CPE/CSC 142
Term Project
Phase 2

Tyler Moua: 33% Contribution
Aaron Rai: 33% Contribution
Micaela Varquez: 33% Contribution

Table of Contents

CSc/CPE 142

Term Project Status Report

Team #13

Complete this form by typing the requested information and include the completed form in your report after TOC. Gray cells will be filled by the instructor.

Name	*% Contribution*	*Grade*
Tyler Moua	*33*	
Aaron Rai	*33*	
Micaela Varquez	*33*	

Please do not write in the first table

Project Report/Presentation 20%	/200
Functionality of the individual components 40%	/400
Functionality of the overall design 25%	/250
Design Approach 5%	/50
Total points	/900

A: List all the instructions that were implemented correctly and verified by the assembly program on your system:

Instructions	Was this instruction fully functional as verified by the assembly program provided? If no, explain. This refers to validation using the complete CPU and not its components.
Signed addition	Yes
Signed subtraction	Yes
Move	Yes
SWAP	Yes

Instructions	Was this instruction fully functional as verified by the assembly program provided? If no, explain. This refers to validation using the complete CPU and not its components.
AND immediate	Yes
OR immediate	Yes
Load byte unsigned	Yes
Store byte	Yes
Load	Yes
Store	Yes
Branch on less than	Yes
Branch on greater than	Yes
Branch on equal	Yes
jump	Yes
halt	Yes

B: Fill out the next table:

Individual Components	Does your system have this component?	List the student who designed and verified the block	Does it work?	List problems with the component, if any.
ALU	Yes	Tyler	Yes	
ALU control unit	Yes	Tyler	Yes	
Memory Unit	Yes	Aaron	Yes	

Individual Components	Does your system have this component?	List the student who designed and verified the block	Does it work?	List problems with the component, if any.
Register File	Yes	Aaron	Yes	
PC	Yes	Micaela	Yes	
IR	Yes	Micaela	Yes	
Other registers	No			
Multiplexors	Yes	Micaela	Yes	
exception handler 1. Unknown opcode 2. Arith. Overflow …..	Yes	Tyler Aaron Micaela	Yes	
Control Units 1. main 2. forwarding 3. lw hazard detection	Yes	Tyler Aaron Micaela	Yes	Flush occurs 1 stage later than expected. I.e. the first "stage to be flushed" is only flushed one cycle after a flush has been detected.

How many stages do you have in your pipeline? 5

C: **State any issue regarding the overall operation of the datapath? Be Specific.**

There is no sign extension in the datapath for B or C type instructions. Instead they are zero extended. This is because there was an issue with the provided instruction "LW R6, 8(R9)". You cannot represent 8 as a 4-bit signed integer. The range is from -7 to 7.

Control Logic Representation

ALU Control Unit:

The ALU Control Unit sends an ALUControl signal to the Main ALU to communicate the arithmetic needed for the instruction in EX. The inputs include the ALUOP signal sent by the Control Unit as well as the Function Code of the instruction in the WB stage of the pipeline. The following is a truth table describing the output of the ALU Control unit based on various input values.

Input		**Output**
ALUOP	FunctionCode	ALUControl
0001 (A-Type)	0000 (ADD)	000
	0001 (SUB)	001
	1110 (MOVE)	010
	1111 (SWAP)	011
1001 (AND)	n/a	100
1010 (OR)	n/a	101
0100 (Load Byte Unsigned)	n/a	000
0101 (Store Byte)	n/a	000
0110 (Load)	n/a	000
0111 (Store)	n/a	000
1100 (Branch Less Than)	n/a	000
1101 (Branch Greater Than)	n/a	000
1110 (Branch Equal)	n/a	000
0010 (Jump)	n/a	000

BC Hazard Control Unit:

The BC Hazard Control Unit is the component that checks the datapath for hazard that require a bubble to be added in the pipeline. The inputs for this unit include: the opcodes for each instruction within the pipeline (ID, EX, MEM, WB) as well as the Hazard signal. The only output of this unit is the StopPC Signal.

The following a truth table for the BC Hazard Control Unit containing input values that enable the StopPC signal. Input values that not recorded in the table will result in a StopPC value of 0.

Input				**Output**
IDOP	EXOP	MEMOP	Hazard	StopPC
0001 (A-Type)	0110 OR 0100 (LW or LBU)	n/a	x1	1
1100 (BLT) OR 1101 (BGT) OR 1110 (BE)	0001 (A-Type)	n/a	1x	1
	0110 OR 0100 (LW or LBU)	n/a	1x	1
	n/a	0110 OR 0100 (LW or LBU)	1x	1

Branch Equator:

The Branch Equator is the component that checks the values in Operand 1 and R15 to determine if a branch is valid. The inputs for this unit include the values of Operand 1 and R15 in ID, Back to Back and One Away forwarding Values, the BranchSelect, and HazardSelect signals, as well as the Hazard, Branch, and Jump signals. The only output of this component is the BranchingSoFlush Signal.

There is an internal reg "Operand" whose value is determined by the hazard and hazard select signals. The following is a truth table for determination of the value.

Input		**Reg**
Hazard	Hazard Select	Operand
0	n/a	Op1
1	001	BTB[15:0]
1	010	BTB[31:16]
1	011	OneAway[15:0]
1	100	OneAway[31:16]

The following a truth table for the Branch Equator component, containing input and internal logic that enables the BranchingSoFlush signal. Input values that not recorded in the table will result in a BranchingSoFlush value of 0.

Input	**Logic**	**Output**
Jump		BranchingSoFLush
0	Operand < R15	1
0	Operand!<R15	1
0	Operand==R15	1
1	n/a	1

Control Unit:

The Control Unit sends signals to many components of the datapath throughout each cycle of the pipeline. The inputs for this unit include: the opcodes for each instruction within the pipeline (ID, EX, MEM, WB), the function code of the instruction in WB, as well as the overflow signal. Additionally, this unit checks the opcode within ID to ensure that only predefined opcode values are used. Else, the halt signal will be enabled, stopping the pipeline.

The following are truth tables based on an instruction within a given stage of the pipeline. Note that these are processed concurrently, i.e. the Control Unit sends the signals for each instruction within their respective stages at the same time.

