



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

Abstract.....	2
Introduction.....	4
Datapath.....	5
Design Implementation.....	6
Instruction Fetch.....	7
Program Counter (PC)	7
Pc incremental:.....	8
Instruction Memory.....	9
Instruction Decode	11
Register File	11
Instruction Execute.....	12
ALU	12
Data Memory.....	13
Writeback	14
Control Unit.....	14
Control signals Table.....	17
Discussion of Control signals:	20
Lbu:	27
LBs.....	28
AND:.....	29
SW:	30
CALL:	31
SUB:	32
JUMP:	33
RET:	34
Extenders	35
Instruction Register	40
Control Unit Boolean:.....	18
Conclusion	41
Appendices:	42

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

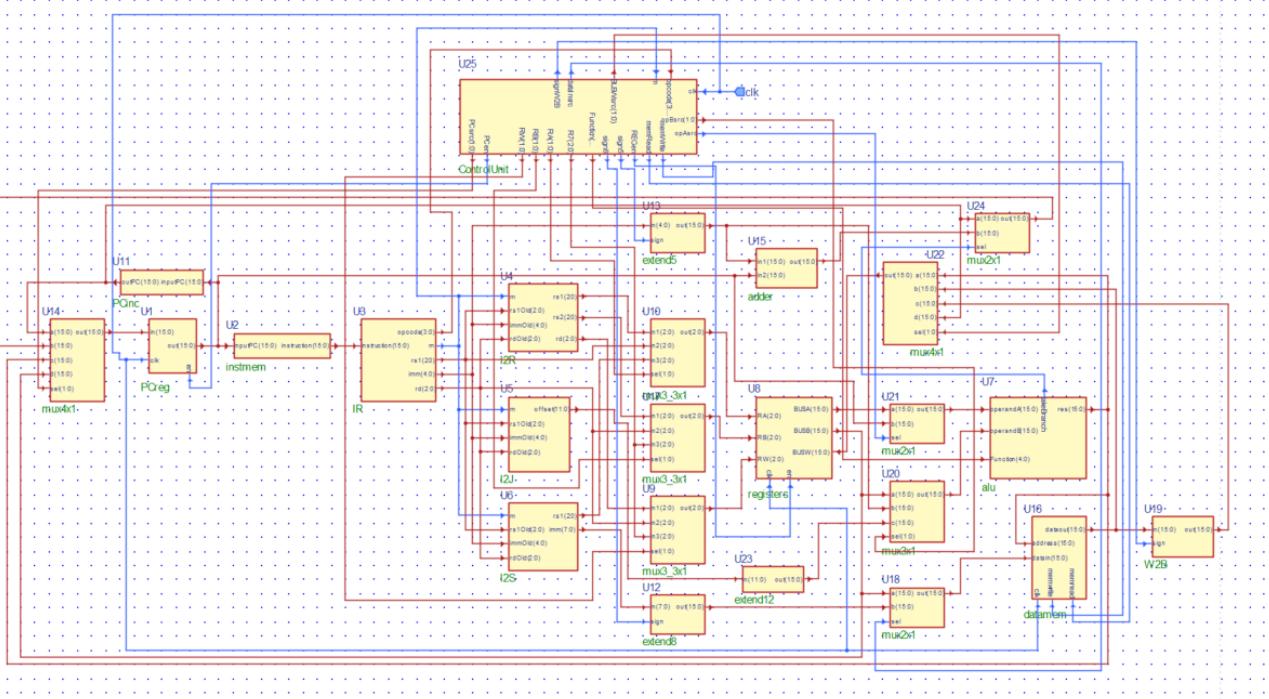


Figure 1: Datapath Diagram

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

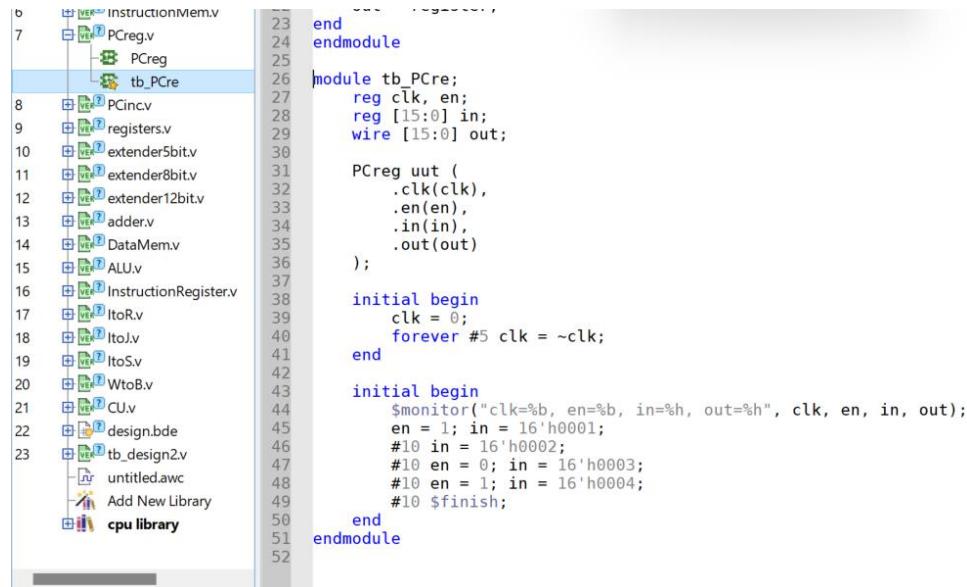
the data we are storing in "REGW" for example ALU results, loading word or byte from memory, and the output of PC incremented.

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.



```
0 InstructionMem.v
1
2 PCreg.v
3 PCregr
4
5 tb_PCre
6
7 PCincv
8 registers.v
9 extender5bit.v
10 extender8bit.v
11 extender12bit.v
12 adder.v
13 DataMem.v
14 ALU.v
15
16 InstructionRegister.v
17 ItoR.v
18 ItoJ.v
19 ItoS.v
20 WtoB.v
21 CU.v
22 design.bde
23 tb_design2.v
24 untitled.awc
25 Add New Library
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
```

```
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
```

```
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
```

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.

