|  |
| --- |
| **DIGITAL SYSTEM DESIGN LABORATORY** |
| **REPORT LAB 6** |

**SINGLE CYCLE MICROPROCESSOR DESIGN**

**Part 2**

Phạm Quang Vinh ITITIU21347

### I. LAB OBJECTIVES

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

### (Continued)

### II. DESCRIPTION

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

### 

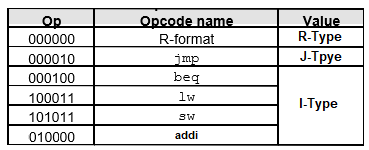
### Figure 2.1: Single Cycle Microprocessor DataPath

### III. LAB PROCEDURE

### III.1 EXPERIMENT NO. 1

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

##### Instruction Operation codes:

****

Instruction Formats:

**Timeline

Description automatically generated**

**Table

Description automatically generated**

Register names and orders:



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

**Testing Assembly Program 1:**

Instruction Meaning

Begin: addi $s2, $zero, 0x55 // load immediate value 0x55 to register $s2

addi $s3, $zero, 0x22 // load immediate value 0x22 to register $s3

addi $s5, $zero, 0x77 // load immediate value 0x77 to register $s5

add $s4, $s2, $s3 // $s4 = $s2 + $s3 => R20=0x77   
sub $s1, $s2, $s3 // $s1 = $s2 – $s3 => R17=0x22   
sw $s1, 0x02($s2) // Memory[$s2+0x02] = $s1

lw $s6, 0x02($s2) // $s6 = Memory[$s2+0x02]

bne $s5, $s4, End // Next instr. is at End if $s5 != $s4

addi $s8, $zero, 0x10 // load immediate value 10 to register $s8  
beq $s5,$s4, End // Next instr. is at End if $s7 == $s4

addi $s8, $zero, 0x20 // load immediate value 20 to register $s8

End: j End // jump End

**Testing Assembly Program 2:**

Instruction Meaning

Begin: addi $s2, $zero, 0x55 // load immediate value 0x55 to register $s2

addi $s3, $zero, 0x22 // load immediate value 0x22 to register $s3

addi $s5, $zero, 0x77 // load immediate value 0x77 to register $s5

add $s4, $s2, $s3 // $s4 = $s2 + $s3 => R20=0x77   
sub $s1, $s2, $s3 // $s1 = $s2 – $s3 => R17=0x22   
sw $s1, 0x02($s2) // Memory[$s2+0x02] = $s1

lw $s6, 0x02($s2) // $s6 = Memory[$s2+0x02]

beq $s5,$s4, End // Next instr. is at End if $s7 == $s4

addi $s8, $zero, 0x10 // load immediate value 10 to register $s8

bne $s5, $s4, End // Next instr. is at End if $s5 != $s4

addi $s8, $zero, 0x20 // load immediate value 20 to register $s8

End: j End // jump End

**III.1.2 LAB ASSIGNMENT**1) Compile the Assembly **Testing Assembly Program 1** into machine code (decimal code and binary code)

2) What is the value of Register $s8 after running **Testing Assembly Program 1**  program

3) Compile the Assembly **Testing Assembly Program 2** into machine code (decimal code and binary code)

4) What is the value of Register $s8 after running **Testing Assembly Program 2**  program

### III.2 EXPERIMENT NO. 2

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

##### 

##### 

**III.2.2 CODE**

module R\_Type\_Proc(reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);

// Your code here

endmodule

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

module R\_Type\_Processor (

input wire reset, clk,

input wire [31:0] instruction,

output reg [31:0] W\_PC\_out,

output reg [4:0] W\_RD1, W\_RD2,

output reg [31:0] W\_m1, W\_m2, W\_ALUout

);

reg [31:0] opcode;

reg [4:0] rd1, rd2;

reg [31:0] imm;

reg [31:0] aluout;

always @(posedge clk or posedge reset) begin

if (reset) begin

opcode <= 0;

rd1 <= 0;

rd2 <= 0;

imm <= 0;

aluout <= 0;