Input	**Output**			
OpcodeID	OffsetSelect	Branch	BranchSelect	Jump
0001 (A-Type)	00	0	0	0
1001 (AND)	01	0	0	0
1010 (OR)	01	0	0	0
0100 (Load Byte Unsigned)	00	0	0	0
0101 (Store Byte)	00	0	0	0
0110 (Load)	00	0	0	0
0111 (Store)	00	0	0	0
1100 (Branch Less Than)	01	01	00	0
1101 (Branch Greater Than)	01	01	01	0
1110 (Branch Equal)	01	01	10	0
0010 (Jump)	10	n/a	n/a	1

Input	**Output**		
OpcodeEX	ALUOP	ALUSRC1	ALUSRC2
0001 (A-Type)	OpcodeEX	00	0
1001 (AND)	OpcodeEX	01	0
1010 (OR)	OpcodeEX	01	0
0100 (Load Byte Unsigned)	OpcodeEX	00	1
0101 (Store Byte)	OpcodeEX	00	1
0110 (Load)	OpcodeEX	00	1
0111 (Store)	OpcodeEX	00	1
1100 (Branch Less Than)	OpcodeEX	10	0
1101 (Branch Greater Than)	OpcodeEX	10	0
1110 (Branch Equal)	OpcodeEX	10	0
0010 (Jump)	xxxx	0	0

Input	**Output**	
OpcodeMEM	MemRead	StoreOffset
0001 (A-Type)	0	0
1001 (AND)	0	0
1010 (OR)	0	0
0100 (Load Byte Unsigned)	1	0
0101 (Store Byte)	1	1
0110 (Load)	1	0
0111 (Store)	1	0
1100 (Branch Less Than)	0	0
1101 (Branch Greater Than)	0	0
1110 (Branch Equal)	0	0
0010 (Jump)	0	0

Input		**Output**		
OpcodeEX	FunctionCode	MemToReg	RegWrite	WriteOp2
0001 (A-Type)	1111 (Swap)	00	1	1
	!(1111) (Not Swap)	00	1	0
1001 (AND)	n/a	00	1	0
1010 (OR)	n/a	00	1	0
0100 (Load Byte Unsigned)	n/a	01	1	0
0101 (Store Byte)	n/a	0	0	0
0110 (Load)	n/a	01	1	0
0111 (Store)	n/a	0	0	0
1100 (Branch Less Than)	n/a	0	0	0
1101 (Branch Greater Than)	n/a	0	0	0
1110 (Branch Equal)	n/a	0	0	0
0010 (Jump)	n/a	0	0	0

Register Forwarding Unit:

The Register Forwarding Unit is a component that detects hazards based on the operands of various instructions within the pipeline and outputs the hazard signal as well as various MUX selection signals. The inputs of this unit include: The first operand of the instruction in ID as well as both operands of each instruction within the other stages of the pipeline (EX, MEM, WB).

The following are truth tables for the Register Forwarding Unit containing input values that enable the HazardDetected signal. Input values that not recorded in the table will result in a HazardDetected value of 0. Additionally, each truth table in broken between different MUX selection signals. When there is no hazard detected, the selection signal for the MUXes are set to 0.

NOTE: Mux4 was removed from the final pipeline, but the signal "ForwardToMux4" is used by the BranchEquator component for forwarding.

Input	**Output**	
EXOP2	ForwardToMux3	HazardDetected
MEMOP1	001	1
WBOP1	011	1
MEMOP2	010	1
WBOP2	100	1

Input	**Output**	
IDOP	ForwardToMux4	HazardDetected
EXOP1	001	1
MEMOP1	011	1
EXOP2	010	1
MEMOP2	100	1

Input	**Output**	
EXOP1	ForwardToMux5	HazardDetected
MEMOP1	001	1
WBOP1	011	1
MEMOP2	010	1
WBOP2	100	1

Input	**Output**	
BTBOP1	ForwardToMux6	HazardDetected
OAOP1	001	1

Verilog Source Code

ALUControlUnit

```
module ALUControlUnit (input [3:0] ALUOP, FunctionCode,
                                          output reg [2:0] ALUControl);
always @(*)
begin
        case (ALUOP)
                //A-TYPE
                4'b0001:
                begin
                        case (FunctionCode)
                                //ADD
                                4'b0000: ALUControl = 000;
                                //SUB
                                4'b0001: ALUControl = 001;
                                //MOVE
                                4'b1110: ALUControl = 010;
                                //SWAP
                                4'b1111: ALUControl = 011;
                                default ALUControl = 011;
                        endcase
                end
                //AND
                4'b1001: ALUControl = 100;
                //OR
                4'b1010: ALUControl = 101;
                //All other cases require adding. Branching not handled here.
                default: ALUControl = 000;
        endcase
endendmodule
```

BCHazardControlUnit

```
module BCHazardControlUnit(input [3:0] IDOP, EXOP, MEMOP, WBOP,
                                    input [1:0]Hazard,
                                    output reg StopPC);
always @(*)
begin
        StopPC = 00;
        //if we have an A type hazard:
        if(Hazard[0])
        begin
                //if we have an A-type in ID:
                if(IDOP == 4'b0001)
                begin
                        //if we have a LW in EX:
                        if((EXOP == 4'b0110)||(EXOP == 4'b0100))
                                StopPC=01;
                end
        end
        if(Hazard[1])
        begin
                //If we have a branch in ID:
                if((IDOP == 4'b1100)||(IDOP == 4'b1101)||(IDOP == 4'b1110))
                begin

                        //if we have a A-type in EX:
                        if(IDOP == 4'b0001)
                        begin
                                StopPC= 01;
                        end
                        //If we have a LW in MEM
                        else if((MEMOP == 4'b0110)||(MEMOP == 4'b0100))
                        begin
                                StopPC=01;
                        end
                        //If we have a LW in EX
                        else if((EXOP == 4'b0110)||(EXOP == 4'b0100))
                        begin
                                StopPC=01;
                        end
                end
        end
end
endmodule
```

BranchEquator

```
module BranchEquator(input [15:0] Op1,R15,
                     input [31:0] BTB, OneAway,
                     input [1:0]   BranchSelect,
                     input [2:0] HazardSelect,
                     input Hazard, Branch, Jump,
                     output reg BranchingSoFlush);
reg Negative, Zero;
reg [15:0] Operand;
always @(*)
begin
        Negative = 1'b0;
        Zero = 1'b0;
        BranchingSoFlush=1'b0;
        Operand = Op1;
        if(Hazard)
        begin
                case(HazardSelect)
                3'b001: Operand = BTB [15:0];
                3'b010: Operand = BTB [31:16];
                3'b011: Operand = OneAway [15:0];
                3'b100: Operand = OneAway [31:16];
                endcase
        end
        if(Operand < R15)
                Negative = 1'b1;
        if (Operand == R15)
                Zero = 1'b1;
        case(BranchSelect)
                //BLT
                2'b00: BranchingSoFlush = (Negative & Branch) | Jump;
                //BGT
                2'b01: BranchingSoFlush = (!Negative & !Zero & Branch) | Jump;
                //BEQ, covers 2'b10, 2'b11
                2'b10: BranchingSoFlush = (Zero & Branch) | Jump;
                2'b11: BranchingSoFlush = (Zero & Branch) | Jump;
                default: BranchingSoFlush = 0;
        endcase

end
endmodule
```