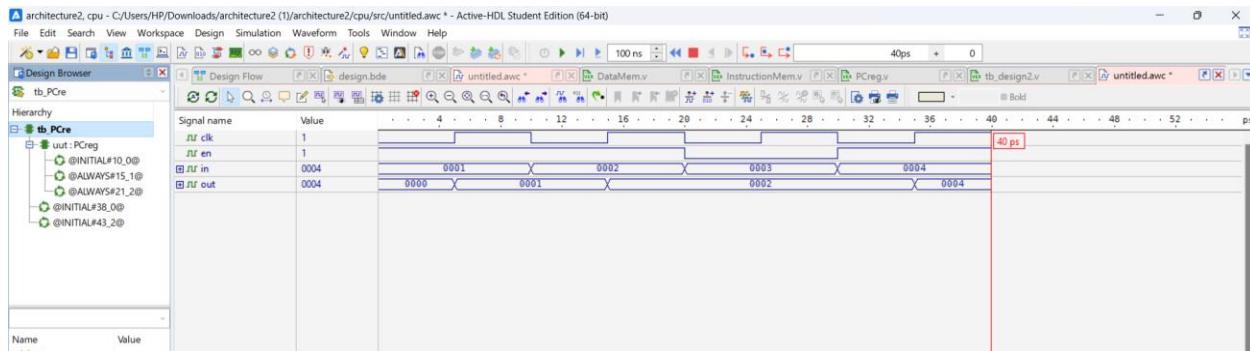


Figure 4: PC Testbench Waveform

Pc incremental:

```

architecture2,cpu - C:/Users/HP/Downloads/architecture2 (1)/architecture2/cpu/src/PCinc.v - Active-HDL Student Edition (64-bit)
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser Design Flow untitled.awc DataMem.v InstructionM... PCReg.v tb_design2.v tb_design2.v
tb_PCre
O. Unsorted
    Workspace 'architecture2
        cpu
            Add New File
                mux2x1.v
                mux3x1.v
                mux4x1.v
                mux3_2x1.v
                mux3_3x1.v
                PCReg.v
                tb_PCre
                PCinc.v
                registers.v
                extender5bit.v
                extender8bit.v
                extender12bit.v
                adder.v
                DataMem.v
                ALU.v
                InstructionRegister.v
                ItoR.v
                ItoJ.v
1 module PCinc(
2     input wire [15:0] inputPC,
3     output reg [15:0] outPC
4 );
5
6 always @* begin
7     outPC = inputPC + 2;
8 end
9 endmodule
10
11 module tb_PCinc;
12     reg [15:0] inputPC;
13     wire [15:0] outPC;
14
15     PCinc uut (
16         .inputPC(inputPC),
17         .outPC(outPC)
18     );
19
20 initial begin
21     $monitor("inputPC=%h, outPC=%h", inputPC, outPC);
22     inputPC = 16'h0000;
23     #10 inputPC = 16'h0002;
24     #10 inputPC = 16'h0004;
25     #10 $finish;
26 end
27
28 endmodule

```

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.

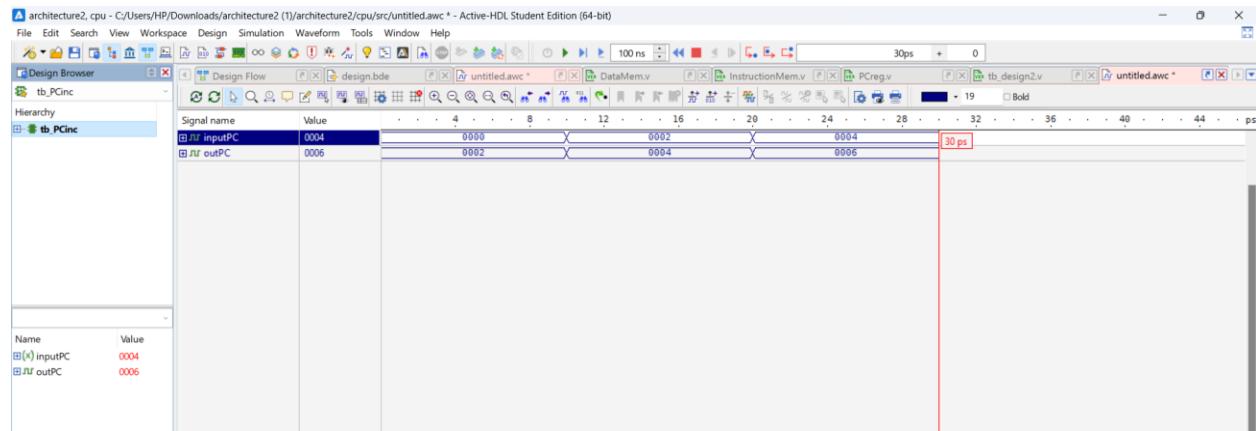


Figure 5: PC incremental Testbench Waveform

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.

O. By order

```

2   mux3x1.v
3   mux4x1.v
4   mux3_2x1.v
5   mux3_3x1.v
6   InstructionMem.v
7   Creg.v
8   PCincv
9   registers.v
10  extender5bit.v
11  extender8bit.v
12  extender12bit.v
13  adder.v
14  DataMem.v
15  ALU.v
16  InstructionRegister.v
17  ItoR.v
18  ItoJ.v
19  ItoSv
20  WtoB.v
21  CU.v
22  design.bde
23  tb_design2.v
untitled.awc
Add New Library
cpu library

```

```

37 initial begin
38     // initialize the memory with the program
39
40     // mem[0] = ADDI R0, R0, -1; R0 = 0 since it cannot be modified
41     mem[0] = {3'b000, 5'b11111};
42     mem[1] = {ADDI, 1'b0, 3'b000};
43
44     // mem[2] = Sv R0, 16; mem[0] = 16'hFFFF
45     mem[2] = {7'b1111111, 1'b0};
46     mem[3] = {Sv, 3'b000, 1'b1};
47
48     // mem[4] = BGTZ R1, 5; PC = PC + 3 = 5 x 2 = 10
49     mem[4] = {3'b000, 5'b00011};
50     mem[5] = {BGTZ, 1'b1, 3'b001};
51
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'b0, 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'h0000};

94     mem[20] = 16'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
102
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

```

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.

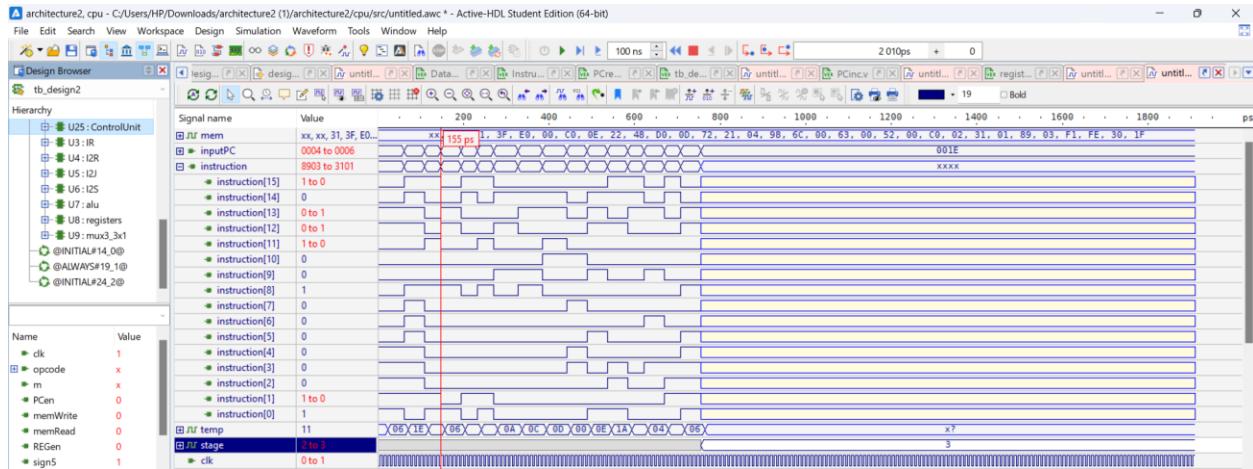


Figure 7: Instruction Memory Testbench Waveform

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.



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.

