Faculty of Engineering and Technology

Electrical and Computer Engineering Department

ENCS4370

Computer Architecture

Course Project Report #2

Student Name: Ramiz Shahin

ID:1201661

Student Name: Nour Battat

ID:1210075

Student Name: Robert Bannoura

ID:1210574

Instructors: Dr. Ayman Hroub & Dr.Aziz Qaroush

June – 2024

Abstract

The aim of the project is to create a multi-cycle Reduced Instruction Set Computing (RISC) processor in Verilog. The main goal is to develop a processor that efficiently performs instructions in multiple cycles, improving the speed of performance. The project comprises of various stages, instruction fetching, decoding, execution, and memory access, with different modules created to perform these stages of processing. The process involves meticulous testing and verification to ensure the correctness of all the modules, and the processor as a whole. The expected outcome of the project is a full design of a multi-cycle processor that can correctly demonstrate and perform the different stages of processing, while maintaining efficiency and performance.

Contents

Introduction

In digital design and computer architecture, Verilog is used to code and test various modules to construct a complete system. This project employs Verilog to create different modules involved in processing instructions for a multi-cycle Reduced Instruction Set Computing (RISC) processor, which is divided into distinct stages. The processor processes instructions through five stages: instruction fetch (IF), decoding (ID), execution (E), memory access (M), and write-back (WB). Each stage consists of specific modules: the IF stage includes the Program Counter (PC) and instruction memory; the ID stage includes the register file for managing processor registers; the E stage performs ALU operations using the ALU module; the M stage handles memory access through the data memory module; and the WB stage determines the data to be written back to the register file. This modular approach ensures efficient and accurate instruction processing.

Datapath

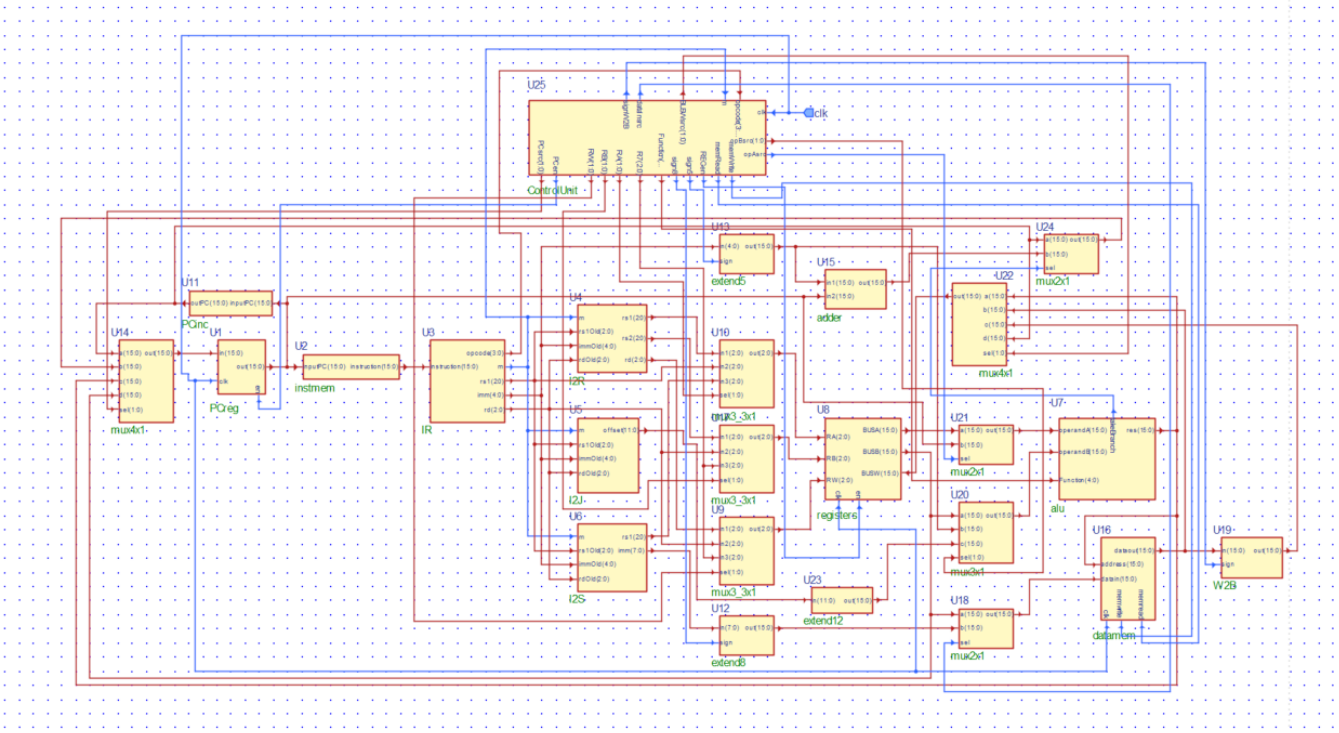


Figure 1: Datapath Diagram

The provided Screenshot shows a complex datapath design commonly found in CPU architectures, it has various components such as registers, multiplexers (MUXes), an Arithmetic Logic Unit (ALU), a control unit, and memory interfaces. Central to this design is the ALU, which performs essential arithmetic and logical operations on data inputs. Registers store intermediate data and results, while MUXes control the data flow by selecting inputs based on control signals generated by the control unit. The instruction register holds the current instruction, which the control unit decodes to generate the necessary control signals. The Program Counter (PC) maintains the address of the next instruction, facilitating the instruction fetch process from memory. During operation, the datapath follows a systematic sequence: fetching the instruction from memory via the PC, decoding the instruction to determine the operation and generate control signals, executing the operation in the ALU with data from registers or immediate values, and then either writing the result back to registers or accessing memory if the instruction involves a load or store operation. The control unit orchestrates this process, ensuring that each component operates in harmony to execute the instruction set effectively. This integration of components allows for the efficient execution of computational tasks, forming the core operational framework of a CPU.

Design Implementation

The implementation of our project isn't a big difference than the single/multi cycle design we learnt in the course slides, we decided to use multi-cycle to get the full mark ;)

Of course, we did some modifications, in order to optimize performance and ensure seamless functionality.

Our implementation included adding a fourth PC source used for the RET instruction. The details of our implementation is as the following:

1- Instruction fetching is done by taking the address of the PC register, and getting the instruction from the instruction memory based on

little-endian byte ordering.

2- In the decoding stage, the instruction is saved in the instruction register as an I-type instruction, and the output of the IR is wired

to different type converters (R-type, J-type, and S-type), and the registers sources are specified based on the signals from the CU (whether

from the I-type, R-type, J-type, or S-type).

3- The execution function of the instruction, is defined by the CU. A "takeBranch" signal was added to specify whether the branch is taken,

or just increment the PC by two if branch requirements not reached.

4- Data memory is utilized for loading and storing instructions, with an added signal to specify whether a 16-bit word or an 8-bit byte is being stored.

5- The write-back stage has four different sources for writing to the RW. The first source is the ALU results, the second is data memory

when loading a word, the third is data memory when loading a byte, and the fourth is from the PC register.

As for the design choices, I chose to deliver a static 3'b111 value from the CU to both, Register B source and Register W source, which will be used

to write, and read the return address when using the CALL and RET instructions. I also decided to add a third signal to the "data memory" with

"memwrite" and "memread", which is "wrByte" -read as write byte- to store a single byte to the memory (used with Sv instruction). Other signals are

commonly used in other designs, such as "PCsrc" to choose whether we're branching, jumping, or just incrementing. "BUSWsrc" to choose the source of

the data we are storing in "REGW" for example ALU results, loading word or byte from memory, and the output of PC incremented.

Instruction Fetch

The instruction fetch stage is a fundamental part of a CPU's operation where the next instruction to be executed is retrieved from memory. It begins with the CPU fetching the instruction address from the program counter, accessing memory, and loading the instruction into a dedicated register. This stage ensures that the CPU is prepared to decode and execute the instruction efficiently in subsequent stages of the instruction cycle. Efficient handling of the fetch stage is critical for overall CPU performance, as it sets the groundwork for timely execution of program instructions and supports advanced techniques like pipelining for enhanced throughput.

Program Counter (PC)

The Program Counter (PC) acts like a guidebook for the CPU, keeping track of where it should look next for instructions to carry out. It's essentially a memory address pointer that tells the CPU where in the program's instructions it currently is. When the CPU finishes fetching an instruction, the PC moves forward to point to the next instruction in line. This process ensures that the CPU follows the program's sequence correctly, step by step, ensuring smooth and orderly execution of tasks. The Verilog module `PCreg` implements a 16-bit Program Counter (PC) register. It takes inputs for clock (`clk`), enable (`en`), and a 16-bit data input (`in`) to update the PC's value. The `out` output reflects the current value stored in the PC register. Initialization sets the PC to zero, and it updates on the positive edge of `clk` when `en` is active, loading `in` into the register. The testbench (`tb_PCre`) generates a clock signal, controls `en` and `in` inputs over time, and monitors the PC register's output (`out`) during simulation.

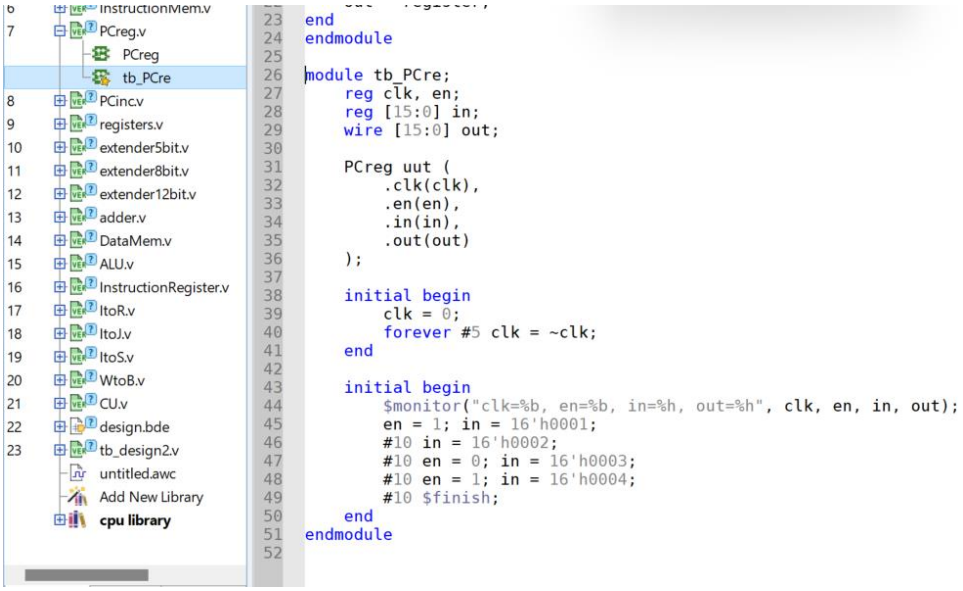


Figure 3: PC Module Testbench

This testbench effectively simulates the behavior of the PCreg module under different input conditions (en and in changes) while monitoring the output (out) at each simulation step. It demonstrates the functionality of the PCreg module in a controlled environment to verify its correctness and operation.

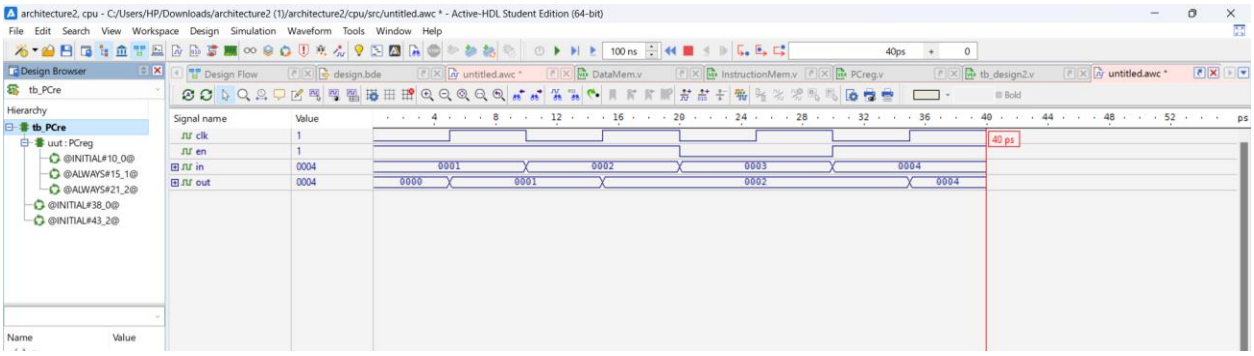
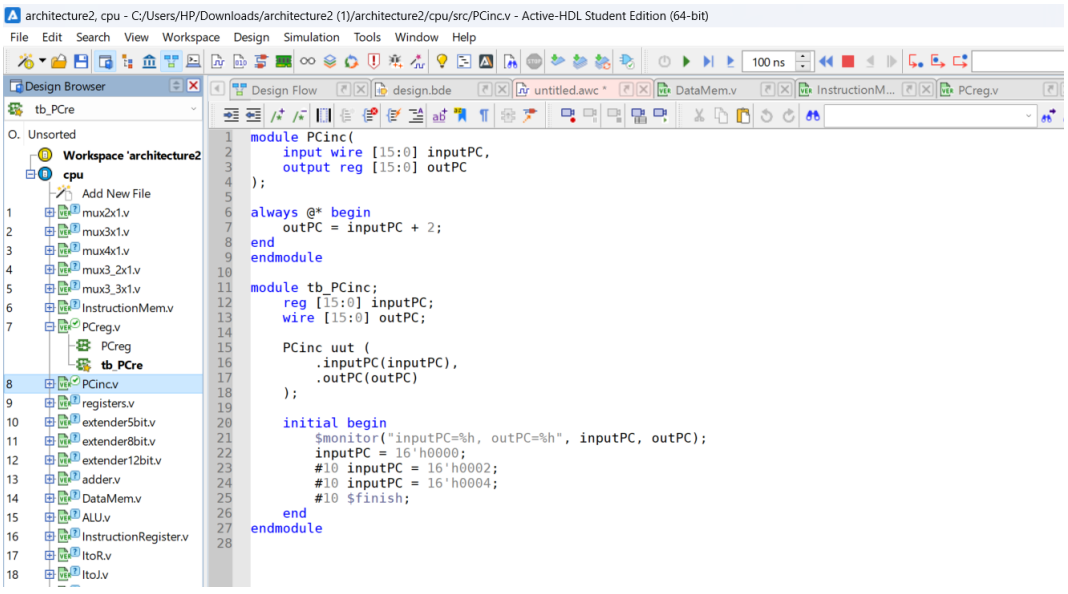


Figure 4: PC Testbench Waveform

Pc incremental:



The Verilog code provided consists of two modules: `PCinc` and its corresponding testbench `tb_PCinc`. The `PCinc` module incrementally adds 2 to a 16-bit input (`inputPC`) and outputs

the result (`outPC`). Inside `PCinc`, an `always @*` block continuously computes `outPC` as `inputPC + 2`, ensuring that whenever `inputPC` changes, `outPC` updates accordingly.

In the `tb_PCinc` testbench, `inputPC` is initialized to `16'h0000`, and subsequent changes (`16'h0002` and `16'h0004`) are timed using delays (`#10`). The `$monitor` statement in the testbench prints the values of `inputPC` and `outPC` in hexadecimal format at each simulation step. As expected, the simulation results demonstrate the correct functionality of the `PCinc` module: `outPC` increments from `0002` to `0004` and then to `0006`, reflecting the incremented values of `inputPC`. This setup effectively verifies that the `PCinc` module operates as intended, incrementing input values by 2 and producing the expected output values in a simulated environment.