ControlUnit

```
module ControlUnit(input [3:0] OpcodeID, OpcodeEX, OpcodeMEM, OpcodeWB,
                   input [3:0] FunctionCode,
                   input Overflow,
                   output reg RegWrite, Branch, Jump, Halt, WriteOP2, MemRead,
                   output reg MemWrite, StoreOffset, ALUSRC2,
                   output reg [1:0] MemToReg, OffsetSelect, BranchSelect, ALUSRC1,
                   output reg [3:0] ALUOP);
always @(*)
begin
//Default all Signals to 0
//Signals will be turned on when needed.
        WriteOP2=0;
        RegWrite=0;
        Branch=0;
        Jump=0;
        Halt=0;
        WriteOP2=0;
        MemRead=0;
        ALUSRC1=0;
        ALUSRC2=0;
        MemToReg=00;
        MemWrite=0;
        OffsetSelect=00;
        StoreOffset=0;
        BranchSelect=0;
        //Instructions in ID
        if(((OpcodeID!= 4'b0001)&&(OpcodeID!= 4'b1001)&&
          (OpcodeID!= 4'b1010)&&(OpcodeID!= 4'b0100)&&
          (OpcodeID!= 4'b0101)&&(OpcodeID!= 4'b0110)&&
          (OpcodeID!= 4'b0111)&&(OpcodeID!= 4'b1100)&&
          (OpcodeID!= 4'b1101)&&(OpcodeID!= 4'b1110)&&
          (OpcodeID!= 4'b0010))||Overflow)
          begin
                Halt = 1;
          end
        case (OpcodeID)
                //AND
                4'b1001:
                begin
                        OffsetSelect=01;
                end
                //OR
                4'b1010:
                begin
                        OffsetSelect=01;
                end
                //BLT
                4'b1100:
                begin
                        Branch=1;
                        OffsetSelect=01;
                end
                //BGT
                4'b1101:
                begin
                        Branch=1;
                        OffsetSelect=01;
                        BranchSelect=01;
                end
                //BEQ
                4'b1110:
                begin
                        Branch=1;
                        OffsetSelect=01;
                        BranchSelect=10;
                end
                //Jump
                4'b0010:
                begin
                        Jump = 1;
                        OffsetSelect = 10;
                end
        endcase

        //Instructions in EX
        case (OpcodeEX)
                //A-TYPE
                4'b0001:
                begin
                        ALUOP = OpcodeEX;
                end
                //AND
                4'b1001:
                begin
                        ALUOP = OpcodeEX;
                        ALUSRC1 =01;
                end
                //OR
                4'b1010:
                begin
```

```
                ALUOP = OpcodeEX;
                ALUSRC1 =01;
        end
        //Load Byte Unsigned
        4'b0100:
        begin
                ALUOP = OpcodeEX;
                ALUSRC2 =1;
        end
        //Store Byte
        4'b0101:
        begin
                ALUOP = OpcodeEX;
                ALUSRC2 =1;
        end
        //Load
        4'b0110:
        begin
                ALUOP = OpcodeEX;
                ALUSRC2 =1;
        end
        //Store
        4'b0111:
        begin
                ALUOP = OpcodeEX;
                ALUSRC2 =1;
        end
        //BLT
        4'b1100:
        begin
                ALUOP = OpcodeEX;
                ALUSRC1 =10;
        end
        //BGT
        4'b1101:
        begin
                ALUOP = OpcodeEX;
                ALUSRC1 =10;
        end
        //BEQ
        4'b1110:
        begin
                ALUOP = OpcodeEX;
                ALUSRC1 =10;
        end
endcase
```

```
//Instructions in MEM
case (OpcodeMEM)
	//Load Byte Unsigned
	4'b0100:
	begin
		MemRead = 1;
	end
	//Store Byte
	4'b0101:
	begin
		MemWrite = 1;
		StoreOffset = 1;
	end
	//Load
	4'b0110:
	begin
		MemRead = 1;
	end
	//Store
	4'b0111:
	begin
		MemWrite = 1;
	end
endcase
//Instructions in WB
case (OpcodeWB)
	//A-TYPE
	4'b0001:
	begin
		MemToReg = 00;
		RegWrite = 1;
		if(FunctionCode == 4'b1111)
		 WriteOP2=1;
	end
	//AND
	4'b1001:
	begin
		MemToReg = 00;
		RegWrite = 1;
	end
	//OR
	4'b1010:
	begin
		MemToReg = 00;
		RegWrite = 1;
```

```
                end
                //Load Byte Unsigned
                4'b0100:
                begin
                        MemToReg = 10;
                        RegWrite = 1;
                end
                //Load
                4'b0110:
                begin
                        MemToReg = 01;
                        RegWrite = 1;
                end
        endcase
end
endmodule
```