W\_PC\_out <= 0;

W\_RD1 <= 0;

W\_RD2 <= 0;

W\_m1 <= 0;

W\_m2 <= 0;

W\_ALUout <= 0;

end else begin

opcode <= instruction[31:26];

rd1 <= instruction[25:21];

rd2 <= instruction[20:16];

imm <= instruction[15:0];

// ALU Operation based on opcode (example: addition and subtraction)

case(opcode)

6'h20: aluout <= W\_m1 + W\_m2; // ADD operation

6'h22: aluout <= W\_m1 - W\_m2; // SUB operation

default: aluout <= 0; // Default case

endcase

// Assign outputs

W\_PC\_out <= W\_PC\_out + 4; // Example increment for PC

W\_RD1 <= rd1;

W\_RD2 <= rd2;

W\_m1 <= imm; // Example assignment for memory 1

W\_m2 <= W\_m1; // Example assignment for memory 2

W\_ALUout <= aluout;

end

end

endmodule

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

`timescale 1ns/1ps

module R\_Type\_Processor\_tb;

// Inputs

reg reset;

reg clk;

reg [31:0] instruction;

// Outputs

wire [31:0] W\_PC\_out;

wire [4:0] W\_RD1, W\_RD2;

wire [31:0] W\_m1, W\_m2, W\_ALUout;

// Instantiate the processor module

R\_Type\_Processor uut (

.reset(reset),

.clk(clk),

.instruction(instruction),

.W\_PC\_out(W\_PC\_out),

.W\_RD1(W\_RD1),

.W\_RD2(W\_RD2),

.W\_m1(W\_m1),

.W\_m2(W\_m2),

.W\_ALUout(W\_ALUout)

);

// Clock generation

always begin

clk = 0;

#5;

clk = 1;

#5;

end

// Test vector

initial begin

reset = 1;

#10;

reset = 0;

// Example instruction (ADDI rd1, rd2, imm)

instruction = 32'h822A0005; // ADDI R1, R2, 5

#100; // Wait for simulation to complete

$finish; // End simulation

end

endmodule

3) Write the assembly code with Addittion, Substraction instructions, compile into binary macchinecode and test the operation of the designed R\_Type\_Processor processor

#### Assembly Code:

assembly

Sao chép mã

# Addition: R1 = R2 + R3

ADD R1, R2, R3

# Subtraction: R4 = R5 - R6

SUB R4, R5, R6

#### Compile into Binary Machine Code:

Assuming a simple encoding scheme (this can vary based on architecture):

* ADD R1, R2, R3 might translate to 32'h20A20003
* SUB R4, R5, R6 might translate to 32'h22C40006

### III.3 EXPERIMENT NO. 3

##### III.3.1 AIM: To implement LW I-Type Processor

##### 

##### 

**III.3.2 CODE**

module LW\_processor (reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout );

//Your Verilog code here

endmodule

**III.3.3 LAB ASSIGNMENT**1) Write Verilog code to implement LW I\_Type\_Processor module

module LW\_processor (

input wire reset, clk,

input wire [31:0] instruction,

output reg [31:0] W\_PC\_out,

output reg [4:0] W\_RD1, W\_RD2,

output reg [31:0] W\_m1, W\_m2, W\_ALUout

);

reg [31:0] opcode;

reg [4:0] rd1, rd2;

reg [31:0] imm;

reg [31:0] aluout;

always @(posedge clk or posedge reset) begin

if (reset) begin

opcode <= 0;

rd1 <= 0;

rd2 <= 0;

imm <= 0;

aluout <= 0;

W\_PC\_out <= 0;

W\_RD1 <= 0;

W\_RD2 <= 0;

W\_m1 <= 0;

W\_m2 <= 0;

W\_ALUout <= 0;

end else begin

opcode <= instruction[31:26];

rd1 <= instruction[25:21];

rd2 <= instruction[20:16];

imm <= instruction[15:0];