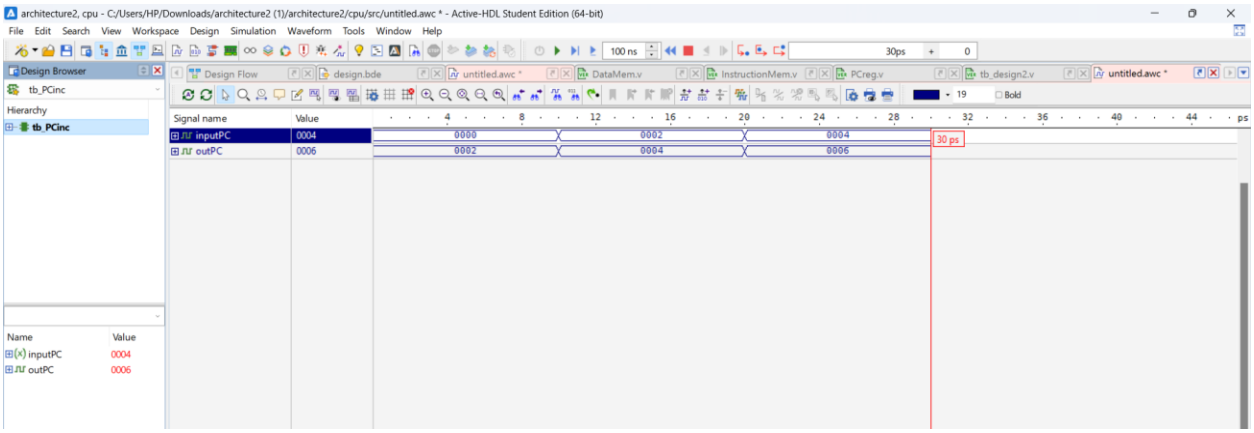


Figure 5: PC incremental Testbench Waveform

Instruction Memory

The instmem module models an instruction memory. It contains an internal memory (mem) initialized with various machine instructions represented by predefined opcodes (parameter declarations). Each instruction is stored in 8-bit segments (mem[0] to mem[31]). The module continuously updates the instruction output based on the inputPC input, fetching the appropriate instructions from memory. The instructions are split into two parts: the lower 8 bits (mem[inputPC[15:0]]) and the upper 8 bits (mem[inputPC[15:0] + 1]). This arrangement facilitates fetching of instructions based on the current program counter (inputPC), allowing the module to output the correct instruction in response to changes in inputPC.

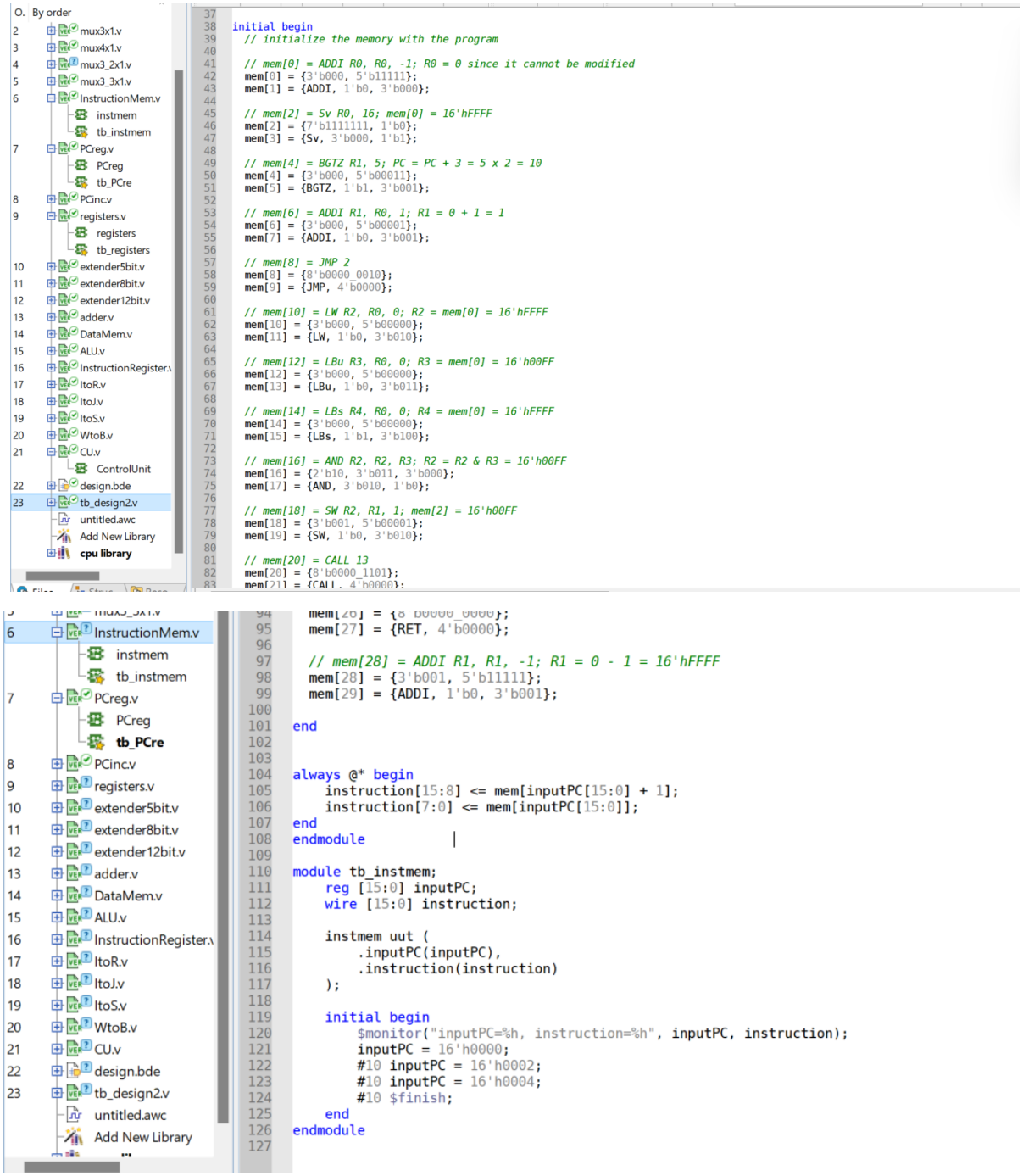


Figure 6: Instruction Memory Testbench

The instruction memory testbench in Figure 6 determines the address holding the instruction, which is set in the instruction memory module.

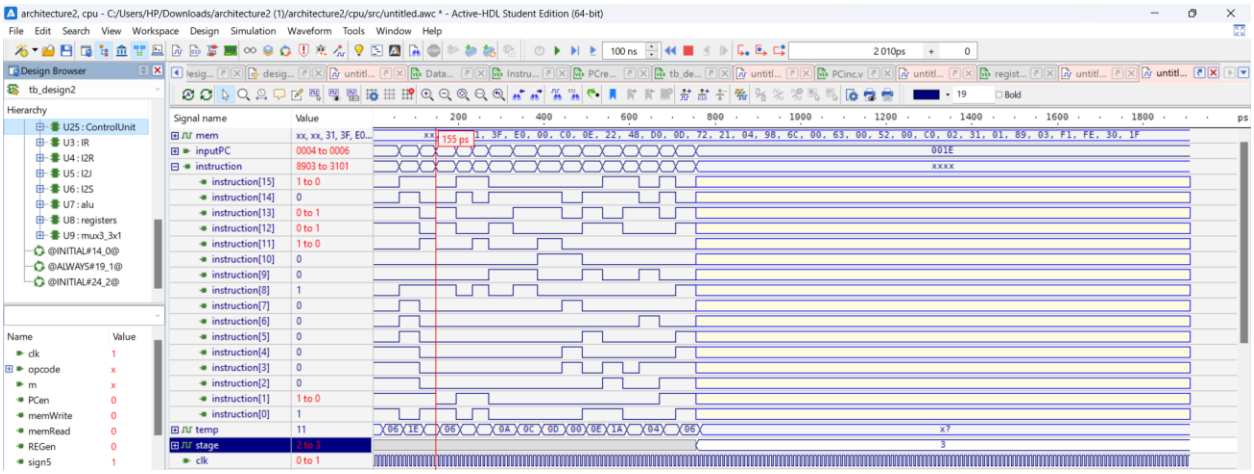


Figure 7: Instruction Memory Testbench Waveform

Here, instruction fetch is happening, the opcode has changed to 3 >> ADDI and the instructions should hold the value 0011 as defined in the code of instmem module

Instruction Decode

This stage decodes the instruction to be executed, and the operands needed to execute said instruction.

Register File

The register file contains all the registers within the CPU. The register file is used to organize and manage the registers of the CPU. This module supports read and write operations during the execution of an instruction.

```
module tb_registers;
    reg clk, en;
    reg [2:0] RA, RB, RW;
    reg [15:0] BUSW;
    wire [15:0] BUSA, BUSB;

    registers uut (
        .clk(clk),
        .en(en),
        .RA(RA),
        .RB(RB),
        .RW(RW),
        .BUSW(BUSW),
        .BUSA(BUSA),
        .BUSB(BUSB)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        $monitor("clk=%b, en=%b, RA=%b, RB=%b, RW=%b, BUSW=%h, BUSA=%h, BUSB=%h, reg[RA]=%h, reg[RB]=%h, reg[RW]=%h", clk, en, RA, RB, RW, BUSW, BUSA, BUSB, uu
        en = 1; RW = 3'b000; BUSW = 16'hAAAA;
        #10 RA = 3'b001; RB = 3'b000;
        #10 RW = 3'b010; BUSW = 16'h5555;
        #10 RA = 3'b010; RB = 3'b001;
        #10 $finish;
    end
endmodule
```

Figure 8: Register File Testbench

The testbench for the register file sets some test cases for content of registers, testing cases with write enabled and write disabled, seeing the waveform in Figure 9, the registers store the correct values when writing is enabled, and overwriting of registers is blocked when writing is disabled.

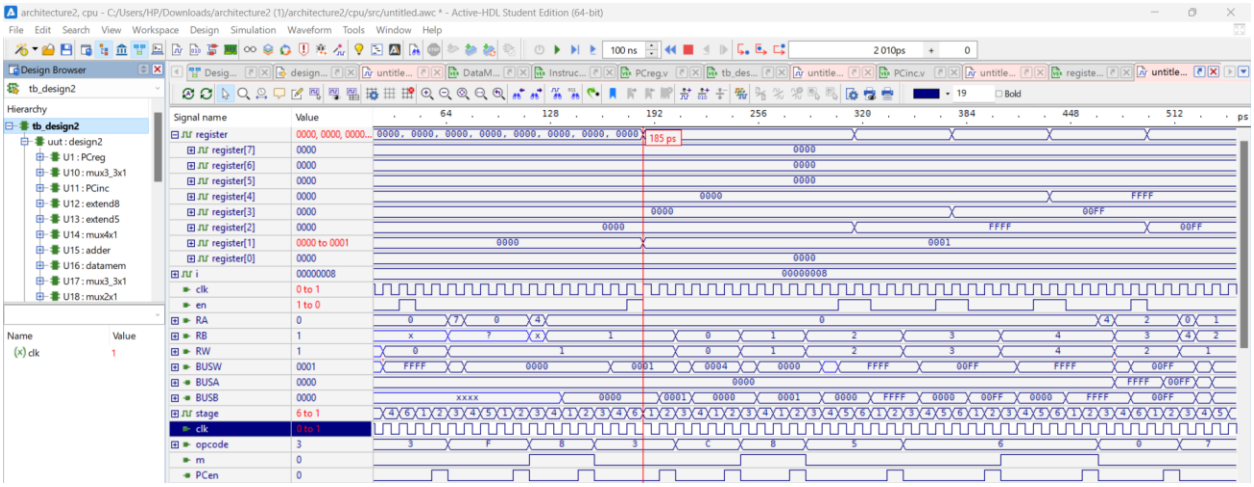


Figure 9: Register File Testbench Waveform

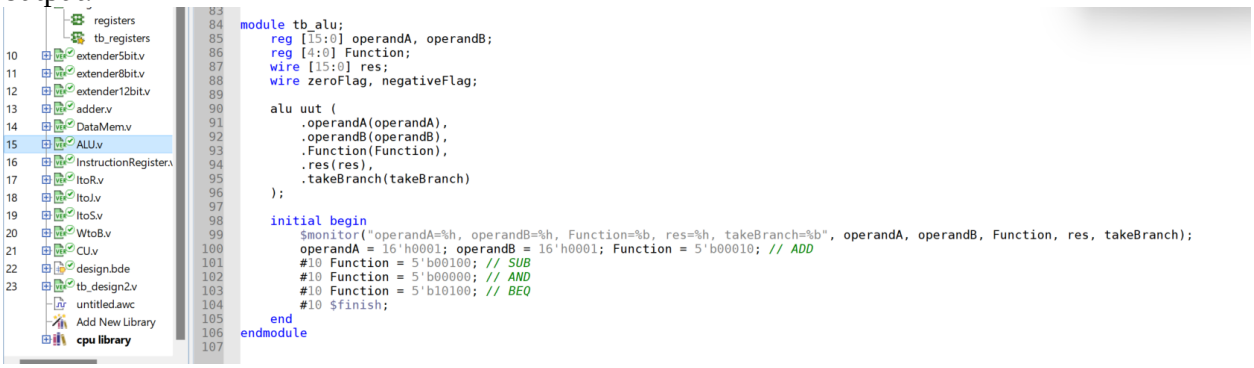
In stage 6 to 1 the instruction ADDi writes back the result from the ALU to Reg[1] and the enable was changed from 0 to 1 for writing , the values of register RA=0 and RB=1 and the right result was written on RW

Instruction Execute

The ALU executes the decoded instruction from the decode stage, performing the specified operation with the appropriate operands.

ALU

The Arithmetic Logic Unit (ALU) is responsible for performing both arithmetic and logical operations on the operands that it receives, based on the received opcode. The ALU Operation signal determines the needed operation for execution of instructions.
Operands A and B are given test values to check that the ALU operations provide the correct output.



```
registers
tb_registers
extender5bit.v
extender8bit.v
extender12bit.v
adder.v
DataMem.v
ALU.v
InstructionRegister.v
ItoR.v
ItoJ.v
ItoS.v
WtoB.v
CU.v
design.bde
tb_design2.v
untitled.awc
Add New Library
cpu library
```

```
module tb_alu;
    reg [15:0] operandA, operandB;
    reg [4:0] Function;
    wire [15:0] res;
    wire zeroFlag, negativeFlag;

    alu uut (
        .operandA(operandA),
        .operandB(operandB),
        .Function(Function),
        .res(res),
        .takeBranch(takeBranch)
    );

    initial begin
        $monitor("operandA=%h, operandB=%h, Function=%b, res=%h, takeBranch=%b", operandA, operandB, Function, res, takeBranch);
        operandA = 16'h0001; operandB = 16'h0001; Function = 5'b00010; // ADD
        #10 Function = 5'b00100; // SUB
        #10 Function = 5'b00000; // AND
        #10 Function = 5'b10100; // BEQ
        #10 $finish;
    end
endmodule
```

Figure 10: ALU Testbench

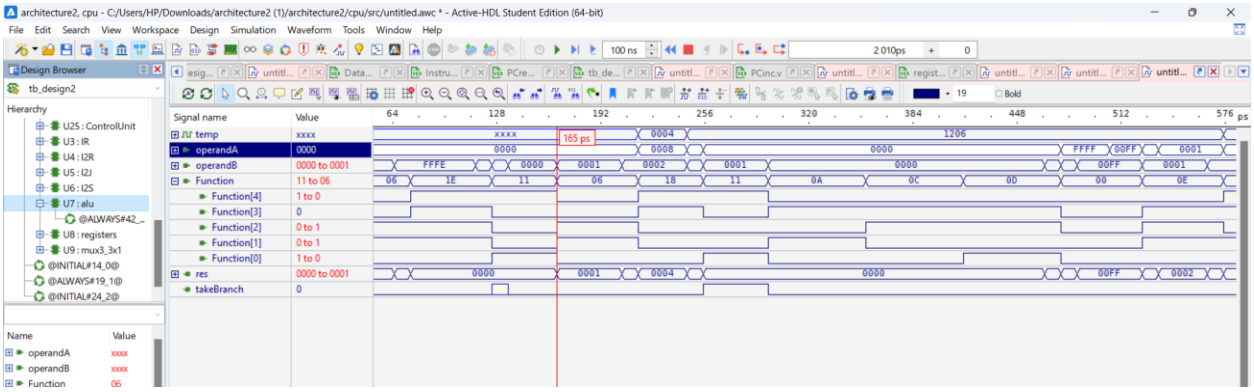


Figure 11: ALU Testbench Waveform

In the test bench, firstly the value of OpA was zero and we added 1 by the opcode of Adding function “00110” which was stored in OpB , so the final result as shown is 0+1=1 which is correct as expected.

Data Memory

The data memory is the component that stores data to be read and written by the processor. It is used to store and handle variables during processing.

```
module tb_datamem;
    reg clk;
    reg [15:0] address, datain;
    reg memread, memwrite;
    wire [15:0] dataout;

    datamem uut (
        .clk(clk),
        .address(address),
        .datain(datain),
        .memread(memread),
        .memwrite(memwrite),
        .dataout(dataout)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        $monitor("clk=%b, address=%h, datain=%h, memread=%b, memwrite=%b, dataout=%h, mem[add]=%h", clk, address, datain, memread, memwrite, dataout, uut.mem[a
        address = 16'h0000; datain = 16'hAAAA; memwrite = 1; memread = 0;
        #10 memwrite = 0; memread = 1;
        #10 address = 16'h0001; datain = 16'h5555; memwrite = 1; memread = 0;
        #10 memwrite = 0; memread = 1;
        #10 $finish;
    end
endmodule
```

Figure 12: Data Memory Testbench

The `tb_datamem` Verilog testbench verifies the operation of the `datamem` module, which simulates a basic memory unit. It initializes and controls signals like `clk`, `address`, `datain`, `memread`, and `memwrite` to organize read and write operations on an internal memory array (`mem`). Using `$monitor`, the testbench tracks and displays changes in these signals along with the resulting `dataout`, ensuring the module correctly stores and retrieves data during simulation. The test scenarios involve writing `16'hAAAA` to address `16'h0000`, then reading from both `16'h0000` and `16'h0001`, validating the `datamem` module's functionality in a simulated digital environment.