CPU

```
`include "RegisterFile.v";
`include "InstructionMemory.v";
`include "DataMemory.v";
`include "MainALU.v";
`include "PC.v";
`include "ControlUnit.v";
`include "ALUControlUnit.v";
`include "IFID.v";
`include "IDEX.v";
`include "EXMEM.v";
`include "MEMWB.v";
`include "MUX1.v";
`include "MUX2.v";
`include "MUX3.v";
`include "MUX5.v";
`include "MUX6.v";
`include "MUX7.v";
`include "BranchEquator.v"
`include "SignExtendID.v"
`include "SignExtendWB.v"
`include "SignExtendMEM.v"
`include "ZeroExtend.v"
`include "ShiftLeft.v"
`include "RegisterForwardingUnit.v"
`include "BCHazardControlUnit.v"

module CPU (input clk, rst);
//Data Wires
wire [31:0] JBPC, ALUResultEX, ALUResultMEM, ALUResultWB, ResultWB, NewPC;
wire [31:0] SEData, SEByte, BTBForward, OneAwayForward;
wire [15:0] PCMUXResult, InstructionIF,InstructionID, InstructionEX;
wire [15:0] InstructionMEM, InstructionWB, SEImmdID, SEImmdEX,Op1ToStore;
wire [15:0] PCALU2OP, PCALU2OPShifted,SEOp1,Op1ToStore1;
wire [15:0] ReadDataMEM, ReadDataWB, PCOut,OP1ID, OP2ID, OP1EX, OP2EX;
wire [15:0] OP1MEM, R15ID, R15EX, PCToAdd, Four, Eight, Twelve;
wire [15:0]  M3Result, M5Result, ReadDataExtended;

//Control Signals
wire ForwardToMux6, StayHalted, StopPC, Overflow, Branch, Jump, Halt, WriteOP2,
RegWrite,  ALUSRC2;
wire MemRead, MemWrite, StoreOffset, BranchingSoFlush, BranchingSoFlushEX;
wire [3:0] ALUOPID,  ALUOPEX;
wire [2:0] ALUControl,   ForwardToMux4, ForwardToMux3, ForwardToMux5;
wire [1:0] Hazard, MemToReg, ALUSRC1,  OffsetSelect, BranchSelect;
```

```
//DataPath:

//IF:
MUX1 M1(.A(NewPC[15:0]), .B(JBPC[15:0]), .BranchingSoFlush(BranchingSoFlush),
        .Result(PCMUXResult));

PC ProgramCounter(.NewPC(PCMUXResult), .clk(clk), .rst(rst),
                  .Halt(Halt), .StayHalted(StayHalted), .StopPC(StopPC), .PC(PCOut));

MainALU PCALU1(.Op1(PCOut), .Op2(16'h0002), .ALUControl(3'b000), .Result(NewPC));

InstructionMemory IM(.ReadAddress(PCOut), .clk(clk),.rst(rst),
                     .Instruction(InstructionIF));

IFID IFID(.PCIN(NewPC[15:0]),.InstructionIn(InstructionIF), .clk(clk), .rst(rst), .Halt(Halt),
       .PCOUT(PCToAdd), .InstructionOut(InstructionID), .FlushIn(BranchingSoFlush),
       .FlushOut(BranchingSoFlushEX), .StopPC(StopPC), .StayHalted(StayHalted),
       .OldInstruction(InstructionID));

//ID:
RegisterFile RF(.ReadReg1(InstructionID[11:8]), .ReadReg2(InstructionID[7:4]),
           .WriteReg1(InstructionWB[11:8]), .WriteReg2(InstructionWB[7:4]),
           .WriteData1(ResultWB[15:0]), .WriteData2(ResultWB[31:16]),
           .clk(clk), .rst(rst), .RegWrite(RegWrite), .WriteOP2(WriteOP2),
           .ReadData1(OP1ID), .ReadData2(OP2ID), .R15(R15ID));

ControlUnit CU(.OpcodeID(InstructionID[15:12]),.OpcodeEX(InstructionEX[15:12]),
           .OpcodeMEM(InstructionMEM[15:12]), .OpcodeWB(InstructionWB[15:12]),
           .FunctionCode(InstructionWB[3:0]),.Overflow(Overflow),
           .OffsetSelect(OffsetSelect), .MemToReg(MemToReg),
           .StoreOffset(StoreOffset), .MemRead(MemRead),
           .ALUSRC1(ALUSRC1), .ALUSRC2(ALUSRC2), .MemWrite(MemWrite),
           .BranchSelect(BranchSelect),.RegWrite(RegWrite), .ALUOP(ALUOPID),
           .Branch(Branch), .Jump(Jump), .Halt(Halt), .WriteOP2(WriteOP2));

SignExtendID SEID1(.a(InstructionID[3:0]), .b(InstructionID[11:0]),
              .ResultA(Four), .ResultB(Twelve));

ZeroExtend ZEID(.a(InstructionID[7:0]), .Result(Eight));

MUX2       M2(.four(Four),.eight(Eight),.twelve(Twelve),
                .offsetSelect(OffsetSelect),.Result(SEImmdID));

ShiftLeft SL(.a(SEImmdID), .Result(PCALU2OPShifted));
```

```
MainALU PCALU2(.Op1(PCToAdd), .Op2(PCALU2OPShifted), .ALUControl(3'b000),
                    .Result(JBPC));

BranchEquator BE(.Op1(OP1ID),.Hazard(Hazard[1]), .R15(R15ID),
                  .BranchSelect(BranchSelect), .BTB(BTBForward),
                  .OneAway(OneAwayForward), .HazardSelect(ForwardToMux4),
                  .Branch(Branch), .Jump(Jump), .BranchingSoFlush(BranchingSoFlush));

IDEX   IDEX(.InstructionIn(InstructionID), .OP1In(OP1ID),.OP2In(OP2ID), .clk(clk),
             .StopPC(StopPC), .rst(rst), .ALUOPIn(ALUOPID), .ALUOPOut(ALUOPEX),
             .OP1Out(OP1EX), .SEImmdIn(SEImmdID),.OP2Out(OP2EX),
             .InstructionOut(InstructionEX), .SEImmdOut(SEImmdEX), .R15In(R15ID),
             .R15Out(R15EX),.flush(BranchingSoFlushEX));
//EX:
MUX3 M3(.SEIMMD(SEImmdEX), .Op2(OP2EX), .Btb(BTBForward),
          .oneAway(OneAwayForward), .R15(R15EX), .hazard(Hazard[0]),
          .ALUSRC(ALUSRC1), .ForwardToMux3(ForwardToMux3),
          .Result(M3Result));

MUX5 M5(.SEIMMD(SEImmdEX), .Op1(OP1EX), .Btb(BTBForward),
          .oneAway(OneAwayForward), .hazard(Hazard[0]), .ALUSRC(ALUSRC2),
          .ForwardToMux5(ForwardToMux5), .Result(M5Result));

ALUControlUnit ACU(.ALUOP(ALUOPID), .FunctionCode(InstructionEX[3:0]),
                      .ALUControl(ALUControl));

MainALU MALU(.Op1(M5Result), .Op2(M3Result), .ALUControl(ALUControl),
                  .Overflow(Overflow), .Result(ALUResultEX));

EXMEM EXMEM(.InstructionIn(InstructionEX), .OP1In(OP1EX), .OP2In(OP2EX),
                  .ALUResultIn(ALUResultEX), .clk(clk), .rst(rst),
                  .ALUResultOut(ALUResultMEM), .InstructionOut(InstructionMEM),
                  .OP1Out(OP1MEM),.BTBForward(BTBForward));

RegisterForwardingUnit RFU(.IDOP1(InstructionID[11:8]),
                             .ForwardToMux6(ForwardToMux6), .OP1(InstructionEX[11:8]),
                             .OP2(InstructionEX[7:4]), .BTBOP1(InstructionMEM[11:8]),
                             .BTBOP2(InstructionMEM[7:4]), .OAOP1(InstructionWB[11:8]),
                             .OAOP2(InstructionWB[7:4]),
                             .ForwardToMux3(ForwardToMux3),
                             .ForwardToMux4(ForwardToMux4),
                             .ForwardToMux5(ForwardToMux5),
                             .HazardDetected(Hazard),
                             .OpcodeEX(InstructionEX[15:12]),
                             .OpcodeMEM(InstructionMEM[15:12]),
                             .FunctionCodeMEM(InstructionMEM[3:0]),
                                        .OpcodeWB(InstructionWB[15:12]),
                                        .FunctionCodeWB(InstructionWB[3:0]));

BCHazardControlUnit BCHCU(.IDOP(InstructionID[15:12]), .EXOP(InstructionEX[15:12]),
                          .MEMOP(InstructionMEM[15:12]),
                          .WBOP(InstructionWB[15:12]), .Hazard(Hazard),
                          .StopPC(StopPC));

//MEM:
SignExtendMEM SEMEM(.a(OP1MEM[7:0]), .Result(SEOp1));

MUX6 M6(.A(OP1MEM), .B(SEOp1), .ForwardValue(ResultWB[15:0]),
        .StoreOffset(StoreOffset), .Forward(ForwardToMux6), .Result(Op1ToStore));

DataMemory DM(.Address(ALUResultMEM[15:0]), .WriteData(Op1ToStore),
              .StoreOffset(StoreOffset), .clk(clk), .rst(rst), .memWrite(MemWrite),
              .ReadData(ReadDataMEM), .WriteByte(OP1MEM[7:0]));

MEMWB MEMWB(.InstructionIn(InstructionMEM), .ReadDataIn(ReadDataMEM),
            .ALUResultIn(ALUResultMEM), .clk(clk), .rst(rst),
            .OneAwayForward(OneAwayForward), .OP1In(Op1ToStore),
            .ReadDataOut(ReadDataWB), .InstructionOut(InstructionWB),
            .ALUResultOut(ALUResultWB));

//WB:
ZeroExtend ZEWB(.a(ReadDataWB[7:0]), .Result(ReadDataExtended));

SignExtendWB SEWB(.a(ReadDataExtended), .b(ReadDataWB), .ResultA(SEByte),
                  .ResultB(SEData));

MUX7 M7(.alu(ALUResultWB), .eight(SEByte), .sixteen(SEData), .memToReg(MemToReg),
        .Result(ResultWB));

endmodule
```