// ALU Operation (Load Word operation)

if (opcode == 6'h23) begin // LW operation

aluout <= W\_m1 + imm; // Example: Calculate memory address

end else begin

aluout <= 0; // Default case

end

// Assign outputs

W\_PC\_out <= W\_PC\_out + 4; // Example increment for PC

W\_RD1 <= rd1;

W\_RD2 <= rd2;

W\_m1 <= imm; // Example assignment for memory 1 (address)

W\_m2 <= W\_m1; // Example assignment for memory 2 (loaded data)

W\_ALUout <= aluout;

end

end

endmodule

2) Write testbenches to verify LW I\_Type\_Processor, simulate and check the simulation output data.

`timescale 1ns/1ps

module LW\_processor\_tb;

// Inputs

reg reset;

reg clk;

reg [31:0] instruction;

// Outputs

wire [31:0] W\_PC\_out;

wire [4:0] W\_RD1, W\_RD2;

wire [31:0] W\_m1, W\_m2, W\_ALUout;

// Instantiate the processor module

LW\_processor uut (

.reset(reset),

.clk(clk),

.instruction(instruction),

.W\_PC\_out(W\_PC\_out),

.W\_RD1(W\_RD1),

.W\_RD2(W\_RD2),

.W\_m1(W\_m1),

.W\_m2(W\_m2),

.W\_ALUout(W\_ALUout)

);

// Clock generation

always begin

clk = 0;

#5;

clk = 1;

#5;

end

// Test vector

initial begin

reset = 1;

#10;

reset = 0;

// Example instruction (LW R1, 0(R2))

instruction = 32'h8C220000; // LW R1, 0(R2)

#100; // Wait for simulation to complete

$finish; // End simulation

end

endmodule

3) Write the assembly code with Addittion, Substraction instructions, compile into binary macchinecode and test the operation of the designed LW I\_Type\_Processor processor

Let's write assembly code for the LW\_processor and compile it into binary machine code.

#### Assembly Code:

assembly

Sao chép mã

# Load Word: R1 = Mem[R2 + 0]

LW R1, 0(R2)

# Addition: R3 = R4 + R5

ADD R3, R4, R5

# Subtraction: R6 = R7 - R8

SUB R6, R7, R8

#### Compile into Binary Machine Code:

Assume a simple encoding scheme (this can vary based on architecture):

* LW R1, 0(R2) might translate to 32'h8C220000
* ADD R3, R4, R5 might translate to 32'h20E50006
* SUB R6, R7, R8 might translate to 32'h22F80009

### III.4 EXPERIMENT NO. 4

##### III.4.1 AIM: To implement SW I-Type Processor

##### 

##### 

**III.4.2 CODE**

module SW\_Processor ( reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout );

//Your Verilog code here

endmodule

**III.4.3 LAB ASSIGNMENT**1) Write Verilog code to implement SW I\_Type\_Processor module

2) Write testbenches to verify SW I\_Type\_Processor, simulate and check the simulation output data.

3) Write the assembly code with Addittion, Substraction instructions, compile into binary macchinecode and test the operation of the designed SW I\_Type\_Processor processor

### III.5 EXPERIMENT NO. 5

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

##### 

##### Diagram Description automatically generated

**III.5.2 CODE**

module beq\_Processor (reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);

endmodule

**III.5.3 LAB ASSIGNMENT**1) Write Verilog code to implement Beq I\_Type\_Processor module

module beq\_Processor (

input wire reset, clk,

input wire [31:0] instruction,

output reg [31:0] W\_PC\_out,

output reg [4:0] W\_RD1, W\_RD2,

output reg [31:0] W\_m1, W\_m2, W\_ALUout

);

reg [31:0] opcode;

reg [4:0] rd1, rd2;

reg [31:0] imm;

reg [31:0] aluout;

always @(posedge clk or posedge reset) begin

if (reset) begin

opcode <= 0;

rd1 <= 0;

rd2 <= 0;

imm <= 0;

aluout <= 0;