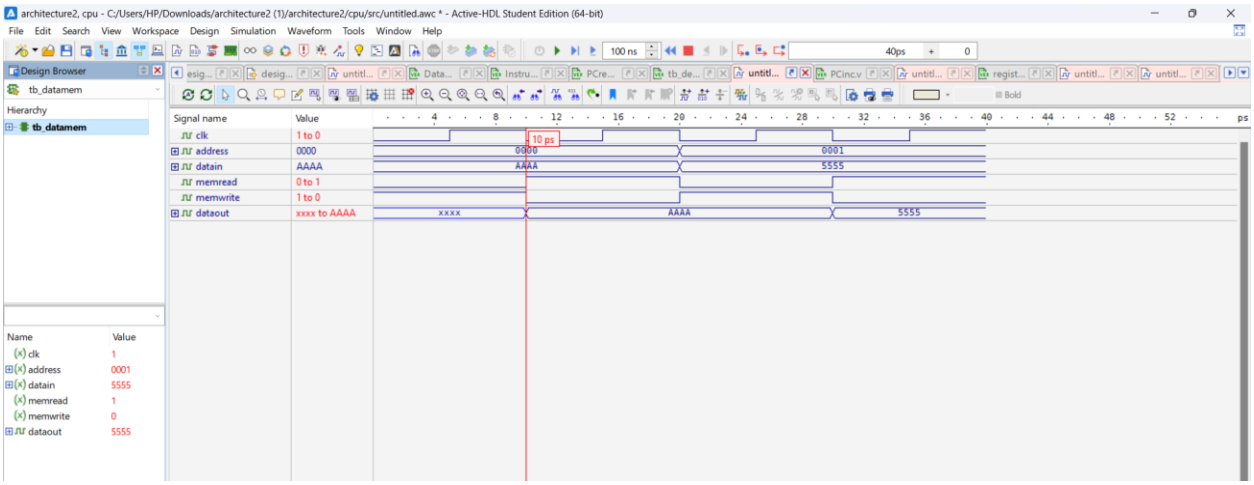


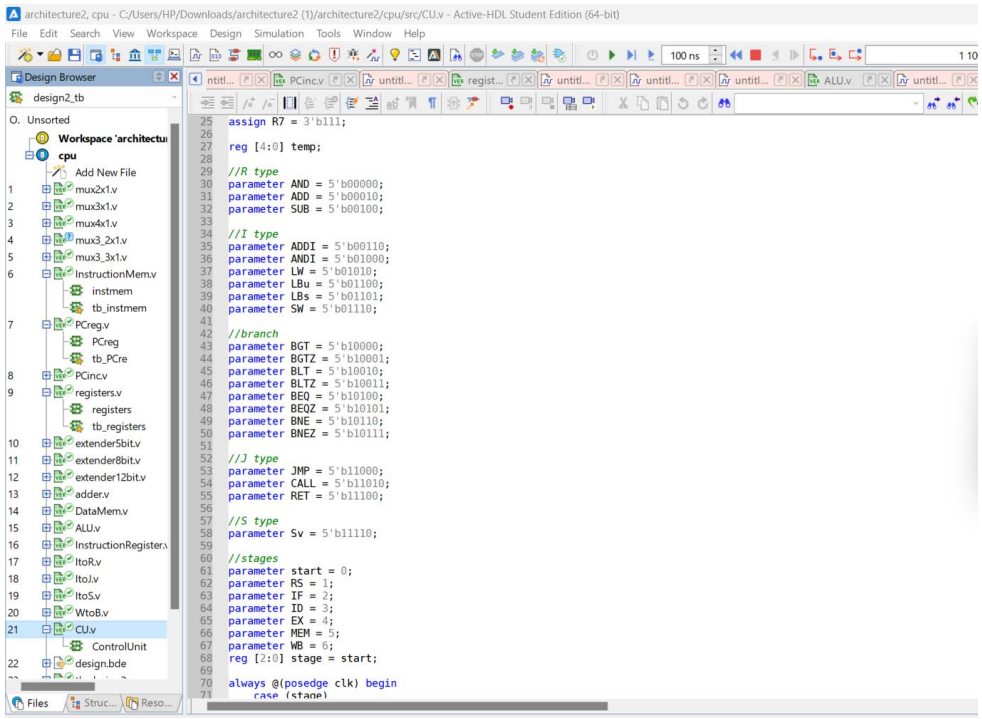
Figure 13: Data Memory Testbench Waveform

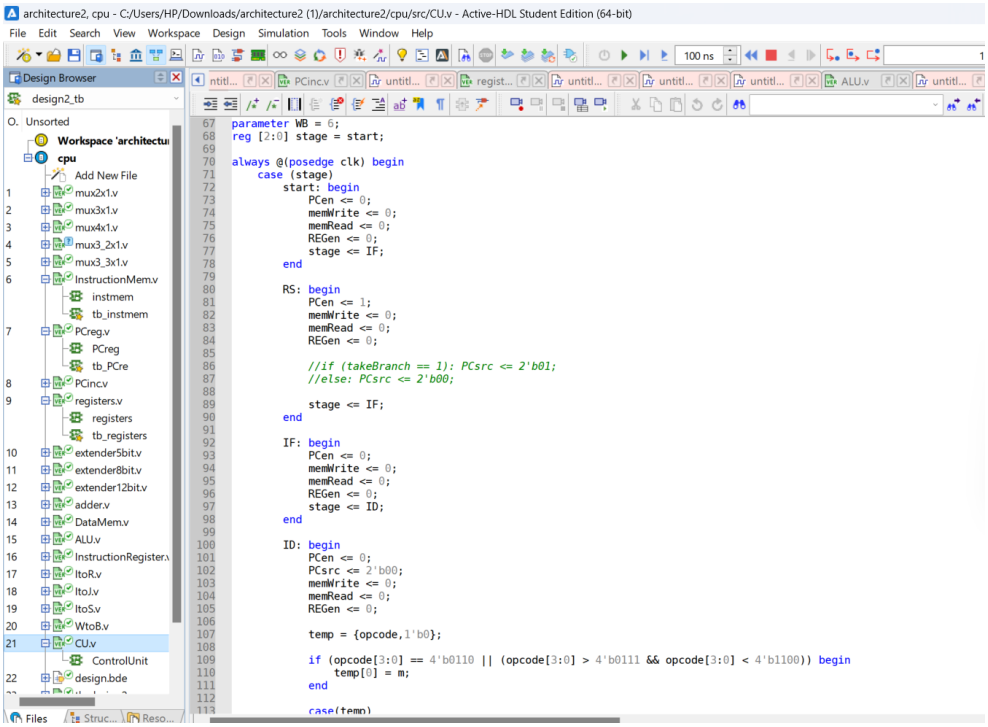
Writeback

The writeback stage completes the task of storing the executed instruction's output back into the register file. This stage follows the completion of all preceding stages, ensuring that the resulting data is accurately written back into the register file.The writeback stage is handled in the processor code, as it is composed of a 1-bit multiplexer, using the Write back signal to determine whether the value to be written back is the ALU result, or the data memory output.

Control Unit

The control unit (CU) sets all input signals for every processing stage. The output signals of the control unit are determined by the instruction code received by the processor. These signals are sent to the processor to manage the flow of all stages and ensure the correct execution of each instruction.





The provided Verilog code defines a `ControlUnit` module that controls the execution of instructions in a simple processor. The module takes an instruction's opcode and mode (`m`), and based on these, sets various control signals that manage the data flow and operations in the processor.

These control signals include enabling the program counter (`PCen`), controlling memory read/write operations (`memWrite` and `memRead`), register enable (`REGen`), and selecting data sources for the ALU and memory operations (`opAsrc`, `opBsrc`, `dataInsrc`, etc.).

The control unit also includes state parameters for different stages of instruction execution such as instruction fetch (`IF`), decode (`ID`), execute (`EX`), memory access (`MEM`), and write-back (`WB`). It uses a state machine to transition between these stages and sets the control signals accordingly based on the current instruction and stage.

The code includes several instruction parameters for different types of operations like arithmetic (`ADD`, `SUB`), immediate operations (`ADDI`, `ANDI`), load/store operations (`LW`, `SW`), branch instructions (`BGT`, `BEQ`), jump instructions (`JMP`, `CALL`), and a special save instruction (`Sv`). Each instruction type is handled in the `ID` and `EX` stages, setting the necessary control signals and transitioning to the appropriate next stage.

A temporary register (`temp`) is used to assist in decoding the opcode combined with the mode bit (`m`), and an always block sensitive to the positive edge of the clock (`clk`) manages the state transitions and control signal updates. Additionally, the module outputs a constant value `R7` as `3'b111`.

In summary, the `ControlUnit` module orchestrates the processor's operations by decoding instructions and setting control signals across different execution stages, ensuring the correct sequence of operations for each instruction type.

Waveform for Control Unit:

Here we can trace the flags and how the control unit controls them, Opcode 3 for ADDI

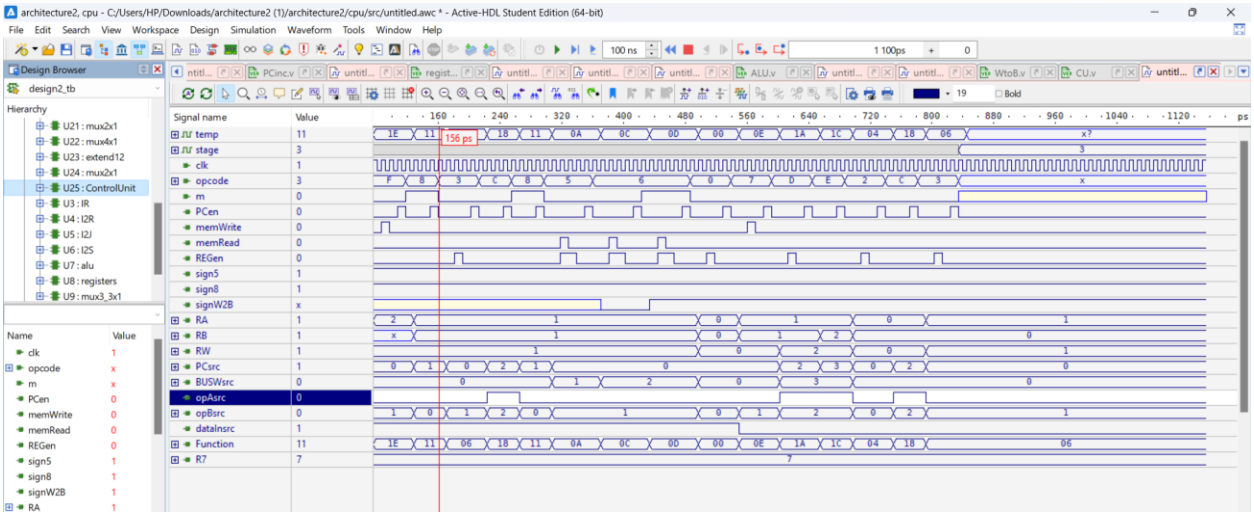


Figure 16: Control Unit Testbench Waveform

Control signals Table

	MEMWR	MEMR	REGEN (REG WRITE)	Sign5	wrByte	SignW2B	PCsrc	BUSWSRC (REG DEST)	opAsrc	opBsrc	RA	RB	RW	dataINsrc	M
R-TYPE															
AND	0	0	1	X	X	X	00	00	0	00	00	00	00	X	X
ADD	0	0	1	X	X	X	00	00	0	00	00	00	00	X	X
SUB	0	0	1	X	X	X	00	00	0	00	00	00	00	X	X
I-TYPE															
ADDI	0	0	1	1	X	X	00	00	0	01	01	X	01	X	X
ANDI	0	0	1	1	X	X	00	00	0	01	01	X	01	X	X
LW	0	1	1	1	X	X	00	01	0	01	01	X	01	X	X
Lbu	0	1	1	1	X	0	00	10	0	01	01	X	01	X	0
Lbs	0	1	1	1	X	1	00	10	0	01	01	X	01	X	1
SW	1	0	0	1	0	X	00	X	0	01	01	01	X	0	X
BGT	0	0	0	1	X	X	01	X	0	00	01	01	X	X	0
BGTZ	0	0	0	1	X	X	01	X	0	00	01	01	X	X	1
BLT	0	0	0	1	X	X	01	X	0	00	01	01	X	X	0
BLTZ	0	0	0	1	X	X	01	X	0	00	01	01	X	X	1
BEQ	0	0	0	1	X	X	01	X	0	00	01	01	X	X	0
BEQZ	0	0	0	1	X	X	01	X	0	00	01	01	X	X	1
BNE	0	0	0	1	X	X	01	X	0	00	01	01	X	X	0
BNEZ	0	0	0	1	X	X	01	X	0	00	01	01	X	X	1
J-TYPE															
JMP	0	0	0	X	X	X	10	X	1	10	X	X	X	X	X
CALL	0	0	1	X	X	X	10	11	1	10	X	X	10	X	X
RET	0	0	0	X	X	X	11	X	X	X	X	10	X	X	X
S-TYPE															
SV	1	0	0	X	1	X	0	X	0	X	10	X	X	1	X

Control Unit Boolean:

MEMWR: SW + SV

MEMR: LW + LBu + LBs

REGEN: AND + ADD + SUB + ANDI + ADDI + LW + Lbu + Lbs

Call Sign5: ADDI + ANDI + LW + Lbu + Lbs + SW + BGT + BGTZ + BLT + BLTZ + BEQ + BEQZ + BNE + BNEZ

WrByte: SV

SignW2B: Lbs

BUSWSRC: Lbs + BGTZ + BLTZ + BEQZ + BNEZ

OpAsrc: JMP + CALL

OpBsrc:

opBsrc[0]:JMP+CALL

opBsrc[1]:LW+LBU+LBS+SW

RA:

RA[0]:SV

RA[1]:ADDI+ANDI+LW+LBU+SW+BGT+BGTZ+BLT+BLTZ+BEQ+BEQZ+BNE+BNEZ

RB:

RB[0]: RET

RB[1]:SW+BGT+BGTZ+BLT+BLTZ+BEQ+BEQZ+BNE+BNEZ

RW:

RW[0]:CALL

RW[1]:ADDI+ANDI+LW+LBU+LBS

DataINsrc: SV

M: Lbs + BGTZ + BLTZ + BEQZ + BNEZ

BusBSRC:

BusBSRC[0]:Lbu+Lbs+CALL

BusBSRC[1]:LW+CALL

PCSRC:

PCSRC[0]:BGT+BGTZ+BLT+BLTZ+BEQ+BEQZ+BNE+BNEZ+RET

PCSRC[1]:JMP+CALL

Discussion of Control signals:

1.MEM WR:

To enable writing on memory when instructions like store is executed

2.MEM R:

To enable reading on memory when instructions like loadB , LW are executed

3.REG EN:

To enable writing back on registers when instructions like ADD, ADDI …etc. are executed

4.Sign 5:

To extend the immediate by 5 bits when I-type register is used

5. wrByte:

To extend the immediate by 8 bits when S-type register is used

6. Sign W2B

Used with J-type instructions as shown in the table

7.PC SRC:

Used To determine the operation on the pc , either adding the pc by 2 or using it for Branch instructions or Jump or for RET

00=PC+2

01=PC+imm > used for branching

10=JUMP

11=RET

8. BUS WSRC:

To determine which result to write from the datapath on the write back WB register

00=ALU

01=MEM

10=MEM

11=CALL

9.OPA src:

Decides From which register (I-type , R-type – J-type) , reg A will take its operand.

10.OPB src:

Decides From which register (I-type , R-type – J-type) , reg B will take its operand.