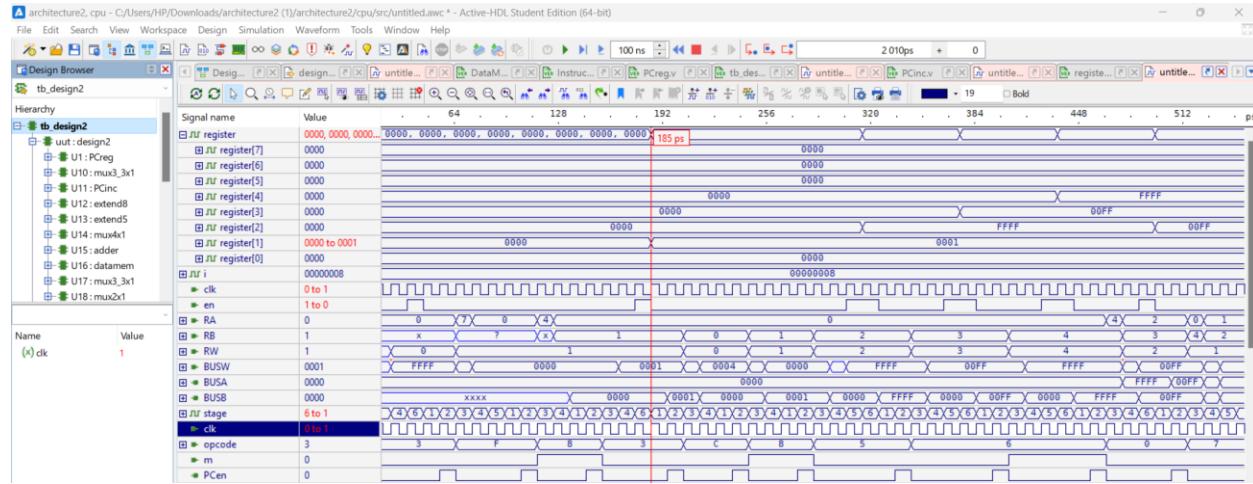


Figure 9: Register File Testbench Waveform

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.

```

83
84 module tb_alu;
85   reg [15:0] operandA, operandB;
86   reg [4:0] Function;
87   wire [15:0] res;
88   wire zeroFlag, negativeFlag;
89
90   alu uut (
91     .operandA(operandA),
92     .operandB(operandB),
93     .Function(Function),
94     .res(res),
95     .takeBranch(takeBranch)
96   );
97
98   initial begin
99     $monitor("operandA=%h, operandB=%h, Function=%b, res=%h, takeBranch=%b", operandA, operandB, Function, res, takeBranch);
100    operandA = 16'h0001; operandB = 16'h0001; Function = 5'b00010; // ADD
101    #10 Function = 5'b01100; // SUB
102    #10 Function = 5'b00000; // AND
103    #10 Function = 5'b10100; // BEQ
104    #10 $finish;
105
106 endmodule
107

```

Figure 10: ALU Testbench

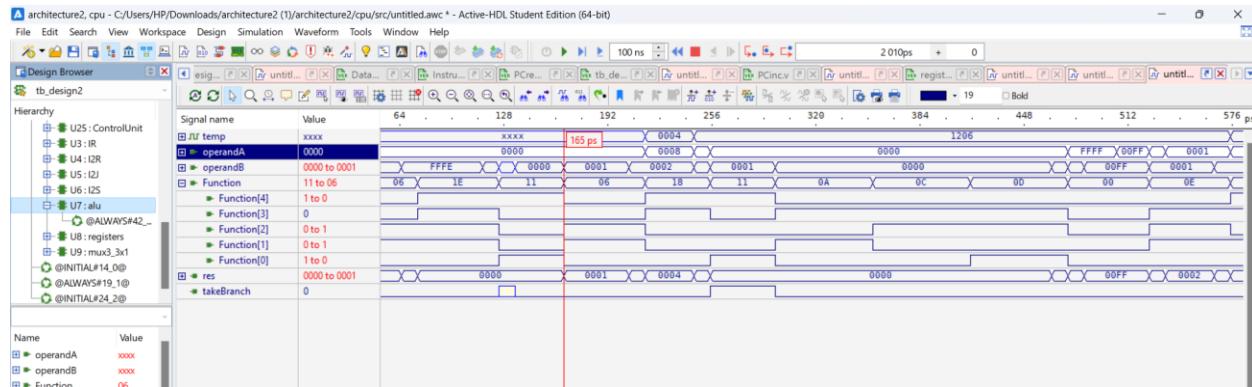


Figure 11: ALU Testbench Waveform

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.

```
File Structure Resources
7   module tb_datamem;
8     instmem;
9     tb_instmem;
10    PCregv;
11    PCreg;
12    PCincv;
13    registers;
14    extenderSbitv;
15    extender8bitv;
16    extender12bitv;
17    adderv;
18    DataMemory;
19    ALUv;
20    InstructionRegisters;
21    ItoRv;
22    ItoLv;
23    ItoSv;
24    WtoBv;
25    CUv;
26    design.bde;
27    tb_design2v;
28    ...
29
30   module tb_datamem;
31     reg clk;
32     reg [15:0] address, datain;
33     reg memread, memwrite;
34     wire [15:0] dataout;
35
36   datamem uut (
37     .clk(clk),
38     .address(address),
39     .datain(datain),
40     .memread(memread),
41     .memwrite(memwrite),
42     .dataout(dataout)
43 );
44
45   initial begin
46     clk = 0;
47     forever #5 clk = ~clk;
48   end
49
50   initial begin
51     $monitor("clk=%b, address=%h, datain=%h, memread=%b, memwrite=%b, dataout=%h, mem[addr]=%h", clk, address, datain, memread, memwrite, dataout, uut.mem[address]);
52     address = 16'h0000; datain = 16'hAAAA; memwrite = 0; memread = 0;
53     #10 memwrite = 0; memread = 1;
54     #10 address = 16'h0001; datain = 16'h5555; memwrite = 1; memread = 0;
55     #10 memwrite = 0; memread = 0;
56     #10 $finish;
57   end
58
59 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.

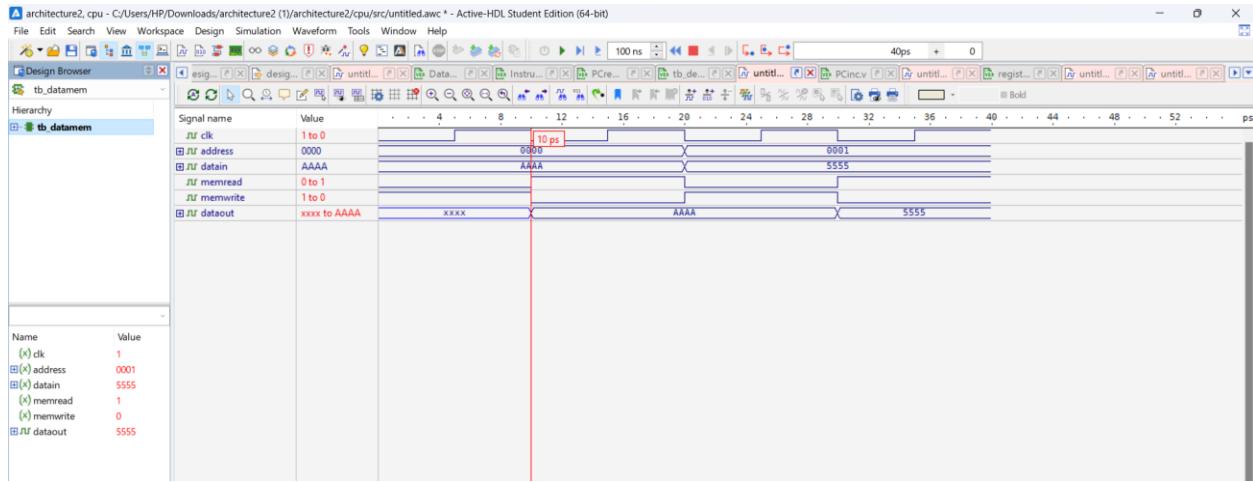


Figure 13: Data Memory Testbench Waveform

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.

A architecture2, cpu - C:/Users/HP/Downloads/architecture2 (1)/architecture2/cpu/src/CU.v - Active-HDL Student Edition (64-bit)