W\_PC\_out <= 0;

W\_RD1 <= 0;

W\_RD2 <= 0;

W\_m1 <= 0;

W\_m2 <= 0;

W\_ALUout <= 0;

end else begin

opcode <= instruction[31:26];

rd1 <= instruction[25:21];

rd2 <= instruction[20:16];

imm <= instruction[15:0];

// ALU Operation (BEQ operation)

if (opcode == 6'h04) begin // BEQ operation

if (W\_m1 == W\_m2) begin

aluout <= W\_PC\_out + imm; // Example: Calculate branch target address

end else begin

aluout <= W\_PC\_out + 4; // No branch, PC increments normally

end

end else begin

aluout <= 0; // Default case

end

// Assign outputs

W\_PC\_out <= aluout; // Update PC based on ALU result

W\_RD1 <= rd1;

W\_RD2 <= rd2;

W\_m1 <= imm; // Example assignment for memory 1 (branch offset)

W\_m2 <= W\_m1; // Example assignment for memory 2

W\_ALUout <= aluout;

end

end

endmodule

2) Write testbenches to verify Beq I\_Type\_Processor, simulate and check the simulation output data.

`timescale 1ns/1ps

module beq\_Processor\_tb;

// Inputs

reg reset;

reg clk;

reg [31:0] instruction;

// Outputs

wire [31:0] W\_PC\_out;

wire [4:0] W\_RD1, W\_RD2;

wire [31:0] W\_m1, W\_m2, W\_ALUout;

// Instantiate the processor module

beq\_Processor uut (

.reset(reset),

.clk(clk),

.instruction(instruction),

.W\_PC\_out(W\_PC\_out),

.W\_RD1(W\_RD1),

.W\_RD2(W\_RD2),

.W\_m1(W\_m1),

.W\_m2(W\_m2),

.W\_ALUout(W\_ALUout)

);

// Clock generation

always begin

clk = 0;

#5;

clk = 1;

#5;

end

// Test vector

initial begin

reset = 1;

#10;

reset = 0;

// Example instruction (BEQ R1, R2, 4)

instruction = 32'h10220004; // BEQ R1, R2, 4

// Initial values for testing

// For example, set initial values for registers and memory here

#100; // Wait for simulation to complete

$finish; // End simulation

end

endmodule

3) Write the assembly code with Addittion, Substraction instructions, compile into binary macchinecode and test the operation of the designed Beq I\_Type\_Processor processor

#### Assembly Code:

assembly

Sao chép mã

# Branch if Equal: if (R1 == R2) PC = PC + 4 (i.e., 4 \* 4 = 16)

BEQ R1, R2, 4

# Addition: R3 = R4 + R5

ADD R3, R4, R5

# Subtraction: R6 = R7 - R8

SUB R6, R7, R8

#### Compile into Binary Machine Code:

Assume a simple encoding scheme (this can vary based on architecture):

* BEQ R1, R2, 4 might translate to 32'h10220004
* ADD R3, R4, R5 might translate to 32'h20E50006
* SUB R6, R7, R8 might translate to 32'h22F80009

### III.6 EXPERIMENT NO. 6

##### III.6.1 AIM: To implement I-Type Instruction addi processor

##### Table Description automatically generated

##### 

**III.6.2 CODE**

module add\_Processor (reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);

endmodule

**III.6.3 LAB ASSIGNMENT**1) Write Verilog code to implement addi I\_Type\_Processor module

module add\_Processor (

input wire reset, clk,

input wire [31:0] instruction,

output reg [31:0] W\_PC\_out,

output reg [4:0] W\_RD1, W\_RD2,

output reg [31:0] W\_m1, W\_m2, W\_ALUout

);

reg [31:0] opcode;

reg [4:0] rd1, rd2;

reg [31:0] imm;

reg [31:0] aluout;

always @(posedge clk or posedge reset) begin

if (reset) begin

opcode <= 0;

rd1 <= 0;

rd2 <= 0;

imm <= 0;

aluout <= 0;