11. RA

Indicator where REG A took its operand

12. RB:

Indicator where REG B took its operand

13.RW

Indicator where to write the data back

14.DataIN src:

0:Data in from Rb

1: Data in from immediate

15. MOD bit:

X= don’t care

0 in load operations= unsigned load

0 in branch = normal comparison

1 in branch = compare to 0

1 in load operations = signed load

By reviewing our snapshots of the waveforms for each component, you can see that they work in correct functionality by running and testing the design using testbenches for each component , which also applies on the complete design of the datapath

~~Here are some test cases for some instructions:~~

SV instruction :

// mem[2] = Sv R0, 16; mem[0] = 16'hFFFF

mem[2] = {7'b1111111, 1'b0};

mem[3] = {Sv, 3'b000, 1'b1};

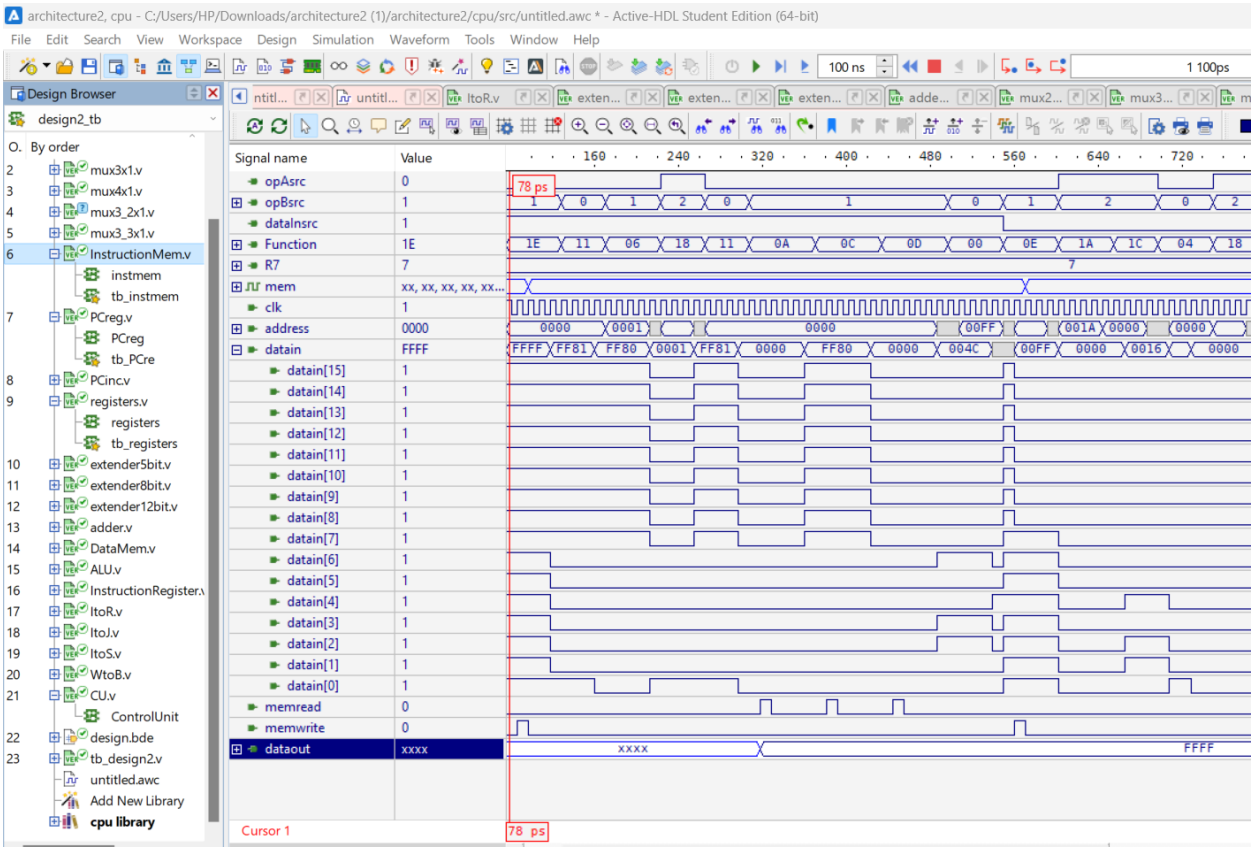


Figure 17: SV test

As we can see the value FFFF is stored in datain.

Branch:

// mem[4] = BGTZ R1, 5; PC = PC + 3 = 5 x 2 = 10

mem[4] = {3'b000, 5'b00011};

mem[5] = {BGTZ, 1'b1, 3'b001};

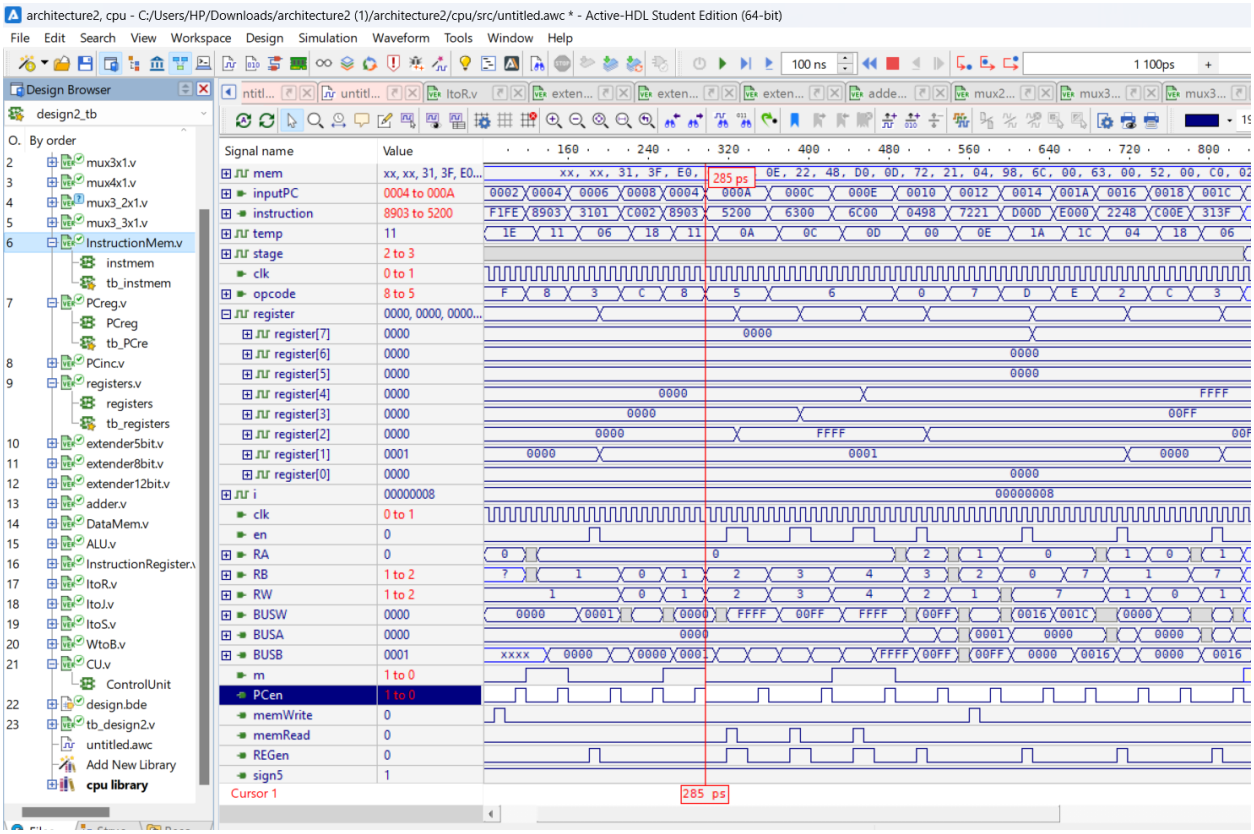


Figure 18:Branch Test

PC value = 000A =10

ADDI:

// mem[6] = ADDI R1, R0, 1; R1 = 0 + 1 = 1

mem[6] = {3'b000, 5'b00001};

mem[7] = {ADDI, 1'b0, 3'b001};

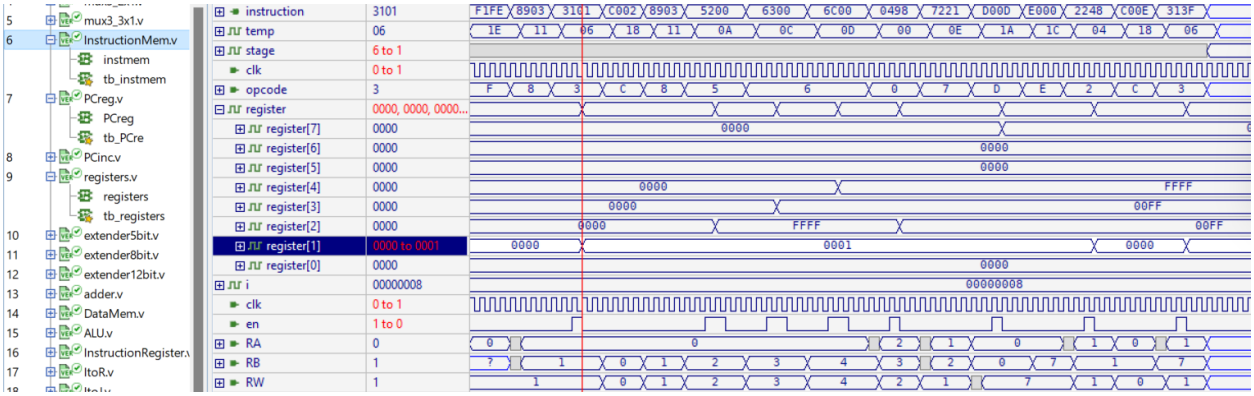


Figure 29:ADDI Test

R1 value changed to 1 by adding immediate 1 to the value 0 inside it

JUMP:

// mem[8] = JMP 2

mem[8] = {8'b0000_0010};

mem[9] = {JMP, 4'b0000};

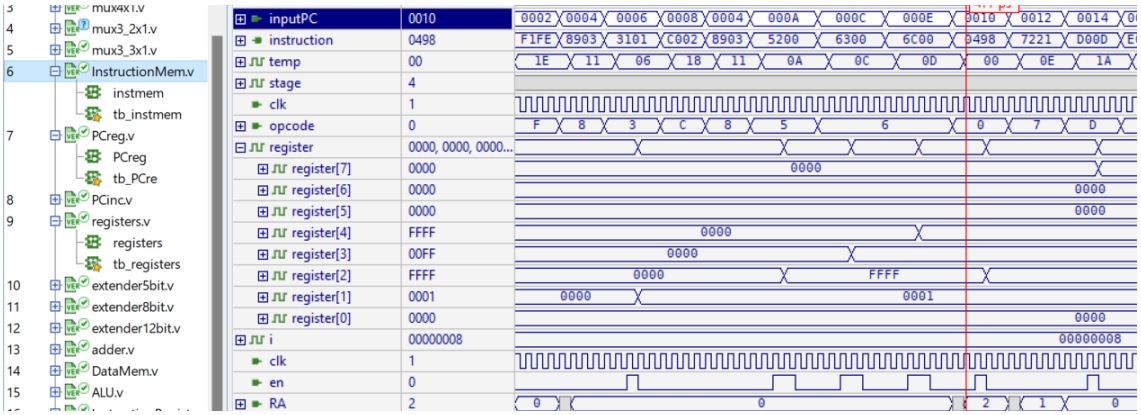


Figure 20: Jump test

Jump to Address 2 ,PC value is 0010

LoadW:

// mem[10] = LW R2, R0, 0; R2 = mem[0] = 16'hFFFF

mem[10] = {3'b000, 5'b00000};

mem[11] = {LW, 1'b0, 3'b010};

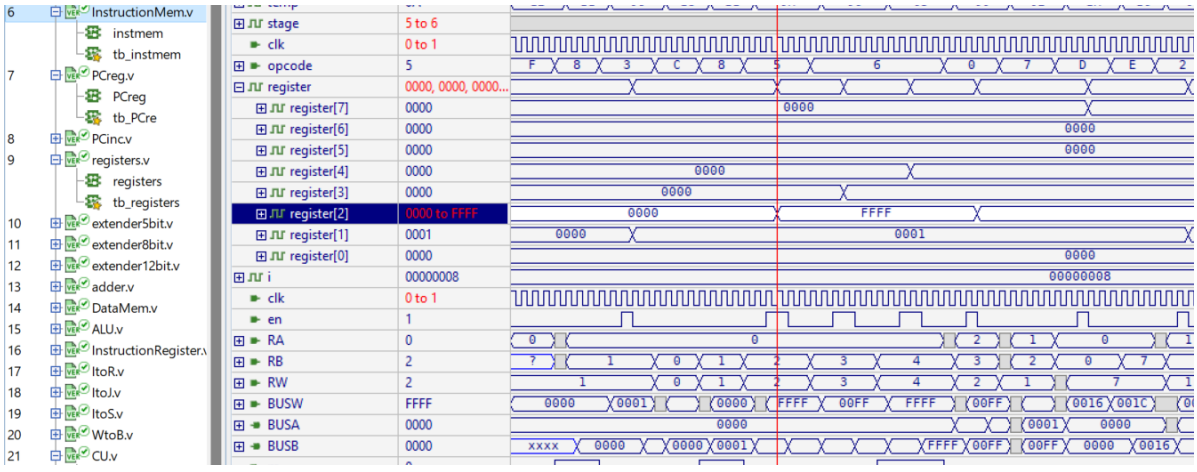


Figure 21: LoadW test

Load the value FFFF from Mem[0] to Register Reg[2]

Lbu:

The instruction LBu (Load Byte Unsigned) in an I-Type (Immediate Type) format is used to load an unsigned byte from memory into a register. This operation reads a byte from a memory location determined by adding an immediate value to the contents of a base register, and then stores the result in a destination register

// mem[12] = LBu R3, R0, 0; R3 = mem[0] = 16'h00FF

mem[12] = {3'b000, 5'b00000};

mem[13] = {LBu, 1'b0, 3'b011};

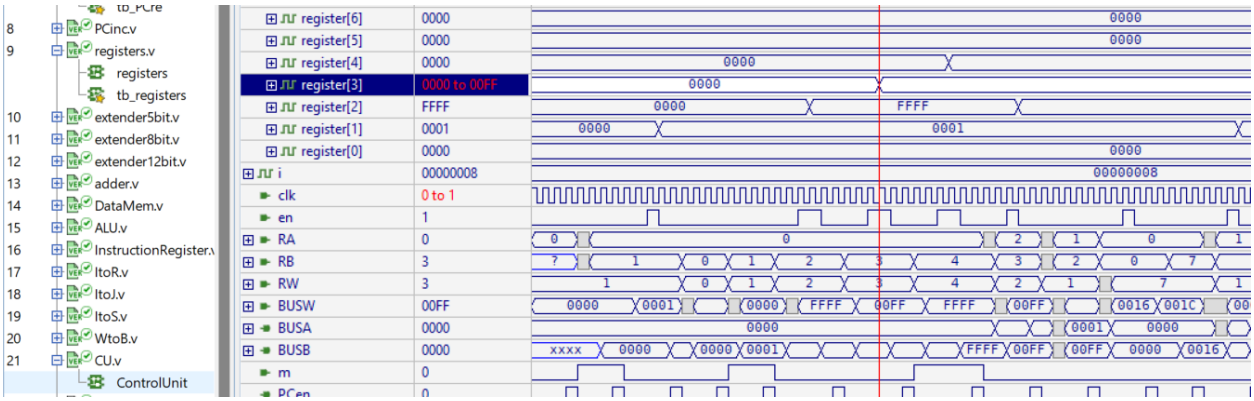


Figure 22: Lbu test

The value of Reg[3] is 00FF

LBs

The LBs (Load Byte Signed) instruction in an I-Type (Immediate Type) format is used to load a signed byte from memory into a register. This operation reads a byte from a memory location determined by adding an immediate value to the contents of a base register, and then stores the sign-extended result in a destination register

// mem[14] = LBs R4, R0, 0; R4 = mem[0] = 16'hFFFF

mem[14] = {3'b000, 5'b00000};

mem[15] = {LBs, 1'b1, 3'b100};

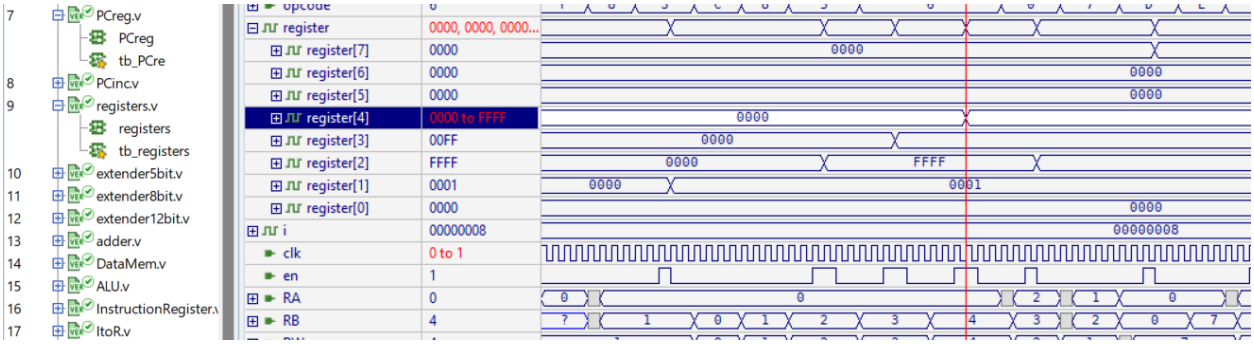


Figure 23: LBs test

The value of Reg[4] is set to FFFF

AND:

// mem[16] = AND R2, R2, R3; R2 = R2 & R3 = 16'h00FF

mem[16] = {2'b10, 3'b011, 3'b000};

mem[17] = {AND, 3'b010, 1'b0};

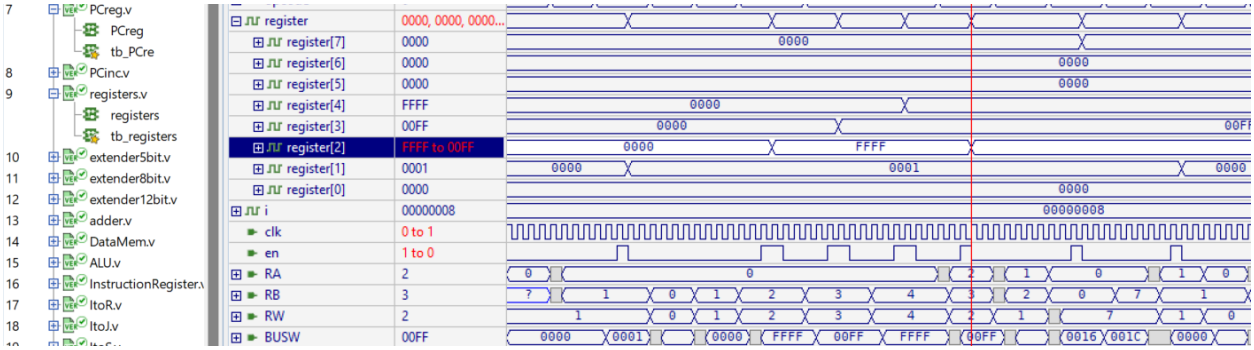


Figure 24: AND test

The value of Reg[2] = 00FF

SW:

The SW (Store Word) instruction in an I-Type (Immediate Type) format is used to store a word from a register into memory. This operation writes the contents of a register to a memory location determined by adding an immediate value to the contents of a base register

// mem[18] = SW R2, R1, 1; mem[2] = 16'h00FF

mem[18] = {3'b001, 5'b00001};

mem[19] = {SW, 1'b0, 3'b010};

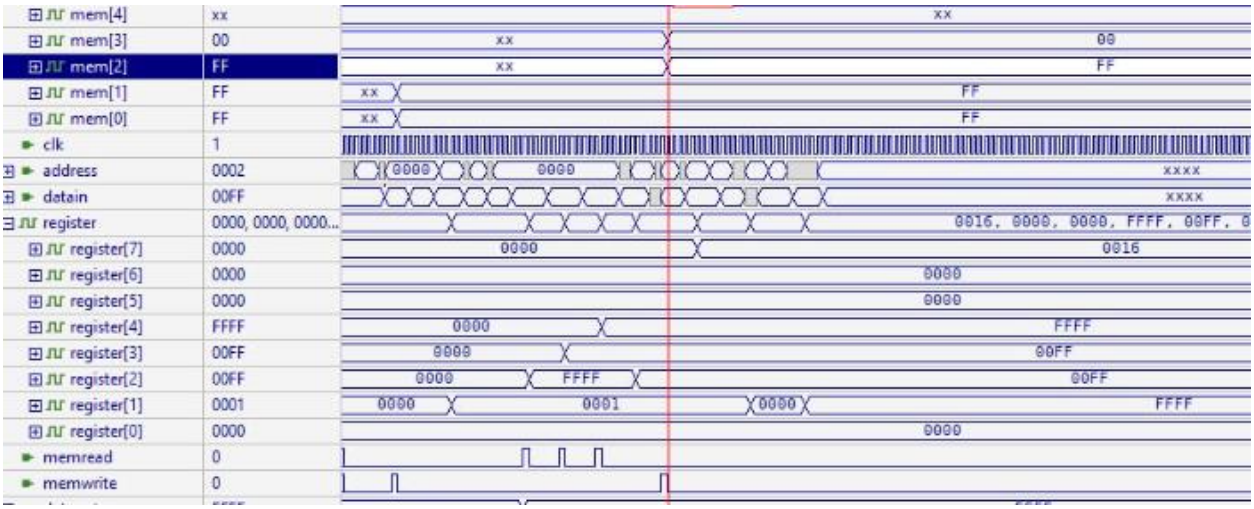


Figure 25: SW test

The value of MEM[2] equals to the value of Reg[2] = 00FF

CALL:

// mem[20] = CALL 13

mem[20] = {8'b0000_1101};

mem[21] = {CALL, 4'b0000};

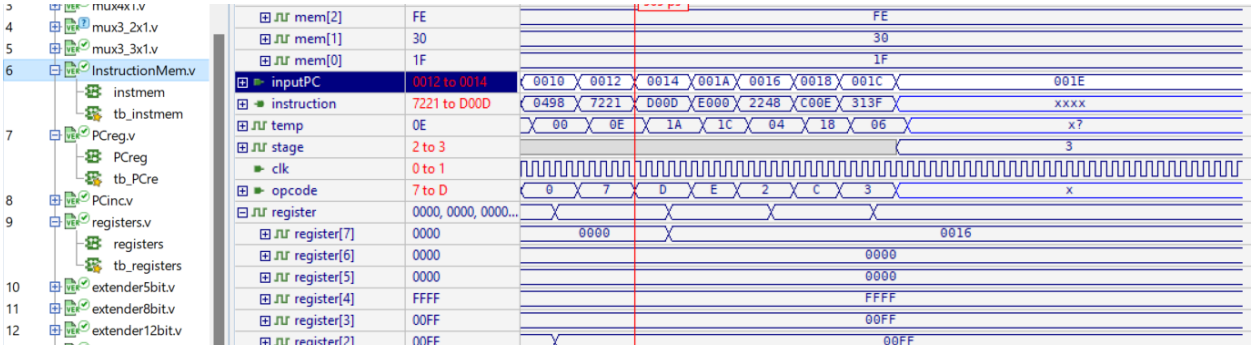


Figure 26: CALL test

PC turns from value 12 to 14 instead of 13 because PC only works on multiples of 2 .

SUB:

// mem[22] = SUB R1, R1, R1; R1 = 1 - 1 = 0

mem[22] = {2'b01, 3'b001, 3'b000};

mem[23] = {SUB, 3'b001, 1'b0};

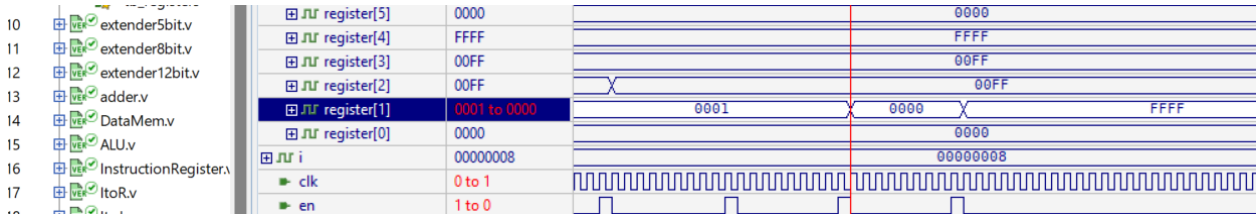


Figure 27: SUB test

The value of the R1 changes from 1 to 0 because we subtract 1 from 1 =0

JUMP:

// mem[24] = JMP 14

mem[24] = {8'b0000_1110};

mem[25] = {JMP, 4'b0000};

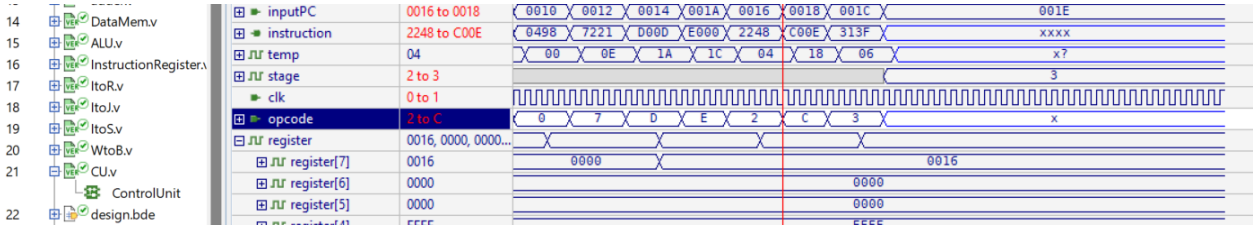


Figure 28: JUMP test

Instruction fetch for jump instruction

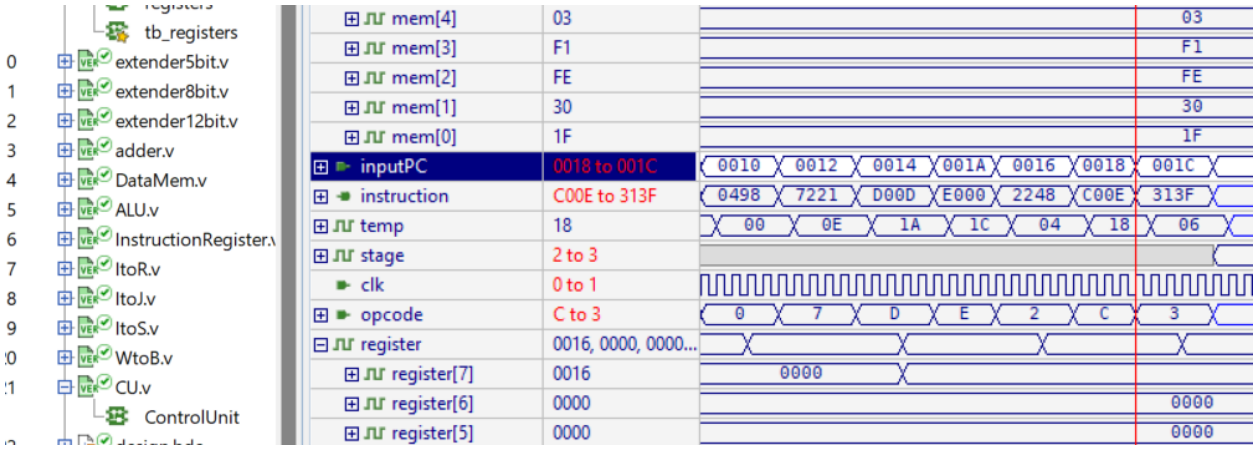


Figure 29: JUMP test

Then, after execution the jump instruction ,1110 was extended by 0 to the left which equals 11100 ==1C which is the new pc value

RET:

The **`RET`** instruction is crucial for returning from subroutines in assembly or machine code programming. By setting the Program Counter to the value stored in the `R7` register, the `RET` instruction ensures that the processor can resume execution at the correct return address, maintaining the correct flow of control in programs that use subroutine calls.

// mem[26] = RET; back to mem[11]

mem[26] = {8'b0000_0000};

mem[27] = {RET, 4'b0000};

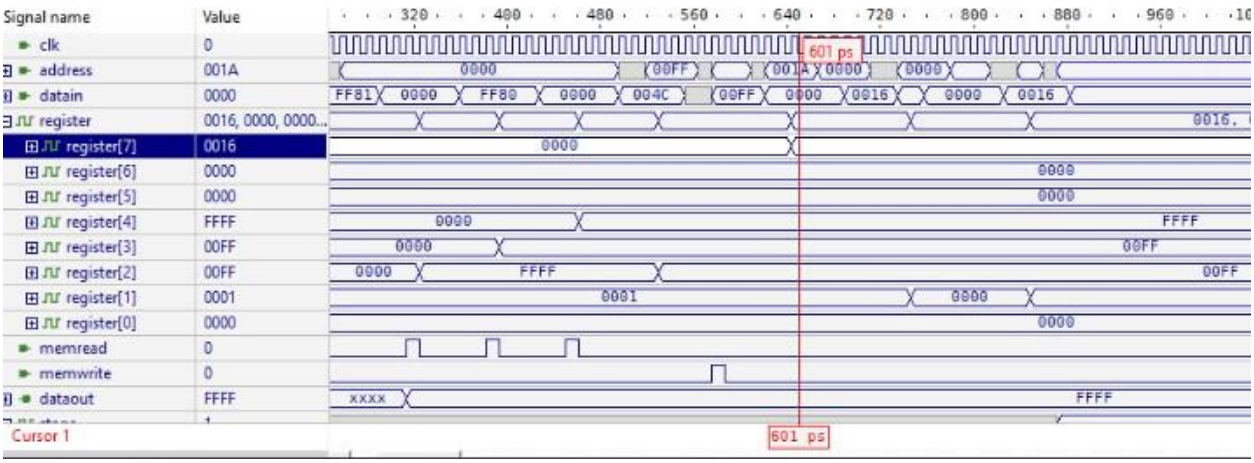


Figure 30: RET test

After the Return instruction, the Program Counter PC equals to R7

Extenders

In Verilog, extenders are typically used to manipulate the bit-width of signals. This involves either extending a smaller bit-width signal to a larger bit-width or truncating a larger bit-width signal to a smaller bit-width.

In our project we have :