```

File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
design2_tb
O. Unsorted
  workspace 'architectur
    cpu
      Add New File
      mux2x1.v
      mux3x1.v
      mux4x1.v
      mux3_2x1.v
      mux3_3x1.v
      InstructionMemv
        instmem
          tb_instmem
      PCregv
        PCreg
          tb_PCre
      PCincv
      registers.v
        registers
          tb_registers
      extender5bitv
      extender8bitv
      extender12bitv
      adder.v
      DataMemv
      ALU.v
      InstructionRegister.v
      ItoRv
      ItoJ.v
      ItoSv
      WtoB.v
      CU.v
        ControlUnit
        design.bde
      Files Struct... Reso...
      1 assign R7 = 3'b111;
      2 reg [4:0] temp;
      3 //R type
      4 parameter AND = 5'b00000;
      5 parameter ADD = 5'b000010;
      6 parameter SUB = 5'b00100;
      7
      8 //I type
      9 parameter ADDI = 5'b00110;
      10 parameter ANDI = 5'b01000;
      11 parameter LW = 5'b01010;
      12 parameter LBu = 5'b01100;
      13 parameter LBs = 5'b01101;
      14 parameter SW = 5'b01110;
      15
      16 //J type
      17 parameter BEQ = 5'b10100;
      18 parameter BEQZ = 5'b10101;
      19 parameter BNE = 5'b10110;
      20 parameter BNEZ = 5'b10111;
      21
      22 //S type
      23 parameter Sv = 5'b11110;
      24
      25 //stages
      26 parameter start = 0;
      27 parameter RS = 1;
      28 parameter IF = 2;
      29 parameter ID = 3;
      30 parameter EX = 4;
      31 parameter MEM = 5;
      32 parameter WB = 6;
      33 reg [2:0] stage = start;
      34
      35 always @(posedge clk) begin
      36   case (stage)
      37     start: begin
      38       PCen <= 0;
      39       memWrite <= 0;
      40       memRead <= 0;
      41       REGen <= 0;
      42       stage <= IF;
      43     end
      44     RS: begin
      45       PCen <= 1;
      46       memWrite <= 0;
      47       memRead <= 0;
      48       REGen <= 0;
      49       //if (takeBranch == 1); PCsrc <= 2'b01;
      50       //else; PCsrc <= 2'b00;
      51       stage <= IF;
      52     end
      53     IF: begin
      54       PCen <= 0;
      55       memWrite <= 0;
      56       memRead <= 0;
      57       REGen <= 0;
      58       stage <= ID;
      59     end
      60     ID: begin
      61       PCen <= 0;
      62       PCsrc <= 2'b00;
      63       memWrite <= 0;
      64       memRead <= 0;
      65       REGen <= 0;
      66       stage <= EX;
      67     end
      68     EX: begin
      69       PCen <= 0;
      70       PCsrc <= 2'b01;
      71       memWrite <= 1;
      72       memRead <= 0;
      73       REGen <= 0;
      74       stage <= MEM;
      75     end
      76     MEM: begin
      77       PCen <= 0;
      78       PCsrc <= 2'b00;
      79       memWrite <= 0;
      80       memRead <= 1;
      81       REGen <= 0;
      82       stage <= WB;
      83     end
      84     WB: begin
      85       PCen <= 0;
      86       PCsrc <= 2'b00;
      87       memWrite <= 0;
      88       memRead <= 0;
      89       REGen <= 0;
      90       stage <= start;
      91     end
      92   endcase
      93 end
      94
      95 end
      96
      97
      98
      99
      100
      101
      102
      103
      104
      105
      106
      107
      108
      109
      110
      111
      112
      113
      114
      115
      116
      117
      118
      119
      120
      121
      122
      123
      124
      125
      126
      127
      128
      129
      130
      131
      132
      133
      134
      135
      136
      137
      138
      139
      140
      141
      142
      143
      144
      145
      146
      147
      148
      149
      150
      151
      152
      153
      154
      155
      156
      157
      158
      159
      160
      161
      162
      163
      164
      165
      166
      167
      168
      169
      170
      171
      172
      173
      174
      175
      176
      177
      178
      179
      180
      181
      182
      183
      184
      185
      186
      187
      188
      189
      190
      191
      192
      193
      194
      195
      196
      197
      198
      199
      200
      201
      202
      203
      204
      205
      206
      207
      208
      209
      210
      211
      212
      213
      214
      215
      216
      217
      218
      219
      220
      221
      222
      223
      224
      225
      226
      227
      228
      229
      230
      231
      232
      233
      234
      235
      236
      237
      238
      239
      240
      241
      242
      243
      244
      245
      246
      247
      248
      249
      250
      251
      252
      253
      254
      255
      256
      257
      258
      259
      260
      261
      262
      263
      264
      265
      266
      267
      268
      269
      270
      271
      272
      273
      274
      275
      276
      277
      278
      279
      280
      281
      282
      283
      284
      285
      286
      287
      288
      289
      290
      291
      292
      293
      294
      295
      296
      297
      298
      299
      300
      301
      302
      303
      304
      305
      306
      307
      308
      309
      310
      311
      312
      313
      314
      315
      316
      317
      318
      319
      320
      321
      322
      323
      324
      325
      326
      327
      328
      329
      330
      331
      332
      333
      334
      335
      336
      337
      338
      339
      340
      341
      342
      343
      344
      345
      346
      347
      348
      349
      350
      351
      352
      353
      354
      355
      356
      357
      358
      359
      360
      361
      362
      363
      364
      365
      366
      367
      368
      369
      370
      371
      372
      373
      374
      375
      376
      377
      378
      379
      380
      381
      382
      383
      384
      385
      386
      387
      388
      389
      390
      391
      392
      393
      394
      395
      396
      397
      398
      399
      400
      401
      402
      403
      404
      405
      406
      407
      408
      409
      410
      411
      412
      413
      414
      415
      416
      417
      418
      419
      420
      421
      422
      423
      424
      425
      426
      427
      428
      429
      430
      431
