

// Inputs and outputs specific to your FPGA kit

// Example:

input clk, reset;

input [31:0] rs, rt, PC\_plus\_4;

input [15:0] offset;

input [1:0] ALUOp;

input Zero;

output reg [31:0] beq\_add;

// Instantiate the beq\_Datapath module

beq\_Datapath dut (

.rs(rs),

.rt(rt),

.offset(offset),

.ALUOp(ALUOp),

.PC\_plus\_4(PC\_plus\_4),

.Zero(Zero),

.beq\_add(beq\_add)

);

// Other logic and connections for your FPGA implementation

endmodule

**III. LAB REPORT GUIDELINES**

Students write up a report on the Verilog HDL implementation experiment projects created in this lab. The lab report should include Verilog code for the module under test, Verilog test bench code and a truth table results, and example data input and output to validate the experiment. Simulation Result in form of Simulation Capture Screen. The implementation results in FPGA Kit, compare the simulation results and implementation results.