1) 5bit extender

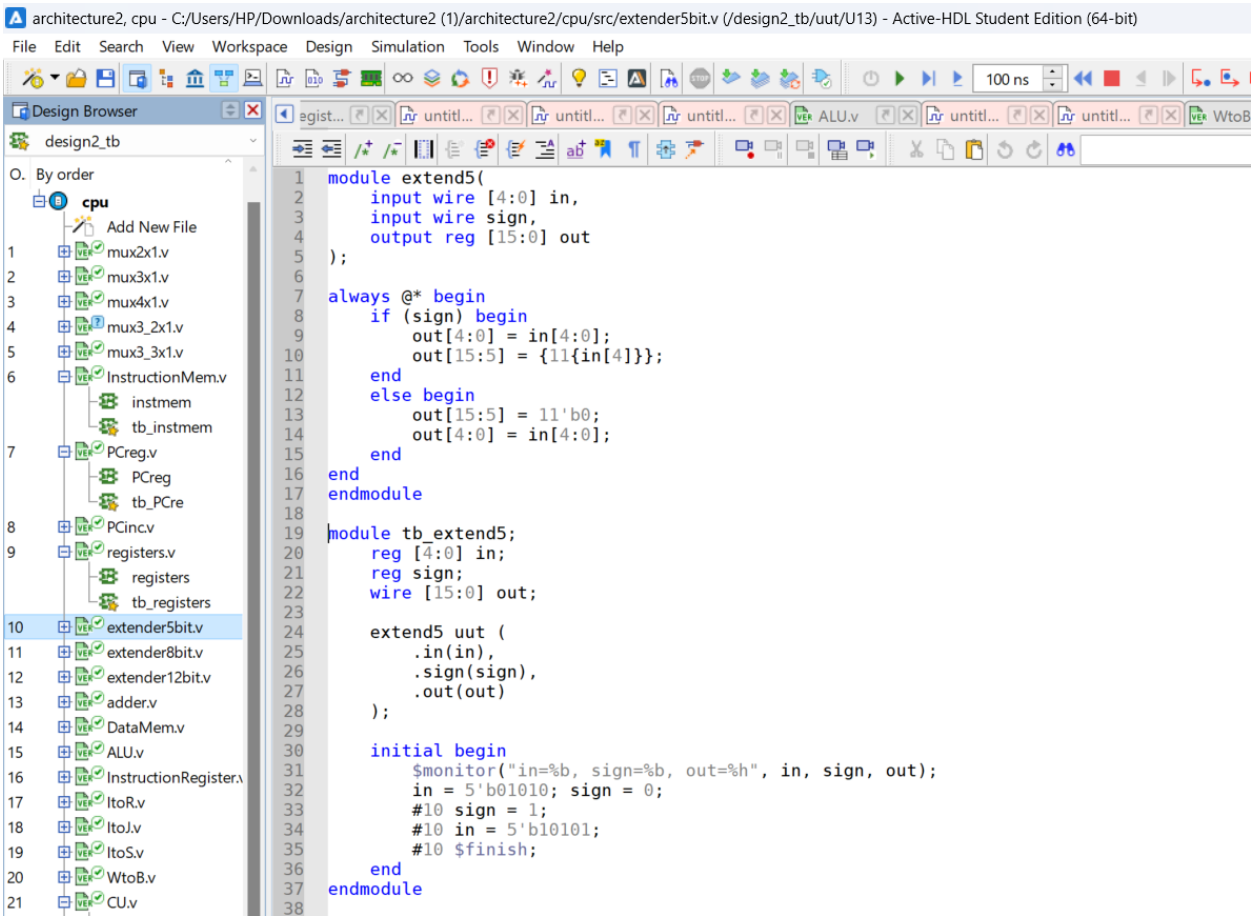
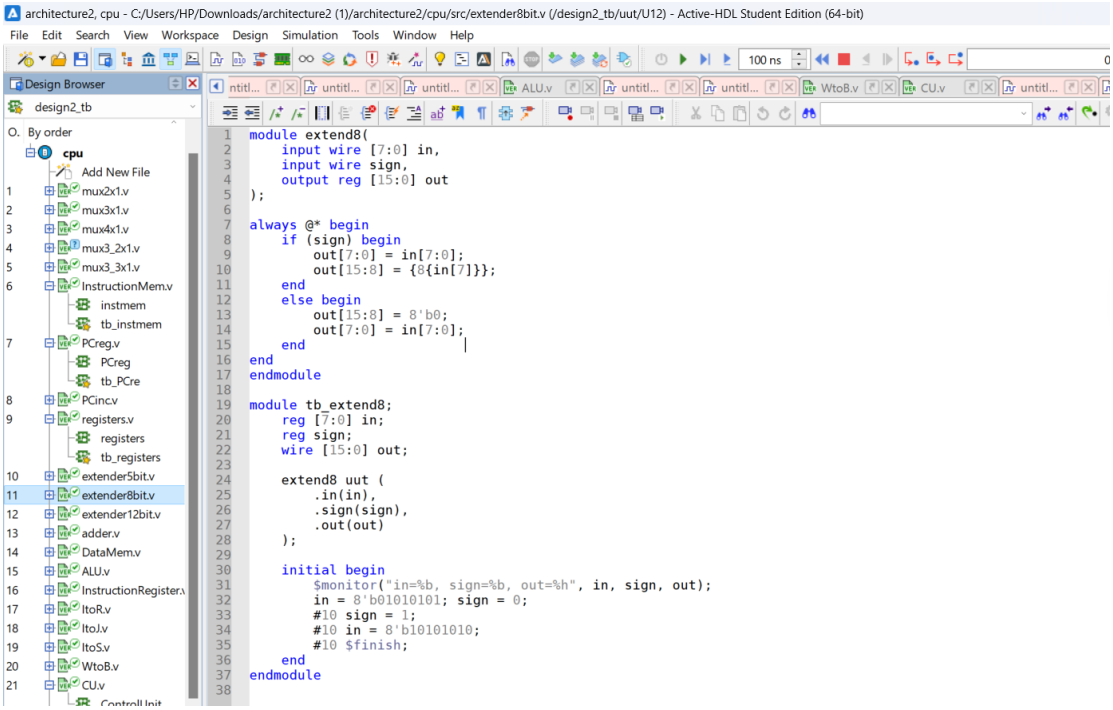
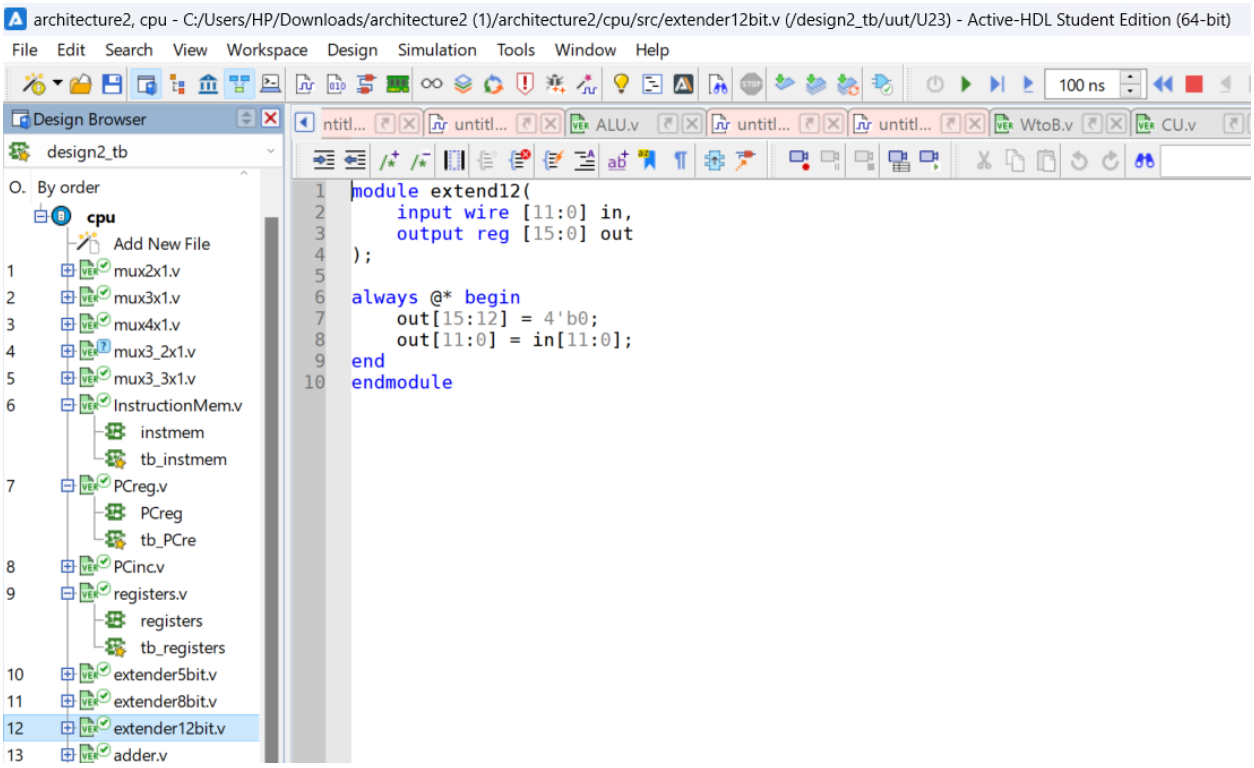


Figure 31: extenders

2) 8 bit extender



3) 12bit extender:



Adders:

Adders in Verilog are digital circuits designed to perform addition operations on binary numbers. They are fundamental components in digital design, used extensively in various applications including arithmetic operations in processors, signal processing, and communication systems. Verilog provides a convenient way to describe adders at different levels of abstraction, from basic full-adders to complex multi-bit adders.

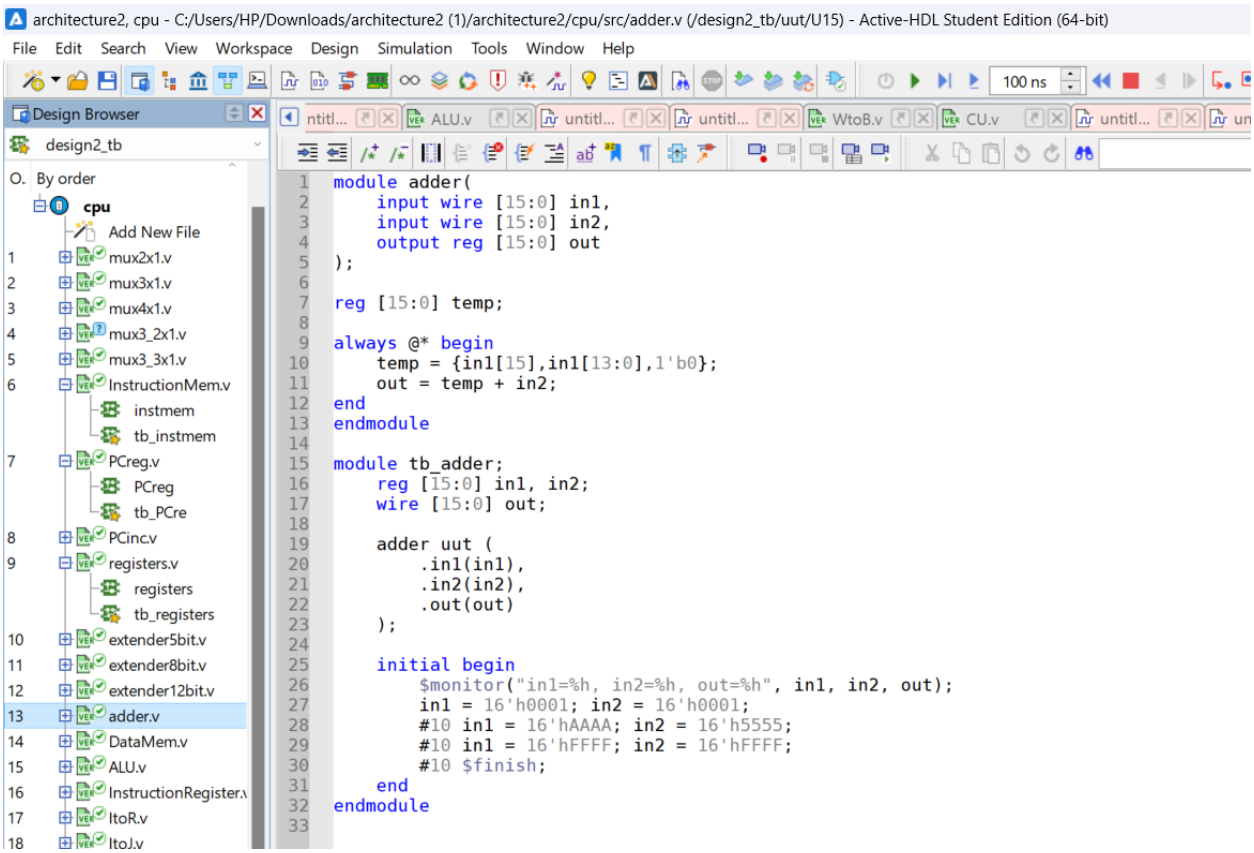


Figure 34: Adders

MUX:

A multiplexer (or mux) is a digital switch that selects one of several input signals and forwards the selected input to a single output line. The selection of the input signal is controlled by a set of select lines or control signals. Multiplexers are fundamental components in digital circuits, commonly used for routing data, managing signal flow, and reducing the number of data paths required for various operations.

MUX 2x1

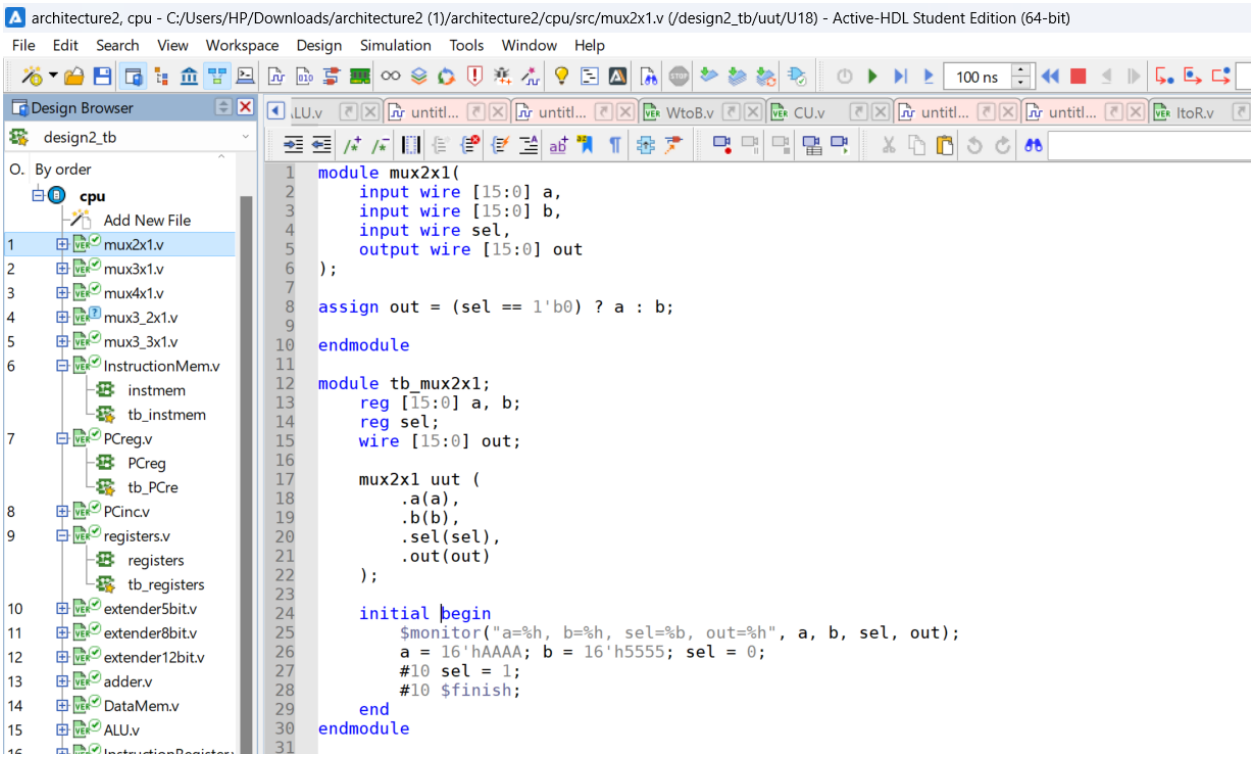
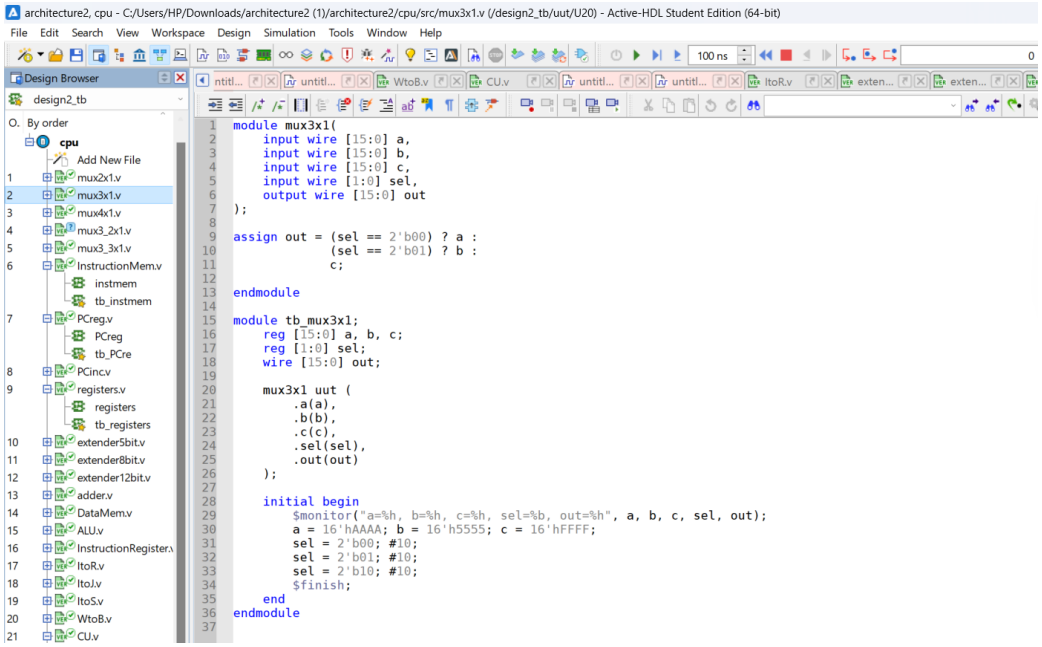
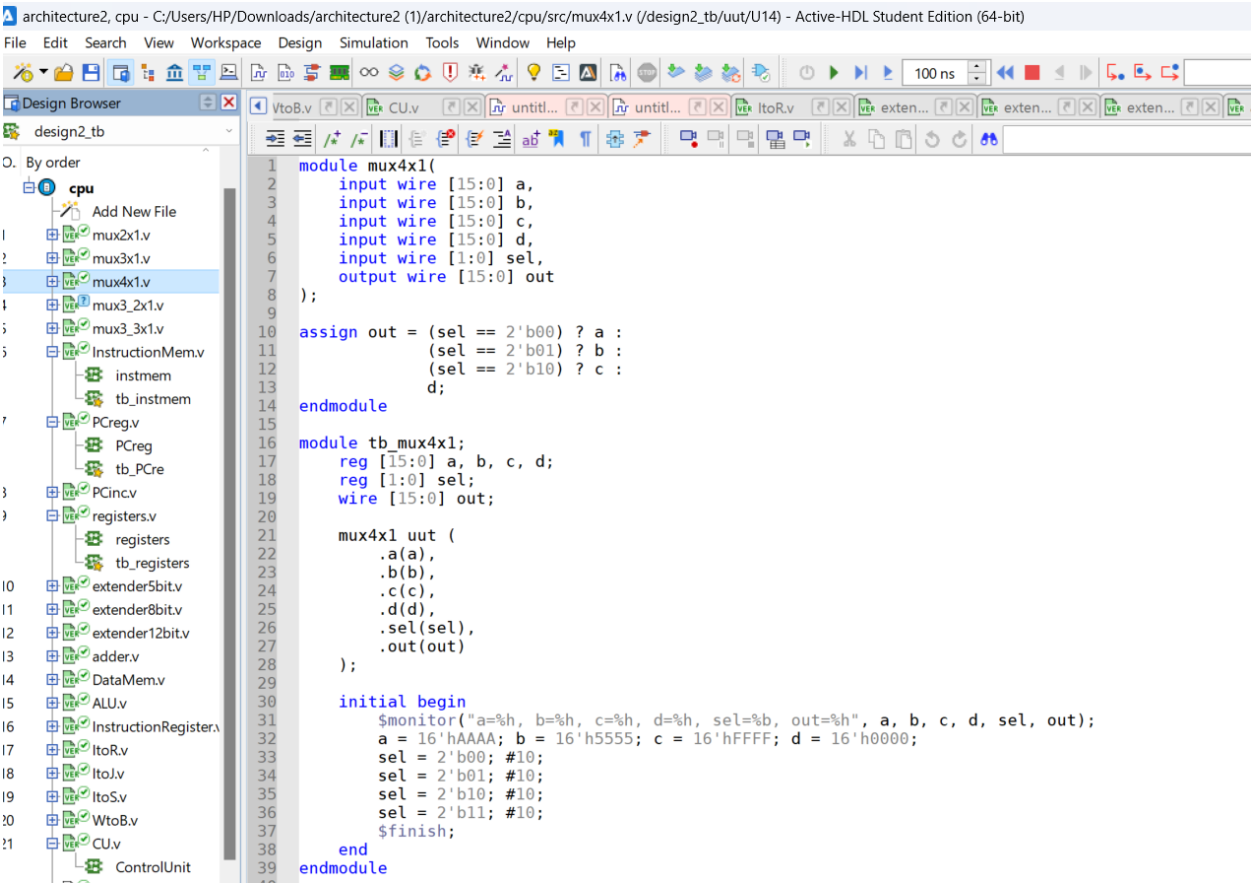


Figure 35: MUX

Mux 3x1



MUX 4x1



Instruction Register

The IR module is a simple and effective way to decode a 16-bit instruction into its constituent parts, which are critical for understanding and executing the instruction. The provided testbench verifies the functionality of the IR module by applying different instructions and monitoring the outputs, ensuring the module works as intended.