      432
      433
      434
      435
      436
      437
      438
      439
      440
      441
      442
      443
      444
      445
      446
      447
      448
      449
      450
      451
      452
      453
      454
      455
      456
      457
      458
      459
      460
      461
      462
      463
      464
      465
      466
      467
      468
      469
      470
      471
      472
      473
      474
      475
      476
      477
      478
      479
      480
      481
      482
      483
      484
      485
      486
      487
      488
      489
      490
      491
      492
      493
      494
      495
      496
      497
      498
      499
      500
      501
      502
      503
      504
      505
      506
      507
      508
      509
      510
      511
      512
      513
      514
      515
      516
      517
      518
      519
      520
      521
      522
      523
      524
      525
      526
      527
      528
      529
      530
      531
      532
      533
      534
      535
      536
      537
      538
      539
      540
      541
      542
      543
      544
      545
      546
      547
      548
      549
      550
      551
      552
      553
      554
      555
      556
      557
      558
      559
      560
      561
      562
      563
      564
      565
      566
      567
      568
      569
      570
      571
      572
      573
      574
      575
      576
      577
      578
      579
      580
      581
      582
      583
      584
      585
      586
      587
      588
      589
      590
      591
      592
      593
      594
      595
      596
      597
      598
      599
      600
      601
      602
      603
      604
      605
      606
      607
      608
      609
      610
      611
      612
      613
      614
      615
      616
      617
      618
      619
      620
      621
      622
      623
      624
      625
      626
      627
      628
      629
      630
      631
      632
      633
      634
      635
      636
      637
      638
      639
      640
      641
      642
      643
      644
      645
      646
      647
      648
      649
      650
      651
      652
      653
      654
      655
      656
      657
      658
      659
      660
      661
      662
      663
      664
      665
      666
      667
      668
      669
      670
      671
      672
      673
      674
      675
      676
      677
      678
      679
      680
      681
      682
      683
      684
      685
      686
      687
      688
      689
      690
      691
      692
      693
      694
      695
      696
      697
      698
      699
      700
      701
      702
      703
      704
      705
      706
      707
      708
      709
      710
      711
      712
      713
      714
      715
      716
      717
      718
      719
      720
      721
      722
      723
      724
      725
      726
      727
      728
      729
      730
      731
      732
      733
      734
      735
      736
      737
      738
      739
      740
      741
      742
      743
      744
      745
      746
      747
      748
      749
      750
      751
      752
      753
      754
      755
      756
      757
      758
      759
      760
      761
      762
      763
      764
      765
      766
      767
      768
      769
      770
      771
      772
      773
      774
      775
      776
      777
      778
      779
      780
      781
      782
      783
      784
      785
      786
      787
      788
      789
      790
      791
      792
      793
      794
      795
      796
      797
      798
      799
      800
      801
      802
      803
      804
      805
      806
      807
      808
      809
      810
      811
      812
      813
      814
      815
      816
      817
      818
      819
      820
      821
      822
      823
      824
      825
      826
      827
      828
      829
      830
      831
      832
      833
      834
      835
      836
      837
      838
      839
      840
      841
      842
      843
      844
      845
      846
      847
      848
      849
      850
      851
      852
      853
      854
      855
      856
      857
      858
      859
      860
      861
      862
      863
      864
      865
      866
      867
      868
      869
      870
      871
      872
      873
      874
      875
      876
      877
      878
      879
      880
      881
      882
      883
      884
      885
      886
      887
      888
      889
      890
      891
      892
      893
      894
      895
      896
      897
      898
      899
      900
      901
      902
      903
      904
      905
      906
      907
      908
      909
      910
      911
      912
      913
      914
      915
      916
      917
      918
      919
      920
      921
      922
      923
      924
      925
      926
      927
      928
      929
      930
      931
      932
      933
      934
      935
      936
      937
      938
      939
      940
      941
      942
      943
      944
      945
      946
      947
      948
      949
      950
      951
      952
      953
      954
      955
      956
      957
      958
      959
      960
      961
      962
      963
      964
      965
      966
      967
      968
      969
      970
      971
      972
      973
      974
      975
      976
      977
      978
      979
      980
      981
      982
      983
      984
      985
      986
      987
      988
      989
      990
      991
      992
      993
      994
      995
      996
      997
      998
      999
      1000
      1001
      1002
      1003
      1004
      1005
      1006
      1007
      1008
      1009
      1010
      1011
      1012
      1013
      1014
      1015
      1016
      1017
      1018
      1019
      1020
      1021
      1022
      1023
      1024
      1025
      1026
      1027
      1028
      1029
      1030
      1031
      1032
      1033
      1034
      1035
      1036
      1037
      1038
      1039
      1040
      1041
      1042
      1043
      1044
      1045
      1046
      1047
      1048
      1049
      1050
      1051
      1052
      1053
      1054
      1055
      1056
      1057
      1058
      1059
      1060
      1061
      1062
      1063
      1064
      1065
      1066
      1067
      1068
      1069
      1070
      1071
      1072
      1073
      1074
      1075