DataMemory

```
module DataMemory #(parameter N = 100)
                            (input [15:0]Address, WriteData,
                              input [7:0] WriteByte,
                              input clk, rst, memWrite, StoreOffset,
                              output [15:0] ReadData);

reg [7:0] Data [N-1:0];
integer i;
assign ReadData = {Data [Address],Data [Address+1]};
always @(posedge clk, negedge rst)
begin
        if(!rst)
        begin
                for (i = 0; i<16; i=i+1)
                        Data[i]<=0;
                Data[0]<=8'h3c;
                Data[1]<=8'hAD;

                Data[2]<=8'h00;
                Data[3]<=8'h00;

                Data[4]<=8'h14;
                Data[5]<=8'h63;

                Data[6]<=8'hDA;
                Data[7]<=8'hED;

                Data[8]<=8'hFE;
                Data[9]<=8'hEB;
                //Data[A]
                Data[10]<=8'hFF;
                Data[11]<=8'hFF;
                //Data[E]
                Data[14]<=8'hCC;
                Data[15]<=8'hCC;
        end
        if(memWrite)
        begin
                if(StoreOffset==1)
                begin
                        Data[Address+1] <=WriteByte;
                end
                else
                begin
                Data [Address] <= WriteData[15:8];
                Data [Address+1] <= WriteData[7:0];
            end
        end
    end
    endmodule
```

EXMEM

```
module EXMEM(input [15:0]InstructionIn, OP1In, OP2In,
                        input [31:0] ALUResultIn,
                        input clk, rst,
                        output reg [31:0] ALUResultOut, BTBForward,
                        output reg [15:0] InstructionOut, OP1Out);

always @(posedge clk, negedge rst)
begin

        if(!rst)
        begin

        end
        else
        begin
                OP1Out<=OP1In;
                ALUResultOut<=ALUResultIn;
                InstructionOut<=InstructionIn;
                BTBForward <= ALUResultIn;
        end
end
endmodule
```

IDEX

```
module IDEX(input [15:0]OP1In, OP2In, InstructionIn, SEImmdIn, R15In,
                        input [3:0] ALUOPIn,
                        input clk, rst, flush, StopPC,
                        output reg [3:0] ALUOPOut,
                        output reg [15:0] OP1Out, OP2Out, InstructionOut,SEImmdOut, R15Out);

always @(posedge clk, negedge rst)
begin

        if(!rst)
        begin
        end
        else if ((flush)||(StopPC==1'b1))
        begin
                InstructionOut <= 16'hxxxx;
        end
        else
        begin
                SEImmdOut <=SEImmdIn;
                OP1Out<=OP1In;
                OP2Out<=OP2In;
                ALUOPOut<=ALUOPIn;
                InstructionOut<=InstructionIn;
                R15Out <= R15In;
        end
end
endmodule
```

IFID

```
module IFID(input [15:0]PCIN, InstructionIn, OldInstruction,
                        input clk, rst, FlushIn, Halt, StopPC, StayHalted,
                        output reg FlushOut,
                        output reg [15:0] PCOUT,  InstructionOut);

always @(posedge clk, negedge rst)
begin

        if(!rst)
        begin

        end
        else if(Halt||StayHalted)
        begin
                InstructionOut<=16'hxxxx;
        end
        else
        if(StopPC)
        begin
        InstructionOut<=OldInstruction;
        end
        else
        begin
                InstructionOut<=InstructionIn;
                PCOUT  <= PCIN;
                FlushOut <=FlushIn;
        end
end
endmodule
```

InstructionMemory

```
module InstructionMemory #(parameter N = 100)
                                        (input [15:0]ReadAddress,
                                         input clk, rst,
                                         output [15:0] Instruction);
reg [7:0] Instructions [N-1:0];
integer i;
assign Instruction = {Instructions[ReadAddress],Instructions[ReadAddress+1]};
always @(posedge clk, negedge rst)
begin
        if(!rst)
        begin
                for (i = 0; i<16; i=i+1)
                        Instructions[i]<=0;
                //Add R1, R2 <= 1120
                Instructions[0] <= 8'h11;
                Instructions[1] <= 8'h20;
                //Sub R2,  R13<= 12D1
                Instructions[2] <= 8'h12;
                Instructions[3] <= 8'hD1;
                //MOV R4, R8 <= 148E
                Instructions[4] <= 8'h14;
                Instructions[5] <= 8'h8E;
                //OR R8, 0000<= A800
                Instructions[6] <= 8'hA8;
                Instructions[7] <= 8'h00;
                //SWP R4,  R6 <= 146F
                Instructions[8] <= 8'h14;
                Instructions[9] <= 8'h6F;
                //[A]:LBU R7,  4(R9)=4794
                Instructions[10] <= 8'h47;
                Instructions[11] <= 8'h94;
                //[C]:ANDi R3,  4C <=934c
                Instructions[12] <= 8'h93;
                Instructions[13] <= 8'h4c;
                //[E]:SUB R14, R14<=1EE1
                Instructions[14] <= 8'h1e;
                Instructions[15] <= 8'he1;
                //[10]:SB R7,  6(R9) <=5796
                Instructions[16] <= 8'h57;
                Instructions[17] <= 8'h96;
                //[12]:LW R6, 8(R9)=6698
                Instructions[18] <= 8'h66;
                Instructions[19] <= 8'h98;
                //[14]:BEQ R7, 4 <= E704
```

```
                    Instructions[20] <= 8'hE7;
                    Instructions[21] <= 8'h04;
                    //[16]:ADD R11, R1 <=1b10
                    Instructions[22] <= 8'h1b;
                    Instructions[23] <= 8'h10;
                    //[18] BLT R7, 5 <= C705
                    Instructions[24] <= 8'hc7;
                    Instructions[25] <= 8'h05;
                    //[1A] ADD R11, R2<=1b20
                    Instructions[26] <= 8'h1b;
                    Instructions[27] <= 8'h20;
                    //[1C] BGT R7, 2 <=d702
                    Instructions[28] <= 8'hd7;
                    Instructions[29] <= 8'h02;
                    //[1E] ADD R1, R1 <= 1110
                    Instructions[30] <= 8'h11;
                    Instructions[31] <= 8'h10;
                    //[20] ADD R1, R1 <= 1110
                    Instructions[32] <= 8'h11;
                    Instructions[33] <= 8'h10;
                    //[22] LW R8, 0(R9) <= 6890
                    Instructions[34] <= 8'h68;
                    Instructions[35] <= 8'h90;
                    //[24] ADD R8, R8 <= 1880
                    Instructions[36] <= 8'h18;
                    Instructions[37] <= 8'h80;
                    //[26] SW R8, 2(R9)= 7892
                    Instructions[38] <= 8'h78;
                    Instructions[39] <= 8'h92;
                    //[28] LW R10, 2(R9)= 6a92
                    Instructions[40] <= 8'h6a;
                    Instructions[41] <= 8'h92;
                    //[2A] ADD R12, R10 <=1ca0
                    Instructions[42] <= 8'h1c;
                    Instructions[43] <= 8'ha0;
                    //[2C] SUB R12, R13 <= 1cd1
                    Instructions[44] <= 8'h1c;
                    Instructions[45] <= 8'hd1;
                    //[2E] ADD R12, R13 <= 1cd0
                    Instructions[46] <= 8'h1c;
                    Instructions[47] <= 8'hd0;
                    Instructions[48] <= 8'h0f; //Exception
                    Instructions[49] <= 8'h20;
          end
end
endmodule
```