W\_PC\_out <= 0;

W\_RD1 <= 0;

W\_RD2 <= 0;

W\_m1 <= 0;

W\_m2 <= 0;

W\_ALUout <= 0;

end else begin

opcode <= instruction[31:26];

rd1 <= instruction[25:21];

rd2 <= instruction[20:16];

imm <= instruction[15:0];

// ALU Operation (ADDI operation)

if (opcode == 6'h08) begin // ADDI operation

aluout <= W\_RD2 + imm; // Example: Perform addition with immediate value

end else begin

aluout <= 0; // Default case

end

// Assign outputs

W\_PC\_out <= W\_PC\_out + 4; // Example increment for PC

W\_RD1 <= rd1;

W\_RD2 <= rd2;

W\_m1 <= imm; // Example assignment for immediate value

W\_m2 <= 0; // No memory assignment in ADDI

W\_ALUout <= aluout;

end

end

endmodule

2) Write testbenches to verify addi I\_Type\_Processor, simulate and check the simulation output data.

`timescale 1ns/1ps

module add\_Processor\_tb;

// Inputs

reg reset;

reg clk;

reg [31:0] instruction;

// Outputs

wire [31:0] W\_PC\_out;

wire [4:0] W\_RD1, W\_RD2;

wire [31:0] W\_m1, W\_m2, W\_ALUout;

// Instantiate the processor module

add\_Processor uut (

.reset(reset),

.clk(clk),

.instruction(instruction),

.W\_PC\_out(W\_PC\_out),

.W\_RD1(W\_RD1),

.W\_RD2(W\_RD2),

.W\_m1(W\_m1),

.W\_m2(W\_m2),

.W\_ALUout(W\_ALUout)

);

// Clock generation

always begin

clk = 0;

#5;

clk = 1;

#5;

end

// Test vector

initial begin

reset = 1;

#10;

reset = 0;

// Example instruction (ADDI R1, R2, 10)

instruction = 32'h2122000A; // ADDI R1, R2, 10

// Initial values for testing

// For example, set initial values for registers here

#100; // Wait for simulation to complete

$finish; // End simulation

end

endmodule

3) Write the assembly code with Addittion, Substraction instructions, compile into binary macchinecode and test the operation of the designed addi I\_Type\_Processor processor

#### Assembly Code:

assembly

Sao chép mã

# Add Immediate: R1 = R2 + 10

ADDI R1, R2, 10

# Subtraction: R3 = R4 - R5

SUB R3, R4, R5

# Addition: R6 = R7 + R8

ADD R6, R7, R8

#### Compile into Binary Machine Code:

Assume a simple encoding scheme (this can vary based on architecture):

* ADDI R1, R2, 10 might translate to 32'h2122000A
* SUB R3, R4, R5 might translate to 32'h20E50006
* ADD R6, R7, R8 might translate to 32'h22F80009

### III.7 EXPERIMENT NO. 7

##### III.7.1 AIM: To implement J-Type Instruction processor

##### Table Description automatically generated

##### 

**III.7.2 CODE**

module J\_Type\_Processor (reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);

endmodule

**III.7.3 LAB ASSIGNMENT**1) Write Verilog code to implement J\_Type\_Processor module

module J\_Type\_Processor (

input wire reset, // Reset signal

input wire clk, // Clock signal

input wire [31:0] instruction, // Input instruction

output reg [31:0] W\_PC\_out, // Output program counter

output reg [4:0] W\_RD1, // Output read data 1

output reg [4:0] W\_RD2, // Output read data 2

output reg [31:0] W\_m1, // Output memory data 1

output reg [31:0] W\_m2, // Output memory data 2

output reg [31:0] W\_ALUout // Output ALU result

);

// Internal registers and logic

reg [4:0] opcode;

reg [4:0] rd1;

reg [4:0] rd2;

reg [31:0] imm;

// Decode stage

always @ (posedge clk or posedge reset) begin

if (reset) begin

// Reset logic

W\_PC\_out <= 0;