      1076
      1077
      1078
      1079
      1080
      1081
      1082
      1083
      1084
      1085
      1086
      1087
      1088
      1089
      1090
      1091
      1092
      1093
      1094
      1095
      1096
      1097
      1098
      1099
      1100
      1101
      1102
      1103
      1104
      1105
      1106
      1107
      1108
      1109
      1110
      1111
      1112
      1113
      1114
      1115
      1116
      1117
      1118
      1119
      1120
      1121
      1122
      1123
      1124
      1125
      1126
      1127
      1128
      1129
      1130
      1131
      1132
      1133
      1134
      1135
      1136
      1137
      1138
      1139
      1140
      1141
      1142
      1143
      1144
      1145
      1146
      1147
      1148
      1149
      1150
      1151
      1152
      1153
      1154
      1155
      1156
      1157
      1158
      1159
      1160
      1161
      1162
      1163
      1164
      1165
      1166
      1167
      1168
      1169
      1170
      1171
      1172
      1173
      1174
      1175
      1176
      1177
      1178
      1179
      1180
      1181
      1182
      1183
      1184
      1185
      1186
      1187
      1188
      1189
      1190
      1191
      1192
      1193
      1194
      1195
      1196
      1197
      1198
      1199
      1200
      1201
      1202
      1203
      1204
      1205
      1206
      1207
      1208
      1209
      1210
      1211
      1212
      1213
      1214
      1215
      1216
      1217
      1218
      1219
      1220
      1221
      1222
      1223
      1224
      1225
      1226
      1227
      1228
      1229
      1230
      1231
      1232
      1233
      1234
      1235
      1236
      1237
      1238
      1239
      1240
      1241
      1242
      1243
      1244
      1245
      1246
      1247
      1248
      1249
      1250
      1251
      1252
      1253
      1254
      1255
      1256
      1257
      1258
      1259
      1260
      1261
      1262
      1263
      1264
      1265
      1266
      1267
      1268
      1269
      1270
      1271
      1272
      1273
      1274
      1275
      1276
      1277
      1278
      1279
      1280
      1281
      1282
      1283
      1284
      1285
      1286
      1287
      1288
      1289
      1290
      1291
      1292
      1293
      1294
      1295
      1296
      1297
      1298
      1299
      1300
      1301
      1302
      1303
      1304
      1305
      1306
      1307
      1308
      1309
      1310
      1311
      1312
      1313
      1314
      1315
      1316
      1317
      1318
      1319
      1320
      1321
      1322
      1323
      1324
      1325
      1326
      1327
      1328
      1329
      1330
      1331
      1332
      1333
      1334
      1335
      1336
      1337
      1338
      1339
      1340
      1341
      1342
      1343
      1344
      1345
      1346
      1347
      1348
      1349
      1350
      1351
      1352
      1353
      1354
      1355
      1356
      1357
      1358
      1359
      1360
      1361
      1362
      1363
      1364
      1365
      1366
      1367
      1368
      1369
      1370
      1371
      1372
      1373
      1374
      1375
      1376
      1377
      1378
      1379
      1380
      1381
      1382
      1383
      1384
      1385
      1386
      1387
      1388
      1389
      1390
      1391
      1392
      1393
      1394
      1395
      1396
      1397
      1398
      1399
      1400
      1401
      1402
      1403
      1404
      1405
      1406
      1407
      1408
      1409
      1410
      1411
      1412
      1413
      1414
      1415
      1416
      1417
      1418
      1419
      1420
      1421
      1422
      1423
      1424
      1425
      1426
      1427
      1428
      1429
      1430
      1431
      1432
      1433
      1434
      1435
      1436
      1437
      1438
      1439
      1440
      1441
      1442
      1443
      1444
      1445
      1446
      1447
      1448
      1449
      1450
      1451
      1452
      1453
      1454
      1455
      1456
      1457
      1458
      1459
      1460
      1461
      1462
      1463
      1464
      1465
      1466
      1467
      1468
      1469
      1470
      1471
      1472
      1473
      1474
      1475
      1476
      1477
      1478
      1479
      1480
      1481
      1482
      1483
      1484
      1485
      1486
      1487
      1488
      1489
      1490
      1491
      1492
      1493
      1494
      1495
      1496
      1497
      1498
      1499
      1500
      1501
      1502
      1503
      1504
      1505
      1506
      1507
      1508
      1509
      1510
      1511
      1512
      1513
      1514
      1515
      1516
      1517
      1518
      1519
      1520
      1521
      1522
      1523
      1524
      1525
      1526
      1527
      1528
      1529
      1530
      1531
      1532
      1533
      1534
      1535
      1536
      1537
      1538
      1539
      1540
      1541
      1542
      1543
      1544
      1545
      1546
      1547
      1548
      1549
      1550
      1551
      1552
      1553
      1554
      1555
      1556
      1557
      1558
      1559
      1560
      1561
      1562
      1563
      1564
      1565
      1566
      1567
      1568
      1569
      1570
      1571
      1572
      1573
      1574
      1575
      1576
      1577
      1578
      1579
      1580
      1581
      1582
      1583
      1584
      1585
      1586
      1587
      1588
      1589
      1590
      1591
      1592
      1593
      1594
      1595
      1596
      1597
      1598
      1599
      1600
      1601
      1602
      1603
      1604
      1605
      1606
      1607
      1608
      1609
      1610
      1611
      1612
      1613
      1614
      1615
      1616
      1617
      1618
      1619
      1620
      1621
      1622
      1623
      1624
      1625
      1626
      1627
      1628
      1629
      1630
      1631
      1632
      1633
      1634
      1635
      1636
      1637
      1638
      1639
      1640
      1641
      1642
      1643
      1644
      1645
      1646
      1647
      1648
```

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

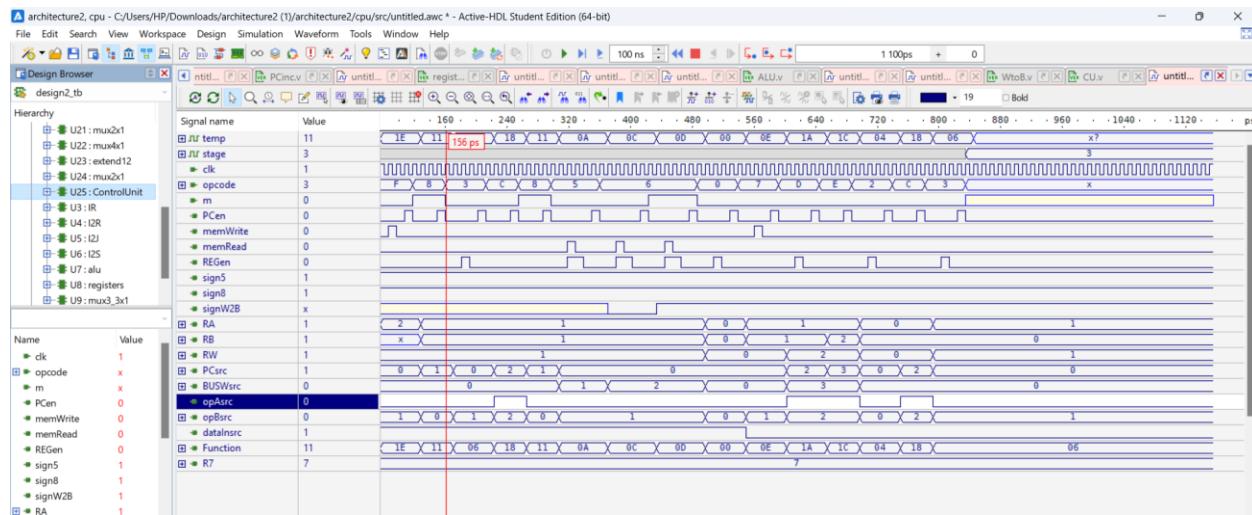


Figure 16: Control Unit Testbench Waveform

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};
```