MainALU

```
module MainALU(input signed [15:0]Op1, Op2,
                        input [2:0] ALUControl,
                        output reg Overflow,
                        output reg signed [31:0] Result);

reg signed [16:0] Result1;
reg signed [15:0] Result2;

always @(*)
begin
        //Default:
        Overflow = 1'b0;
        case (ALUControl)
                //ADD
                3'b000:
                begin
                        Result1 = Op1 + Op2;
                        //Ex 1 1000 vs 1 0000 vs 0 1101, becauase its signed
                        if(Result1[16]!=Result1[15])
                                Overflow = 1;
                end
                //SUB
                3'b001:
                begin
                        Result1 = Op1 - Op2;
                        Overflow = Result1[16];
                end
                //MOVE
                3'b010: Result1 = Op2;
                //SWAP
                3'b011:
                begin
                        Result1 = Op2;
                        Result2 = Op1;
                end
                //AND
                3'b100: Result1 = Op1 & Op2;
                //OR - 101, 110, or 111
                3'b101: Result1 = Op1 | Op2;
                default: Result1 = Op1 | Op2;
        endcase
        Result={Result2, Result1[15:0]};
end
endmodule
```

MEMWB

```
module MEMWB(input [15:0]InstructionIn, ReadDataIn, OP1In,
                        input [31:0] ALUResultIn,
                        input clk, rst,
                        output reg [31:0] ALUResultOut, OneAwayForward,
                        output reg [15:0] ReadDataOut,InstructionOut);

always @(posedge clk, negedge rst)
begin

        if(!rst)
        begin

        end
        else
        begin
                ALUResultOut= ALUResultIn;
                ReadDataOut<=ReadDataIn;
                InstructionOut<=InstructionIn;

        end
end

//Forwarding:
always @(*)
begin
//If we have a load word:
                if((InstructionIn[15:12] == 4'b0110)||(InstructionIn[15:12]  == 4'b0100))
                        OneAwayForward<=ReadDataIn;
                //if we have a store word:
                else if((InstructionIn[15:12] == 4'b0101)||(InstructionIn[15:12]  == 4'b0111))
                        OneAwayForward<={16'h0000,OP1In};
                else
                        OneAwayForward<=ALUResultIn;
end
endmodule
```

MUX1-7

Note: There is no MUX4, as it was removed in the final design.

```
module MUX1(input [15:0]A,B,
                    input BranchingSoFlush,
                    output reg [15:0] Result);
always @(*)
begin
          Result =  A;
          if(BranchingSoFlush)
                    Result = B;

end
endmodule

module MUX2(input [15:0] four, eight, twelve,
                    input [1:0] offsetSelect,
                    output reg [15:0] Result);
always @(*)
begin
          case (offsetSelect)
                    //ALU Result
                    2'b00: Result = four;
                    //16 bit sign extended
                    2'b01: Result = eight;
                    //8 bit zero extended : 10, 11
                    default: Result = twelve;
          endcase
end
endmodule
```

```
module MUX3(input [31:0] Btb, oneAway,
                        input [15:0] SEIMMD, Op2, R15,
                        input hazard,
                        input [1:0] ALUSRC,
                        input [2:0] ForwardToMux3,
                        output reg [15:0] Result);
always @(*)
begin
        case (ALUSRC)
                2'b00: Result = Op2;
                2'b01: Result = SEIMMD;
                2'b10: Result = R15;
        endcase
        if(hazard)
        begin
                case (ForwardToMux3)
                3'b001: Result = Btb [15:0];
                3'b010: Result = Btb [31:16];
                3'b011: Result = oneAway [15:0];
                3'b100: Result = oneAway [31:16];
                endcase
        end
end
endmodule
module MUX5(input [31:0] Btb, oneAway,
                        input [15:0] SEIMMD, Op1,
                        input hazard, ALUSRC,
                        input [2:0] ForwardToMux5,
                        output reg [15:0] Result);
always @(*)
begin
        case (ALUSRC)
                1'b0: Result = Op1;
                1'b1: Result = SEIMMD;
        endcase
        if(hazard)
        begin
                case (ForwardToMux5)
                3'b001: Result = Btb [15:0];
                3'b010: Result = Btb [31:16];
                3'b011: Result = oneAway [15:0];
                3'b100: Result = oneAway [31:16];
        endcase
        end
end
endmodulemodule
```

```
MUX6(input [15:0]A,B, ForwardValue,
           input StoreOffset, Forward,
           output reg [15:0] Result);

always @(*)
begin
          if(Forward)
          begin
                    Result = ForwardValue;
          end
          else
          begin
          Result =  A;
          if(StoreOffset)
                    Result = B;
          end
end
endmodule

module MUX7(input [31:0] alu, eight, sixteen,
                     input [1:0] memToReg,
                     output reg [31:0] Result);
always @(*)
begin
          case (memToReg)
                    //ALU Result
                    2'b00: Result = alu;
                    //16 bit sign extended
                    2'b01: Result = sixteen;
                    //8 bit zero extended : 10, 11
                    default: Result = eight;
          endcase
end
endmodule
```

PC

```
module PC(input [15:0] NewPC,
          input clk, rst, Halt, StopPC,
          output reg StayHalted,
          output reg [15:0] PC);

always @(posedge clk, negedge rst)
begin
	if(!rst)
	begin
		PC <= 0;
		StayHalted <= 0;
	end
	else
	begin
		//If we have a halt, do nothing
		if(Halt || StayHalted)
		begin
			StayHalted = 1;
		end
		else
		begin
			if(StopPC ==1'b1)
			begin
				PC <= NewPC-16'h0002;
			end
			else
			begin
				//Output expected with no hazards in execution.
				PC <= NewPC;
			end
		end
	end
end
endmodule
```