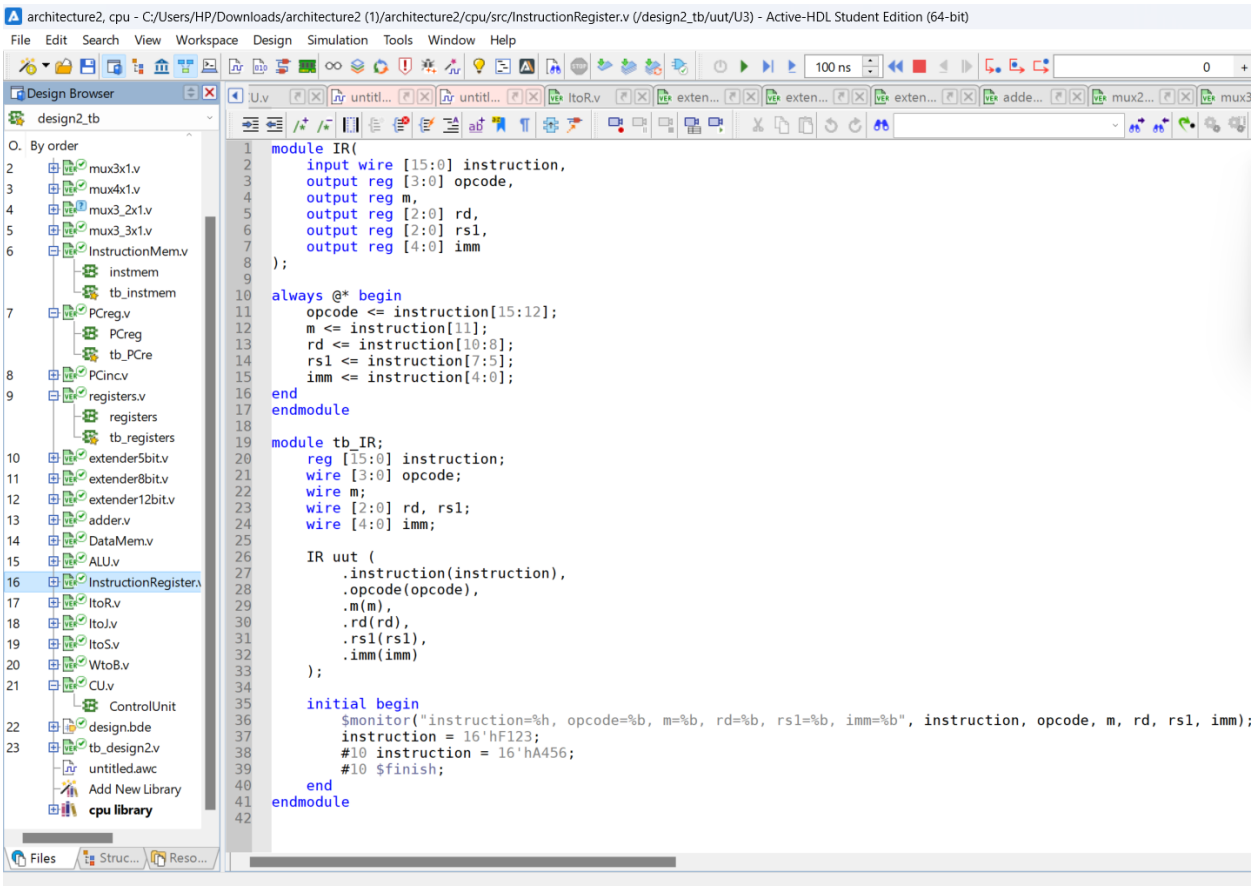


Figure 36: Instruction Register

Conclusion

In this project, we designed and verified a simple Multi Cycle RISC processor in Verilog, adhering to the specified 16-bit instruction and word size, with eight general-purpose registers and multiple instruction types. Our five-stage pipeline (fetch, decode, ALU, memory access, write back) was meticulously planned, focusing on the datapath and control path, including control signal generation. Through rigorous simulation and testing using a comprehensive testbench, we ensured the correctness and completeness of our design. This project enhanced our understanding of computer architecture and Verilog design, demonstrating the effectiveness of teamwork in achieving complex engineering objectives.

Appendices:

Data Memory

```
module datamem(
    input wire clk,
    input wire [15:0] address,
    input wire [15:0] datain,
    input wire memread,
    input wire memwrite,
    input wire wrByte,
    output reg [15:0] dataout
);

reg [7:0] mem[31:0];

initial begin
    for (int i = 0; i < 32; i = i + 1) begin
        mem[i] = 8'd0;
    end
end

always @(posedge clk) begin
    if (memwrite) begin
        if (wrByte) begin
            mem[address] <= datain[7:0];
        end
        else begin
            mem[address] <= datain[7:0];
            mem[address + 1] <= datain[15:8];
        end

    end
end

always @* begin
    if (memread) begin
        dataout <= {mem[address + 1], mem[addres
    end
end
endmodule

module tb_datamem;

  // inputs
  reg clk;
  reg [15:0] address;
  reg [15:0] datain;
  reg memread;
  reg memwrite;
  reg wrByte;

  // outputs
  wire [15:0] dataout;
```

Instruction Mem:

```
module instmem(
    input wire [15:0] inputPC,
    output reg [15:0] instruction
);

reg [7:0] mem[31:0];

parameter AND = 4'b0000;
parameter ADD = 4'b0001;
parameter SUB = 4'b0010;

//I type
parameter ADDI = 4'b0011;
parameter ANDI = 4'b0100;
parameter LW = 4'b0101;
parameter LBu = 4'b0110;
parameter LBs = 4'b0110;
parameter SW = 4'b0111;

//branch
parameter BGT = 4'b1000;
parameter BGTZ = 4'b1000;
parameter BLT = 4'b1001;
parameter BLTZ = 4'b1001;
parameter BEQ = 4'b1010;
parameter BEQZ = 4'b1010;
parameter BNE = 4'b1011;
parameter BNEZ = 4'b1011;

//J type
parameter JMP = 4'b1100;
parameter CALL = 4'b1101;
parameter RET = 4'b1110;

//S type
parameter Sv = 4'b1111;

initial begin
  // initialize the memory with the program

  // mem[0] = ADDI R0, R0, -1; R0 = 0 since it cannot be modified
  mem[0] = {3'b000, 5'b11111};
  mem[1] = {ADDI, 1'b0, 3'b000};

  // mem[2] = Sv R0, 16; mem[0] = 16'hFFFF
  mem[2] = {7'b1111111, 1'b0};
  mem[3] = {Sv, 3'b000, 1'b1};

  // mem[4] = BGTZ R1, 5; PC = PC + 3 = 5 x 2 = 10
  mem[4] = {3'b000, 5'b00011};
  mem[5] = {BGTZ, 1'b1, 3'b001};

```

```
52
53      // mem[6] = ADDI R1, R0, 1; R1 = 0 + 1 = 1
54      mem[6] = {3'b000, 5'b00001};
55      mem[7] = {ADDI, 1'b0, 3'b001};
56
57      // mem[8] = JMP 2
58      mem[8] = {8'b0000_0010};
59      mem[9] = {JMP, 4'b0000};
60
61      // mem[10] = LW R2, R0, 0; R2 = mem[0] = 16'hFFFF
62      mem[10] = {3'b000, 5'b00000};
63      mem[11] = {LW, 1'b0, 3'b010};
64
65      // mem[12] = LBu R3, R0, 0; R3 = mem[0] = 16'h00FF
66      mem[12] = {3'b000, 5'b00000};
67      mem[13] = {LBu, 1'b0, 3'b011};
68
69      // mem[14] = LBs R4, R0, 0; R4 = mem[0] = 16'hFFFF
70      mem[14] = {3'b000, 5'b00000};
71      mem[15] = {LBs, 1'b1, 3'b100};
72
73      // mem[16] = AND R2, R2, R3; R2 = R2 & R3 = 16'h00FF
74      mem[16] = {2'b10, 3'b011, 3'b000};
75      mem[17] = {AND, 3'b010, 1'b0};
76
77      // mem[18] = SW R2, R1, 1; mem[2] = 16'h00FF
78      mem[18] = {3'b001, 5'b00001};
79      mem[19] = {SW, 1'b0, 3'b010};
80
81      // mem[20] = CALL 13
82      mem[20] = {8'b0000_1101};
83      mem[21] = {CALL, 4'b0000};
84
85      // mem[22] = SUB R1, R1, R1; R1 = 1 - 1 = 0
86      mem[22] = {2'b01, 3'b001, 3'b000};
87      mem[23] = {SUB, 3'b001, 1'b0};
88
89      // mem[24] = JMP 14
90      mem[24] = {8'b0000_1110};
91      mem[25] = {JMP, 4'b0000};
92
93      // mem[26] = RET; back to mem[11]
94      mem[26] = {8'b0000_0000};
95      mem[27] = {RET, 4'b0000};
96
97      // mem[28] = ADDI R1, R1, -1; R1 = 0 - 1 = 16'hFFFF
98      mem[28] = {3'b001, 5'b11111};
99      mem[29] = {ADDI, 1'b0, 3'b001};
100
101 end
```

```
103
104 always @* begin
105     instruction[15:8] <= mem[inputPC[15:0] + 1];
106     instruction[7:0] <= mem[inputPC[15:0]];
107 end
108 endmodule
109
110 module tb_instmem;
111     reg [15:0] inputPC;
112     wire [15:0] instruction;
113
114     instmem uut (
115         .inputPC(inputPC),
116         .instruction(instruction)
117     );
118
119     initial begin
120         $monitor("inputPC=%h, instruction=%h", inputPC, instruction);
121         inputPC = 16'h0000;
122         #10 inputPC = 16'h0002;
123         #10 inputPC = 16'h0004;
124         #10 $finish;
125     end
126 endmodule
127
```

IR:

```
module IR(
    input wire [15:0] instruction,
    output reg [3:0] opcode,
    output reg m,
    output reg [2:0] rd,
    output reg [2:0] rs1,
    output reg [4:0] imm
);

always @* begin
    opcode <= instruction[15:12];
    m <= instruction[11];
    rd <= instruction[10:8];
    rs1 <= instruction[7:5];
    imm <= instruction[4:0];
end
endmodule

module tb_IR;
    reg [15:0] instruction;
    wire [3:0] opcode;
    wire m;
    wire [2:0] rd, rs1;
    wire [4:0] imm;

    IR uut (
        .instruction(instruction),
        .opcode(opcode),
        .m(m),
        .rd(rd),
        .rs1(rs1),
        .imm(imm)
    );

    initial begin
        $monitor("instruction=%h, opcode=%b, m=%b, rd=%b, rs1=%b, imm=%b", instruction, opcode, m, rd, rs1, imm);
        instruction = 16'hF123;
        #10 instruction = 16'hA456;
        #10 $finish;
    end
endmodule
```