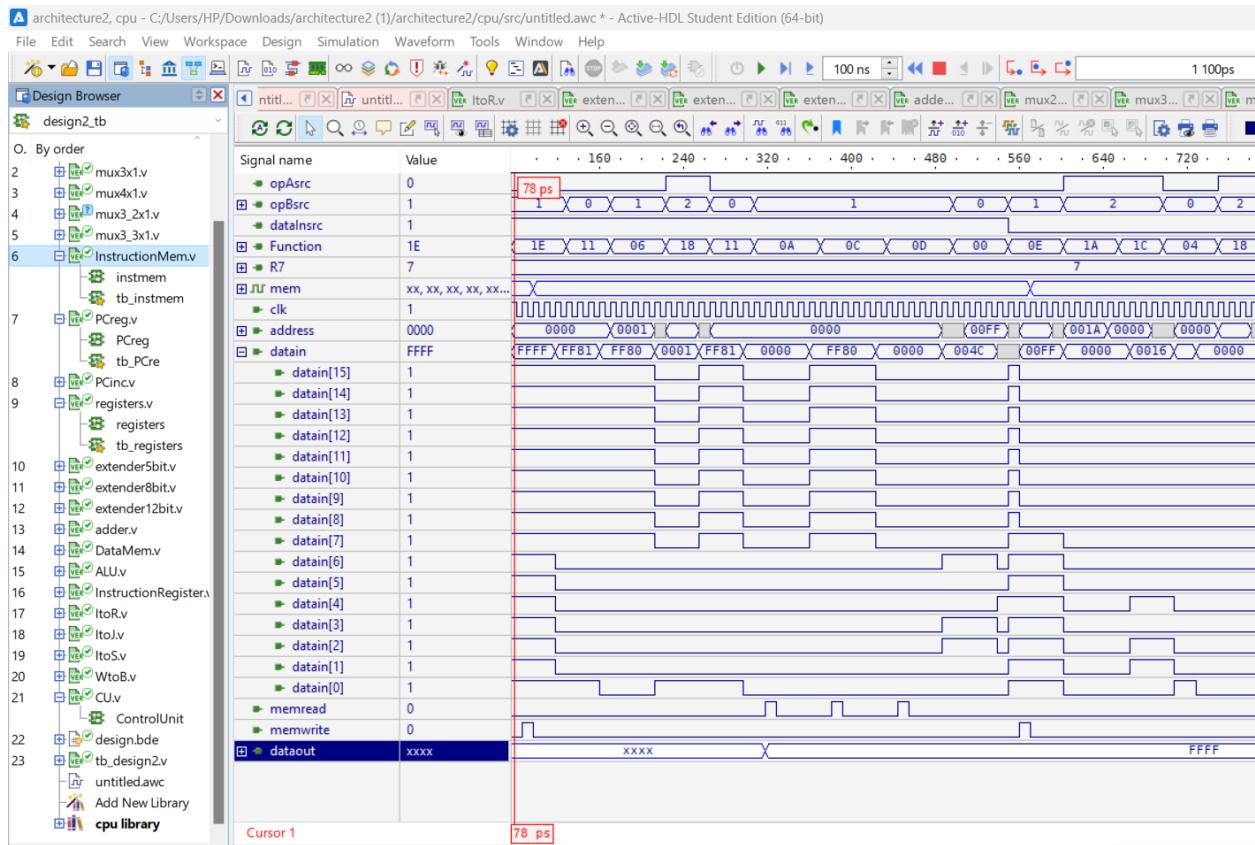


Figure 17: SV test

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};

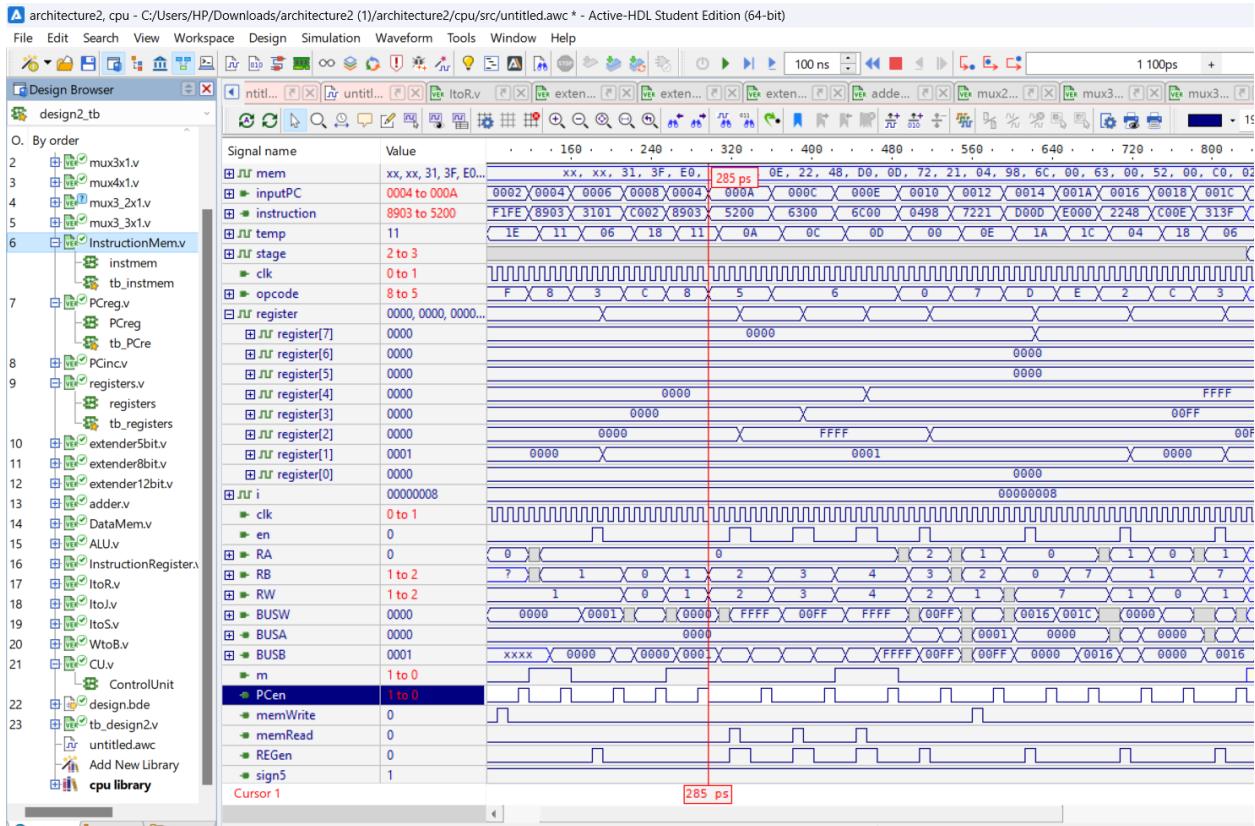


Figure 18: Branch Test

ADDI:

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

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

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

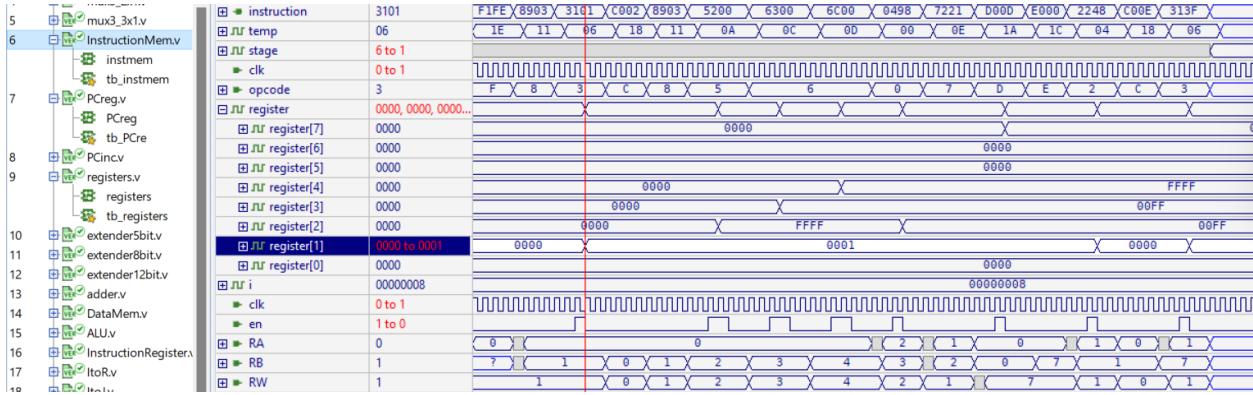


Figure 29:ADDI Test

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};

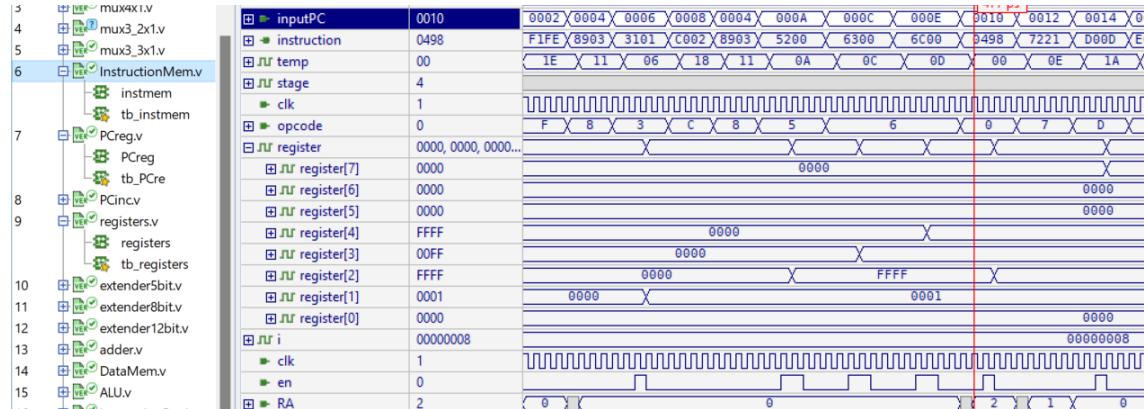


Figure 20: Jump test

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};