RegisterFile

```
module RegisterFile(input [3:0] ReadReg1,ReadReg2,WriteReg1,WriteReg2,
                            input [15:0]  WriteData1, WriteData2,
                            input clk, rst, RegWrite, WriteOP2,
                            output reg [15:0] ReadData1, ReadData2, R15);
reg [15:0] Registers [15:0];
integer i;
always@(*)
begin
        ReadData1 = Registers [ReadReg1];
        ReadData2 = Registers [ReadReg2];
        R15 = Registers [15];
        if(WriteReg1 == ReadReg1)
                ReadData1 = WriteData1;
end
always @(posedge clk, negedge rst)
begin
        if(!rst)
        begin
                for (i = 0; i<16; i=i+1)
                        Registers[i]<=0;
                Registers[0] <= 16'h0000;
                Registers[1] <= 16'h0e12;
                Registers[2] <= 16'h0045;
                Registers[3] <= 16'hF08F;
                Registers[4] <= 16'hF076;
                Registers[5] <= 16'h0084;
                Registers[6] <= 16'h6789;
                Registers[7] <= 16'h00EB;
                Registers[8] <= 16'hFF56;
                Registers[12] <= 16'hCC89;
                Registers[13] <= 16'h0002;
        end
        else
        begin
                if(RegWrite)
                begin
                        Registers [WriteReg1] <= WriteData1;
                        if(WriteOP2)
                                Registers [WriteReg2] <= WriteData2;
                end
        end
end
endmodule
```

RegisterForwardingUnit

```
module RegisterForwardingUnit(input [3:0] OP1, OP2, BTBOP1, BTBOP2, OAOP1, OAOP2,
                              input [3:0] IDOP1, OpcodeEX, OpcodeMEM, OpcodeWB,
                              input [3:0] FunctionCodeMEM, FunctionCodeWB
                              output reg [2:0] ForwardToMux3,ForwardToMux4,
                              output reg [2:0] ForwardToMux5,
                              output reg [1:0]HazardDetected,
                              output reg ForwardToMux6);

always @(*)
begin
        HazardDetected=00;
        ForwardToMux3=000;
        ForwardToMux4=000;
        ForwardToMux5=000;
        ForwardToMux6=0;

//Branch Hazards:
//IDOP1 deals with Mux4 Hazard[1] for branch hazards
        if(IDOP1 == OP1)
        begin
                ForwardToMux4 = 001;
                HazardDetected[1]=1;
        end
        else if(IDOP1 == BTBOP1)
        begin
                ForwardToMux4 = 011;
                HazardDetected[1]=1;
        end

        //Test for OP2 IF Swap(Only time that OP2 changes)
        if((OpcodeMEM == 0001)&& (FunctionCodeMEM ==1111))
        begin
                if(IDOP1 == OP2)
                begin
                        ForwardToMux4 = 010;
                        HazardDetected=1;
                end
        end
        else if((OpcodeWB == 0001)&& (FunctionCodeWB ==1111))
        if(IDOP1 == BTBOP2)
        begin
                ForwardToMux4 = 100;
                HazardDetected=1;
        end
```

```
//OTHER:
if(BTBOP1 == OAOP1)
        if((OpcodeMEM == 4'b0101)||(OpcodeMEM == 4'b0111))
        begin
                ForwardToMux6 = 1;
        end
//OP1 Deals with Mux5
        if(OP1 == BTBOP1)
        begin
                //if load in mem
                if((OpcodeMEM == 4'b0110)||(OpcodeMEM == 4'b0100))
                        ForwardToMux5 = 011;
                //We don't forward to the operand of a SW in EX
                else if((OpcodeEX == 4'b0101)||(OpcodeEX == 4'b0111))
                begin
                        //Do nothing
                end
                else
                        ForwardToMux5 = 001;
                HazardDetected[0]=1;
        end
        else if(OP1 == OAOP1)
        begin
                ForwardToMux5 = 011;
                HazardDetected[0]=1;
        end

        //Test for OP2 IF Swap(Only time that OP2 changes)
        if((OpcodeMEM == 0001)&& (FunctionCodeMEM ==1111))
        begin
                if(OP1 == BTBOP2)
                begin
                //if load
                if((OpcodeMEM == 4'b0110)||(OpcodeMEM == 4'b0100))
                        ForwardToMux5 = 011;
                else
                        ForwardToMux5 = 001;
                        HazardDetected[0]=1;
                end
        end
        else if((OpcodeWB == 0001)&& (FunctionCodeWB ==1111))
        if(OP1 == OAOP2)
        begin
                ForwardToMux5 = 100;
                HazardDetected[0]=1;
        end

//OP2 Deals with Mux3 and ONLY with A or B Type instructions.
        if((OpcodeEX==0001) || (OpcodeEX== 0100)|| (OpcodeEX== 0101)||
                (OpcodeEX==0110) || (OpcodeEX== 0111))
        begin
                if(OP2 == BTBOP1)
                begin
                //if load
                if((OpcodeMEM == 4'b0110)||(OpcodeMEM == 4'b0100))
                        ForwardToMux3 = 011;
                else
                        ForwardToMux3 = 001;
                        HazardDetected[0]=1;
                end
                else if(OP2 == OAOP1)
                begin
                        ForwardToMux3 = 011;
                        HazardDetected[0]=1;
                end

                //Test for OP2 IF Swap(Only time that OP2 changes)
                if((OpcodeMEM == 0001)&& (FunctionCodeMEM ==1111))
                begin
                        if(OP2 == BTBOP2)
                        begin
                        //if load
                        if((OpcodeMEM == 4'b0110)||(OpcodeMEM == 4'b0100))
                                ForwardToMux3 = 011;
                        else
                                ForwardToMux3 = 001;
                                HazardDetected[0]=1;
                        end
                end
                else if((OpcodeWB == 0001)&& (FunctionCodeWB ==1111))
                begin
                        if(OP2 == OAOP2)
                        begin
                                ForwardToMux3 = 100;
                                HazardDetected[0]=1;
                        end
                end
        end
end
endmodule
```

ShiftLeft

```
module ShiftLeft(input [15:0] a,
                        output reg [15:0] Result);
always @(*)
begin

        Result = a << 1;

end
endmodule
```

SignExtentions

NOTE: SignExtention components in ID and MEM function as zero extensions but exists where a sign extend was specified. The purpose of this is detailed in the status report.

```
module SignExtendID(input [3:0] a,
                    input [11:0]b,
                    output reg [15:0] ResultA, ResultB);
always @(*)
begin
	ResultA = {12'h000,a};
	ResultB = {4'h0, b};
end
endmodule

module SignExtendMEM(input [7:0] a,
                     output reg [15:0] Result);
always @(*)
begin
	Result = {8'h00, a};

end
endmodule

module SignExtendWB(input [15:0] a, b,
            output reg [31:0] ResultA, ResultB);
always @(*)
begin

	ResultA = {a[15],a[15],a[15],a[15],
	           a[15],a[15],a[15],a[15],
	           a[15],a[15],a[15],a[15],
	           a[15],a[15],a[15],a[15],
	           a};
	ResultB = {b[15],b[15],b[15],b[15],
	           b[15],b[15],b[15],b[15],
	           b[15],b[15],b[15],b[15],
	           b[15],b[15],b[15],b[15],
	           b};
end
endmodule
```

ZeroExtend

```
module ZeroExtend(input [7:0] a,
                        output reg [15:0] Result);
always @(*)
begin
        Result = {8'h00,a};
end
endmodule
```

Test Assembly Program

Test Assembly Code:
The register file, data memory, and instruction memory modules are initialized based on the given immediate values and instructions. The memory and register file are re-initialized with the given values on each reset.