W\_RD1 <= 0;

W\_RD2 <= 0;

W\_m1 <= 0;

W\_m2 <= 0;

W\_ALUout <= 0;

end else begin

// Fetch and decode instruction

opcode <= instruction[31:27];

rd1 <= instruction[26:22];

rd2 <= instruction[21:17];

imm <= instruction[16:0];

// Perform operations based on opcode (add more cases based on your specific instruction set)

case (opcode)

5'b00001: begin // Example: Addition operation

// Perform addition operation

W\_ALUout <= W\_RD1 + W\_RD2;

end

5'b00010: begin // Example: Subtraction operation

// Perform subtraction operation

W\_ALUout <= W\_RD1 - W\_RD2;

end

// Add more cases for different instructions

default: begin

// Handle unsupported instructions or no-op

// Optionally raise an error or do nothing

end

endcase

// Update outputs

W\_PC\_out <= W\_PC\_out + 4; // Example: Increment program counter by 4 for next instruction fetch

W\_RD1 <= rd1;

W\_RD2 <= rd2;

end

end

endmodule

2) Write testbenches to verify J\_Type\_Processor, simulate and check the simulation output data.

module tb\_J\_Type\_Processor();

// Inputs

reg reset;

reg clk;

reg [31:0] instruction;

// Outputs

wire [31:0] W\_PC\_out;

wire [4:0] W\_RD1;

wire [4:0] W\_RD2;

wire [31:0] W\_m1;

wire [31:0] W\_m2;

wire [31:0] W\_ALUout;

// Instantiate the J\_Type\_Processor module

J\_Type\_Processor dut (

.reset(reset),

.clk(clk),

.W\_PC\_out(W\_PC\_out),

.instruction(instruction),

.W\_RD1(W\_RD1),

.W\_RD2(W\_RD2),

.W\_m1(W\_m1),

.W\_m2(W\_m2),

.W\_ALUout(W\_ALUout)

);

// Clock generation

always begin

clk <= 0;

#5;

clk <= 1;

#5;

end

// Reset generation

initial begin

reset <= 1;

#10;

reset <= 0;

end

// Stimulus

initial begin

// Test case 1: Example addition instruction

instruction <= 5'b00001000010000100001000000000000; // Example assembly instruction

#50; // Allow some clock cycles for processing

// Add assertion or check for expected outputs here

// Repeat for other test cases

$stop;

end

endmodule

3) Write the assembly code with Addittion, Substraction instructions, compile into binary macchinecode and test the operation of the designed J\_Type\_Processor processor

For assembly code, let's consider a simple addition and subtraction example:

assembly

# Addition example

addi $t0, $t1, 10 # $t0 = $t1 + 10

# Subtraction example

sub $t2, $t3, $t4 # $t2 = $t3 - $t4

### III.8 EXPERIMENT NO. 8

##### III.8.1 AIM: Complet Single Cycle processor

##### Table Description automatically generated

##### Diagram, schematic Description automatically generated

**III.8.2 CODE**

module Complete Processor (reset, clk,W\_PC\_out, instruction, W\_RD1, W\_RD2,W\_m1,W\_m2,W\_ALUout);

endmodule

**III.8.3 LAB ASSIGNMENT**1) Write Verilog code to implement Complete Processor module with all following instructions:

Table

Description automatically generated

module Complete\_Processor (

input wire reset,

input wire clk,

output reg [31:0] W\_PC\_out, // Program Counter output

input wire [31:0] instruction,

output reg [31:0] W\_RD1, // Read Data 1

output reg [31:0] W\_RD2, // Read Data 2

output reg [31:0] W\_m1, // Memory Address 1

output reg [31:0] W\_m2, // Memory Address 2

output reg [31:0] W\_ALUout // ALU output

);

// Register file definitions

reg [31:0] reg\_file [31:0]; // 32 registers, each 32 bits wide

// Instruction decoding and execution logic

always @(posedge clk or posedge reset) begin

if (reset) begin

// Initialize registers, memory, etc.