Control unit

```
module ControlUnit(
    input wire clk,
    input wire [3:0] opcode,
    input wire m,
    //input wire takeBranch,
    output reg PCen,
    output reg memWrite,
    output reg memRead,
    output reg REGen,
    output reg sign5,
    output reg sign8,
    output reg signW2B,
    output reg [1:0] RA,
    output reg [1:0] RB,
    output reg [1:0] RW,
    output reg [1:0] PCsrc,
    output reg [1:0] BUSWsrc,
    output reg opAsrc,
    output reg [1:0] opBsrc,
    output reg dataInsrc,
    output reg [4:0] Function,
    output wire [2:0] R7
);

assign R7 = 3'b111;

reg [4:0] temp;

//R type
parameter AND = 5'b00000;
parameter ADD = 5'b00010;
parameter SUB = 5'b00100;

//I type
parameter ADDI = 5'b00110;
parameter ANDI = 5'b01000;
parameter LW = 5'b01010;
parameter LBu = 5'b01100;
parameter LBs = 5'b01101;
parameter SW = 5'b01110;

//branch
parameter BGT = 5'b10000;
parameter BGTZ = 5'b10001;
parameter BLT = 5'b10010;
parameter BLTZ = 5'b10011;
parameter BEQ = 5'b10100;
parameter BEQZ = 5'b10101;
parameter BNE = 5'b10110;
parameter BNEZ = 5'b10111;
```

```
    //J type
    parameter JMP = 5'b11000;
    parameter CALL = 5'b11010;
    parameter RET = 5'b11100;

    //S type
    parameter Sv = 5'b11110;

    //stages
    parameter start = 0;
    parameter RS = 1;
    parameter IF = 2;
    parameter ID = 3;
    parameter EX = 4;
    parameter MEM = 5;
    parameter WB = 6;
    reg [2:0] stage = start;

    always @(posedge clk) begin
        case (stage)
            start: begin
                PCen <= 0;
                memWrite <= 0;
                memRead <= 0;
                REGen <= 0;
                stage <= IF;
            end

            RS: begin
                PCen <= 1;
                memWrite <= 0;
                memRead <= 0;
                REGen <= 0;

                //if (takeBranch == 1): PCsrc <= 2'b01;
                //else: PCsrc <= 2'b00;

                stage <= IF;
            end

            IF: begin
                PCen <= 0;
                memWrite <= 0;
                memRead <= 0;
                REGen <= 0;
                stage <= ID;
            end
```

```
        ID: begin
            PCen <= 0;
            PCsrc <= 2'b00;
            memWrite <= 0;
            memRead <= 0;
            REGen <= 0;

            temp = {opcode,1'b0};

            if (opcode[3:0] == 4'b0110 || (opcode[3:0] > 4'b0111 && opcode[3:0] < 4'b1100)) begin
                temp[0] = m;
            end

            case(temp)
                AND: begin
                    REGen <= 0;
                    RA <= 2'b00;
                    RB <= 2'b00;
                    RW <= 2'b00;
                    BUSWsrc <= 2'b00;
                    opAsrc <= 0;
                    opBsrc <= 2'b00;
                    Function <= temp;
                    stage <= EX;
                end

                ADD: begin
                    REGen <= 0;
                    RA <= 2'b00;
                    RB <= 2'b00;
                    RW <= 2'b00;
                    BUSWsrc <= 2'b00;
                    opAsrc <= 0;
                    opBsrc <= 0;
                    Function <= temp;
                    stage <= EX;
                end

                SUB: begin
                    REGen <= 0;
                    RA <= 2'b00;
                    RB <= 2'b00;
                    RW <= 2'b00;
                    BUSWsrc <= 2'b00;
                    opAsrc <= 0;
                    opBsrc <= 2'b00;
                    Function <= temp;
                    stage <= EX;
                end
```

```
150                 ADDI: begin
151                     REGen <= 0;
152                     sign5 <= 1;
153                     RA <= 2'b01;
154                     RW <= 2'b01;
155                     BUSWsrc <= 2'b00;
156                     opAsrc <= 0;
157                     opBsrc <= 2'b01;
158                     Function <= temp;
159                     stage <= EX;
160                 end
161
162                 ANDI: begin
163                     REGen <= 0;
164                     sign5 <= 1;
165                     RA <= 2'b01;
166                     RW <= 2'b01;
167                     BUSWsrc <= 2'b00;
168                     opAsrc <= 0;
169                     opBsrc <= 2'b01;
170                     Function <= temp;
171                     stage <= EX;
172                 end
173
174                 LW: begin
175                     memRead <= 0;
176                     REGen <= 0;
177                     sign5 <= 1;
178                     RA <= 2'b01;
179                     RW <= 2'b01;
180                     BUSWsrc <= 2'b01;
181                     opAsrc <= 0;
182                     opBsrc <= 2'b01;
183                     Function <= temp;
184                     stage <= EX;
185                 end
186
187                 LBu: begin
188                     memRead <= 0;
189                     REGen <= 0;
190                     sign5 <= 1;
191                     signW2B <= 0;
192                     RA <= 2'b01;
193                     RW <= 2'b01;
194                     BUSWsrc <= 2'b10;
195                     opAsrc <= 0;
196                     opBsrc <= 2'b01;
197                     Function <= temp;
198                     stage <= EX;
```

```
200
201                 LBs: begin
202                     memRead <= 0;
203                     REGen <= 0;
204                     sign5 <= 1;
205                     signW2B <= 1;
206                     RA <= 2'b01;
207                     RW <= 2'b01;
208                     BUSWsrc <= 2'b10;
209                     opAsrc <= 0;
210                     opBsrc <= 2'b01;
211                     Function <= temp;
212                     stage <= EX;
213                 end
214
215                 SW: begin
216                     memWrite <= 0;
217                     REGen <= 0;
218                     sign5 <= 1;
219                     RA <= 2'b01;
220                     RB <= 2'b01;
221                     opAsrc <= 0;
222                     opBsrc <= 2'b01;
223                     dataInsrc <= 0;
224                     Function <= temp;
225                     stage <= EX;
226                 end
227
228                 BGT: begin
229                     sign5 <= 1;
230                     RA <= 2'b01;
231                     RB <= 2'b01;
232                     PCsrc <= 2'b01;
233                     opAsrc <= 0;
234                     opBsrc <= 2'b00;
235                     Function <= temp;
236                     stage <= EX;
237                 end
238
239                 BGTZ: begin
240                     sign5 <= 1;
241                     RA <= 2'b01;
242                     RB <= 2'b01;
243                     PCsrc <= 2'b01;
244                     opAsrc <= 0;
245                     opBsrc <= 2'b00;
246                     Function <= temp;
247                     stage <= EX;
248                 end
```

```
                    BLT: begin
                        sign5 <= 1;
                        RA <= 2'b01;
                        RB <= 2'b01;
                        PCsrc <= 2'b01;
                        opAsrc <= 0;
                        opBsrc <= 2'b00;
                        Function <= temp;
                        stage <= EX;
                    end

                    BLTZ: begin
                        sign5 <= 1;
                        RA <= 2'b01;
                        RB <= 2'b01;
                        PCsrc <= 2'b01;
                        opAsrc <= 0;
                        opBsrc <= 2'b00;
                        Function <= temp;
                        stage <= EX;
                    end

                    BEQ: begin
                        sign5 <= 1;
                        RA <= 2'b01;
                        RB <= 2'b01;
                        PCsrc <= 2'b01;
                        opAsrc <= 0;
                        opBsrc <= 2'b00;
                        Function <= temp;
                        stage <= EX;
                    end

                    BEQZ: begin
                        sign5 <= 1;
                        RA <= 2'b01;
                        RB <= 2'b01;
                        PCsrc <= 2'b01;
                        opAsrc <= 0;
                        opBsrc <= 2'b00;
                        Function <= temp;
                        stage <= EX;
                    end

                    BNE: begin
                        sign5 <= 1;
                        RA <= 2'b01;
                        RB <= 2'b01;
                        PCsrc <= 2'b01;
                        opAsrc <= 0;
                        opBsrc <= 2'b00;
```

```
                        Function <= temp;
                        stage <= EX;
                    end

                    BNEZ: begin
                        sign5 <= 1;
                        RA <= 2'b01;
                        RB <= 2'b01;
                        PCsrc <= 2'b01;
                        opAsrc <= 0;
                        opBsrc <= 2'b00;
                        Function <= temp;
                        stage <= EX;
                    end

                    JMP: begin
                        PCsrc <= 2'b10;
                        opAsrc <= 1;
                        opBsrc <= 2'b10;
                        Function <= temp;
                        stage <= EX;
                    end

                    CALL: begin
                        REGen <= 0;
                        RW <= 2'b10;
                        PCsrc <= 2'b10;
                        BUSWsrc <= 2'b11;
                        opAsrc <= 1;
                        opBsrc <= 2'b10;
                        Function <= temp;
                        stage <= EX;
                    end

                    RET: begin
                        RB <= 2'b10;
                        PCsrc <= 2'b11;
                        Function <= temp;
                        stage <= EX;
                    end

                    Sv: begin
                        memWrite <= 0;
                        sign8 <= 1;
                        RA <= 2'b10;
                        opAsrc <= 0;
                        dataInsrc <= 1;
                        Function <= temp;
                        stage <= EX;
                    end
                endcase
            end

            EX: begin
```

```
            end
        endcase
    end

    EX: begin
        case (temp)
            AND: begin
                REGen <= 1;
                stage <= WB;
            end

            ADD: begin
                REGen <= 1;
                stage <= WB;
            end

            SUB: begin
                REGen <= 1;
                stage <= WB;
            end

            ADDI: begin
                REGen <= 1;
                stage <= WB;
            end

            ANDI: begin
                REGen <= 1;
                stage <= WB;
            end

            LW: begin
                memRead <= 1;
                REGen <= 1;
                stage <= MEM;
            end

            LBu: begin
                memRead <= 1;
                REGen <= 1;
                stage <= MEM;
            end

            LBs: begin
                memRead <= 1;
                REGen <= 1;
                stage <= MEM;
            end

            SW: begin
                memWrite <= 1;
                stage <= MEM;
            end
```

```
                memWrite <= 1;
                stage <= MEM;
            end

            BGT: begin
                stage <= RS;
            end

            BGTZ: begin
                stage <= RS;
            end

            BLT: begin
                stage <= RS;
            end

            BLTZ: begin
                stage <= RS;
            end

            BEQ: begin
                stage <= RS;
            end

            BEQZ: begin
                stage <= RS;
            end

            BNE: begin
                stage <= RS;
            end

            BNEZ: begin
                stage <= RS;
            end

            JMP: begin
                stage <= RS;
            end

            CALL: begin
                REGen <= 1;
                stage <= WB;
            end

            RET: begin
                stage <= RS;
            end

            Sv: begin
                memWrite <= 1;
```

I type to J type Reg

```
module I2J(
    input wire m,
    input wire [2:0] rdOld,
    input wire [2:0] rs1Old,
    input wire [4:0] immOld,
    output reg [11:0] offset
);

always @* begin
    offset <= {m,rdOld,rs1Old,immOld};
end
endmodule

module tb_I2J;
    reg m;
    reg [2:0] rdOld, rs1Old;
    reg [4:0] immOld;
    wire [11:0] offset;

    I2J uut (
        .m(m),
        .rdOld(rdOld),
        .rs1Old(rs1Old),
        .immOld(immOld),
        .offset(offset)
    );

    initial begin
        $monitor("m=%b, rdOld=%b, rs1Old=%b, immOld=%b, offset=%b", m, rdOld, rs1Old, immOld, offset);
        m = 1; rdOld = 3'b101; rs1Old = 3'b010; immOld = 5'b11001;
        #10 rdOld = 3'b111; rs1Old = 3'b011; immOld = 5'b00110;
        #10 $finish;
    end
endmodule

```

I2R

```
module I2R(
    input wire m,
    input wire [2:0] rdOld,
    input wire [2:0] rs1Old,
    input wire [4:0] immOld,
    output reg [2:0] rd,
    output reg [2:0] rs1,
    output reg [2:0] rs2
);

always @* begin
    rd <= {m,rdOld[2:1]};
    rs1 <= {rdOld[0],rs1Old[2:1]};
    rs2 <= {rs1Old[0],immOld[4:3]};
end
endmodule

module tb_I2R;
    reg m;
    reg [2:0] rdOld, rs1Old;
    reg [4:0] immOld;
    wire [2:0] rd, rs1, rs2;

    I2R uut (
        .m(m),
        .rdOld(rdOld),
        .rs1Old(rs1Old),
        .immOld(immOld),
        .rd(rd),
        .rs1(rs1),
        .rs2(rs2)
    );

    initial begin
        $monitor("m=%b, rdOld=%b, rs1Old=%b, immOld=%b, rd=%b, rs1=%b, rs2=%b", m, rdOld, rs1Old, immOld, rd, rs1, rs2);
        m = 1; rdOld = 3'b101; rs1Old = 3'b010; immOld = 5'b11001;
        #10 rdOld = 3'b111; rs1Old = 3'b011; immOld = 5'b00110;
        #10 $finish;
    end
endmodule

```

I2S

```
module I2S(
    input wire m,
    input wire [2:0] rdOld,
    input wire [2:0] rs1Old,
    input wire [4:0] immOld,
    output reg [2:0] rs1,
    output reg [7:0] imm
);

always @* begin
    rs1 <= {m,rdOld[2:1]};
    imm <= {rdOld[0],rs1Old,immOld[4:1]};
end
endmodule

module tb_I2S;
    reg m;
    reg [2:0] rdOld, rs1Old;
    reg [4:0] immOld;
    wire [2:0] rs1;
    wire [7:0] imm;

    I2S uut (
        .m(m),
        .rdOld(rdOld),
        .rs1Old(rs1Old),
        .immOld(immOld),
        .rs1(rs1),
        .imm(imm)
    );

    initial begin
        $monitor("m=%b, rdOld=%b, rs1Old=%b, immOld=%b, rs1=%b, imm=%b", m, rdOld, rs1Old, immOld, rs1, imm);
        m = 1; rdOld = 3'b101; rs1Old = 3'b010; immOld = 5'b11001;
        #10 rdOld = 3'b111; rs1Old = 3'b011; immOld = 5'b00110;
        #10 $finish;
    end
endmodule

```

W2B

```
module W2B(
    input wire [15:0] in,
    input wire sign,
    output reg [15:0] out
);

always @* begin
    if (sign) begin
        out[7:0] = in[7:0];
        out[15:8] = {8{in[7]}};
    end
    else begin
        out[15:8] = 8'b0;
        out[7:0] = in[7:0];
    end
end
endmodule
```

PcInc

```
module PCinc(
    input wire [15:0] inputPC,
    output reg [15:0] outPC
);

always @* begin
    outPC = inputPC + 2;
end
endmodule

module tb_PCinc;
    reg [15:0] inputPC;
    wire [15:0] outPC;

    PCinc uut (
        .inputPC(inputPC),
        .outPC(outPC)
    );

    initial begin
        $monitor("inputPC=%h, outPC=%h", inputPC, outPC);
        inputPC = 16'h0000;
        #10 inputPC = 16'h0002;
        #10 inputPC = 16'h0004;
        #10 $finish;
    end
endmodule
```

PCReg

```
module PCreg(
    input clk,
    input wire en,
    input wire [15:0] in,
    output reg [15:0] out
);

reg [15:0] register;

initial begin
    register = 16'b0;
    out = register;
end

always @(posedge clk) begin
    if (en) begin
        register = in;
    end
end

always @* begin
    out = register;
end
endmodule

module tb_PCre;
    reg clk, en;
    reg [15:0] in;
    wire [15:0] out;

    PCreg uut (
        .clk(clk),
        .en(en),
        .in(in),
        .out(out)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        $monitor("clk=%b, en=%b, in=%h, out=%h", clk, en, in, out);
        en = 1; in = 16'h0001;
        #10 in = 16'h0002;
        #10 en = 0; in = 16'h0003;
        #10 en = 1; in = 16'h0004;
        #10 $finish;
    end
endmodule
```

ALU

```
module alu(
    input wire [15:0] operandA,
    input wire [15:0] operandB,
    input wire [4:0] Function,
    output reg [15:0] res,
    output reg takeBranch
);

reg [15:0] temp;

//R type
parameter AND = 5'b00000;
parameter ADD = 5'b00010;
parameter SUB = 5'b00100;

//I type
parameter ADDI = 5'b00110;
parameter ANDI = 5'b01000;
parameter LW = 5'b01010;
parameter LBu = 5'b01100;
parameter LBs = 5'b01101;
parameter SW = 5'b01110;

//branch
parameter BGT = 5'b10000;
parameter BGTZ = 5'b10001;
parameter BLT = 5'b10010;
parameter BLTZ = 5'b10011;
parameter BEQ = 5'b10100;
parameter BEQZ = 5'b10101;
parameter BNE = 5'b10110;
parameter BNEZ = 5'b10111;

//J type
parameter JMP = 5'b11000;
parameter CALL = 5'b11010;
parameter RET = 5'b11100;

//S type
parameter Sv = 5'b11110;

always @* begin
    res = 16'b0;
    takeBranch = 1'b0;

    case(Function)
        AND: res = operandA & operandB;
        ADD: res = operandA + operandB;
        SUB: res = operandA - operandB;
        BGT: takeBranch = (operandB > operandA) ? 1 : 0;
        BGTZ: takeBranch = (operandB > 0) ? 1 : 0;
        BLT: takeBranch = (operandB < operandA) ? 1 : 0;
        BLTZ: takeBranch = (operandB < 0) ? 1 : 0;
        BEQ: takeBranch = (operandB == operandA) ? 1 : 0;
        BEQZ: takeBranch = (operandB == 0) ? 1 : 0;
        BNE: takeBranch = (operandB == operandA) ? 0 : 1;
        BNEZ: takeBranch = (operandB == 0) ? 0 : 1;

        JMP: begin
            temp = operandB << 1;
            res = {operandA[15:12],temp[11:0]};
        end

        CALL: begin
            temp = operandB << 1;
            res = {operandA[15:12],temp[11:0]};
        end

        RET: res = operandB;

        Sv: res = operandA;
    endcase
end
endmodule

module tb_alu;
    reg [15:0] operandA, operandB;
    reg [4:0] Function;
    wire [15:0] res;
    wire zeroFlag, negativeFlag;

    alu uut (
        .operandA(operandA),
        .operandB(operandB),
        .Function(Function),
        .res(res),
        .takeBranch(takeBranch)
    );

    initial begin
        $monitor("operandA=%h, operandB=%h, Function=%b, res=%h, takeBranch=%b", operandA, operandB, Function, res, takeBranch);
        operandA = 16'h0001; operandB = 16'h0001; Function = 5'b00010; // ADD
        #10 Function = 5'b00100; // SUB
        #10 Function = 5'b00000; // AND
        #10 Function = 5'b10100; // BEQ
        #10 $finish;
    end
endmodule
```

Registers

```
module registers(
    input clk,
    input wire en,
    input wire [2:0] RA,
    input wire [2:0] RB,
    input wire [2:0] RW,
    input wire [15:0] BUSW,
    output reg [15:0] BUSA,
    output reg [15:0] BUSB
);

reg [15:0] register [7:0];
integer i;

initial begin
    for (i = 0; i < 8; i = i + 1) begin
        register[i] = 16'd0;
    end
end

always @(posedge clk) begin
    if (en && (RW != 3'b000)) begin
        register[RW] <= BUSW;
    end

    BUSA <= register[RA];
    BUSB <= register[RB];
end
endmodule

module tb_registers;
    reg clk, en;
    reg [2:0] RA, RB, RW;
    reg [15:0] BUSW;
    wire [15:0] BUSA, BUSB;

    registers uut (
        .clk(clk),
        .en(en),
        .RA(RA),
        .RB(RB),
        .RW(RW),
        .BUSW(BUSW),
        .BUSA(BUSA),
        .BUSB(BUSB)
    );

    initial begin
        clk = 0;
        forever #5 clk = ~clk;
    end

    initial begin
        $monitor("clk=%b, en=%b, RA=%b, RB=%b, RW=%b, BUSW=%h, BUSA=%h, BUSB=%h, reg[RA]=%h, reg[RB]=%h, reg[RW]=%h", clk, en, RA, RB, RW, BUSW, BUSA, BUSB, uut.register[RA], uut.register[RB], uut.register[RW]);
        en = 1; RW = 3'b000; BUSW = 16'hAAAA;
        #10 RA = 3'b001; RB = 3'b000;
        #10 RW = 3'b010; BUSW = 16'h5555;
        #10 RA = 3'b010; RB = 3'b001;
        #10 $finish;
    end
endmodule
```