Register Content
===== ======
R1 0E12
R2 0045
R3 F08F
R4 F076
R5 0084
R6 6789
R7 00EB
R8 FF56
R9 0000
R10 0000
R11 0000
R12 CC89
R13 0002
Others 0000
Instruction Memory
Address Content
===== ========
00 ADD R1, R2
02 SUB R2, R13
04 MOV R4, R8
06 OR R8, 0000
08 SWP R4, R6
0A LBU R7, 4(R9)
0C ANDi R3, 4C
0E SUB R14, R14
10 SB R7, 6(R9)
12 LW R6, 8(R9)
14 BEQ R7, 4
16 ADD R11, R1
18 BLT R7, 5
1A ADD R11, R2
1C BGT R7, 2

```
1E ADD R1, R1
20 ADD R1, R1
22 LW R8, 0(R9)
24 ADD R8, R8
26 SW R8, 2 (R9)
28 LW R10, 2 (R9)
2A ADD R12, R10
2C SUB R12, R13
2E ADD R12, R13
30 EF20
Other memory locations = 0000
Data Memory
00 3CAD
02 0000
04 1463
06 DAED
08 FEEB
0A FFFF
0E CCCC
Other memory locations = 0000
```

Expected Test Results:

Registers:

R0: 0000
R1: 0e57
R2: 0043
R3: 000C
R4: 6789
R5: 0084
R6: FEEB
R7: 0063
R8: 795A
R9: 0000
R10:795A
R11: 0e9a
R12: CC89
R13: 0002
R14: 0000
Others: 0000

Data Memory:
00: 3CAD
02: 795A
04: 1463
06: DA63
08: FEEB
0A: FFFF
0E: CCCC
Other memory locations: 0000

Stimulus Module Used

CPU_Fixture

```
`include "cpu.v"
module cpu_fixture();
reg clk, rst;
reg i;
cpu cpu(.clk(clk), .rst(rst));
initial
   $vcdpluson;
initial
begin
        //Here, Each clk cycle is #20 or 20 ns
        clk = 1'b0;
        forever #10 clk = ~clk;
end
initial
begin

        $monitor("\nPC: %h\nRegisters:\n[0]:%h\n[1]:%h\n[2]:%h\n[3]:%h\n[4]:%h\n[5]:%h\
                    n[6]:%h\n[7]:%h\n[8]:%h\n[9]:%h\n[10]:%h\n[11]:%h\n[12]:%h\n[13]:%h\n[
                    14]:%h\n[15]:%h\n\nMemory:\n[0]:%h\n[1]:%h\n[2]:%h\n[3]:%h\n[4]:%h\n[
                    5]:%h\n[6]:%h\n[7]:%h\n[8]:%h\n[9]:%h\n[A]:%h\n[B]:%h\n[C]:%h\n[D]:%
                    h\n[E]:%h\n[F]:%h",
                        cpu.ProgramCounter.PC,
                        cpu.RF.Registers[0], cpu.RF.Registers[1], cpu.RF.Registers[2],
                        cpu.RF.Registers[3], cpu.RF.Registers[4], cpu.RF.Registers[5],
                        cpu.RF.Registers[6], cpu.RF.Registers[7], cpu.RF.Registers[8],
                        cpu.RF.Registers[9], cpu.RF.Registers[10], cpu.RF.Registers[11],
                        cpu.RF.Registers[12], cpu.RF.Registers[13],cpu.RF.Registers[14],
                        cpu.RF.Registers[15],

                        cpu.DM.Data[0], cpu.DM.Data[1], cpu.DM.Data[2], cpu.DM.Data[3],
                        cpu.DM.Data[4], cpu.DM.Data[5], cpu.DM.Data[6], cpu.DM.Data[7],
                        cpu.DM.Data[8], cpu.DM.Data[9], cpu.DM.Data[10], cpu.DM.Data[11],
                        cpu.DM.Data[12], cpu.DM.Data[13], cpu.DM.Data[14],
                        cpu.DM.Data[15]);
end
initial
begin
        rst = 1'b0;
        #8 rst =1'b1;
end
initial
        #580 $finish;
endmodule
```

Results

```
results Wed Dec 09 17:23:59 2020 1
Chronologic VCS simulator copyright 1991-2018
Contains Synopsys proprietary information.
Compiler version O-2018.09-SP2-3_Full64; Runtime version O-2018.09-SP2-
3_Full64; Dec
9 17:23 2020
VCD+ Writer O-2018.09-SP2-3_Full64 Copyright (c) 1991-2018 by Synopsys Inc.
PC: 0000
Registers:
[0]:0000
[1]:0e12
[2]:0045
[3]:f08f
[4]:f076
[5]:0084
[6]:6789
[7]:00eb
[8]:ff56
[9]:0000
[10]:0000
[11]:0000
[12]:cc89
[13]:0002
[14]:0000
[15]:0000
Memory:
[0]:3c
[1]:ad
[2]:00
[3]:00
[4]:14
[5]:63
[6]:da
[7]:ed
[8]:fe
[9]:eb
[A]:ff
[B]:ff
[C]:00
[D]:00
[E]:cc
[F]:cc

……………
Outputs omitted until the last instruction, where an opcode exception is
detected.
……………

PC: 0030
Registers:
[0]:0000
[1]:0e57
[2]:0043
[3]:000c
```

```
[4]:6789
[5]:0084
[6]:feeb
[7]:0063
[8]:795a
[9]:0000
[10]:0000
[11]:0e9a
[12]:cc89
[13]:0002
[14]:0000
[15]:0000
Memory:
[0]:3c
[1]:ad
[2]:79
[3]:5a
[4]:14
[5]:63
[6]:da
[7]:63
[8]:fe
[9]:eb
[A]:ff
[B]:ff
[C]:00
[D]:00
[E]:cc
[F]:cc
$finish called from file "cpu_fixture.v", line 45.
$finish at simulation time 580
V C S S i m u l a t i o n R e p o r t
Time: 580
CPU Time: 0.310 seconds; Data structure size: 0.0Mb
Wed Dec 9 17:23:59 2020
```