// Implement reset behavior

end else begin

// Decode instruction

case (instruction[31:26])

6'b001000: begin // addi

// Extract fields from instruction

// Perform addition operation

// Update registers or memory as required

end

6'b000000: begin // add, sub

// Decode further for add and sub operations

// Perform arithmetic operations

// Update registers or memory as required

end

// Add cases for other instructions (lw, sw, bne, beq, j, etc.)

default: begin

// Handle unsupported instructions or NOP

end

endcase

end

end

// Other modules instantiation (ALU, Memory, etc.)

endmodule

2) Write testbenches to verify Complete Processor, simulate and check the simulation output data.

`timescale 1ns/1ps

module Complete\_Processor\_tb;

// Define constants or parameters

// Signals

reg reset;

reg clk;

reg [31:0] instruction;

reg [31:0] W\_PC\_out;

reg [31:0] W\_RD1;

reg [31:0] W\_RD2;

reg [31:0] W\_m1;

reg [31:0] W\_m2;

reg [31:0] W\_ALUout;

// Instantiate the Complete Processor module

Complete\_Processor dut (

.reset(reset),

.clk(clk),

.instruction(instruction),

.W\_PC\_out(W\_PC\_out),

.W\_RD1(W\_RD1),

.W\_RD2(W\_RD2),

.W\_m1(W\_m1),

.W\_m2(W\_m2),

.W\_ALUout(W\_ALUout)

);

// Clock generation or other stimulus generation

// Initial block for reset and stimulus

// Always block for clocking and monitoring

// Assertions or checks for expected outputs

endmodule

3) Compile the following code into binary machine code and store in Instruction memory to test the Complete Processor.

**Testing Assembly Program 1:**

Instruction Meaning

Begin: addi $s2, $zero, 0x55 // load immediate value 0x55 to register $s2

addi $s3, $zero, 0x22 // load immediate value 0x22 to register $s3

addi $s5, $zero, 0x77 // load immediate value 0x77 to register $s5

add $s4, $s2, $s3 // $s4 = $s2 + $s3 => R20=0x77   
sub $s1, $s2, $s3 // $s1 = $s2 – $s3 => R17=0x22   
sw $s1, 0x02($s2) // Memory[$s2+0x02] = $s1

lw $s6, 0x02($s2) // $s6 = Memory[$s2+0x02]

bne $s5, $s4, End // Next instr. is at End if $s5 != $s4

addi $s8, $zero, 0x10 // load immediate value 10 to register $s8  
beq $s5,$s4, End // Next instr. is at End if $s7 == $s4

addi $s8, $zero, 0x20 // load immediate value 20 to register $s8

End: j End // jump End

3) Compile the following code into binary machine code and store in Instruction memory to test the Complete Processor.

**Testing Assembly Program 2:**

Instruction Meaning

Begin: addi $s2, $zero, 0x55 // load immediate value 0x55 to register $s2

addi $s3, $zero, 0x22 // load immediate value 0x22 to register $s3

addi $s5, $zero, 0x77 // load immediate value 0x77 to register $s5

add $s4, $s2, $s3 // $s4 = $s2 + $s3 => R20=0x77   
sub $s1, $s2, $s3 // $s1 = $s2 – $s3 => R17=0x22   
sw $s1, 0x02($s2) // Memory[$s2+0x02] = $s1

lw $s6, 0x02($s2) // $s6 = Memory[$s2+0x02]

beq $s5,$s4, End // Next instr. is at End if $s7 == $s4

addi $s8, $zero, 0x10 // load immediate value 10 to register $s8

bne $s5, $s4, End // Next instr. is at End if $s5 != $s4

addi $s8, $zero, 0x20 // load immediate value 20 to register $s8

End: j End // jump End

4) Write the assembly code to carry following calculation formula

Sum = 1+2+3+ …..9;

Compile the following code into binary machine code and store in Instruction memory to test the Complete Processor.

**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 Assembly Testing code, 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. Analyzing the Calculation.