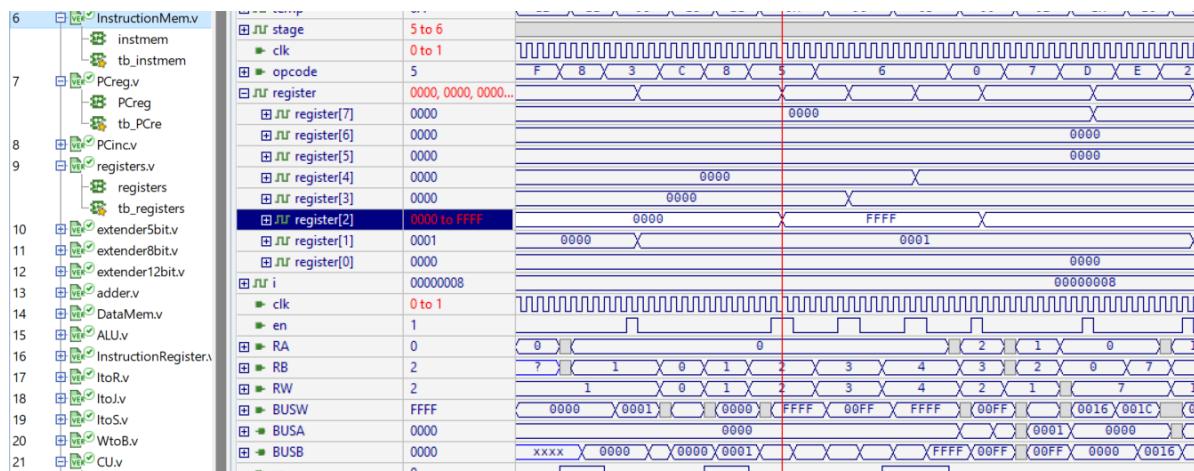


Figure 21: LoadW test

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};
```

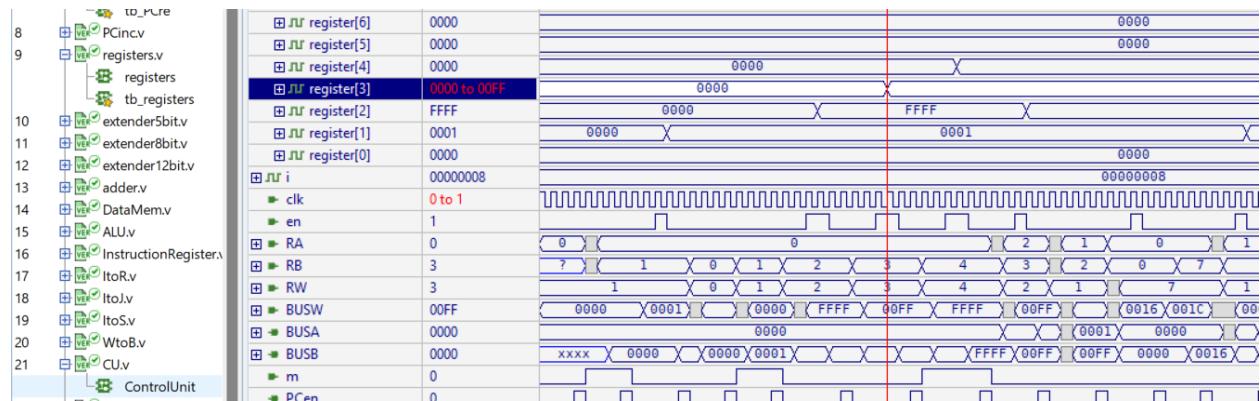


Figure 22: Lbu test

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};
```

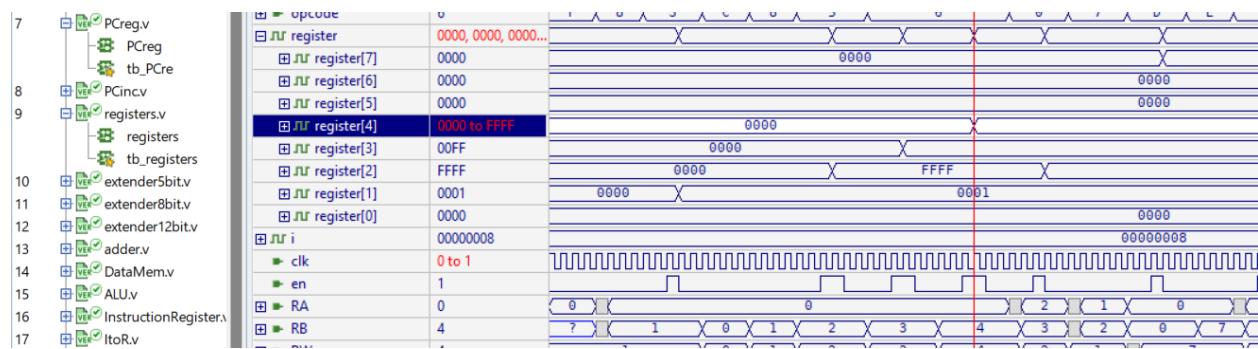


Figure 23: LBs test

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};

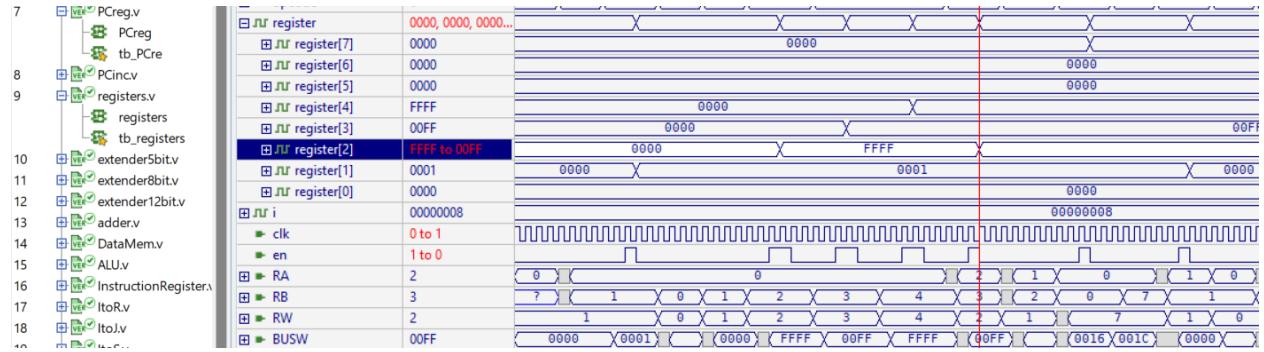


Figure 24: AND test

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};
```

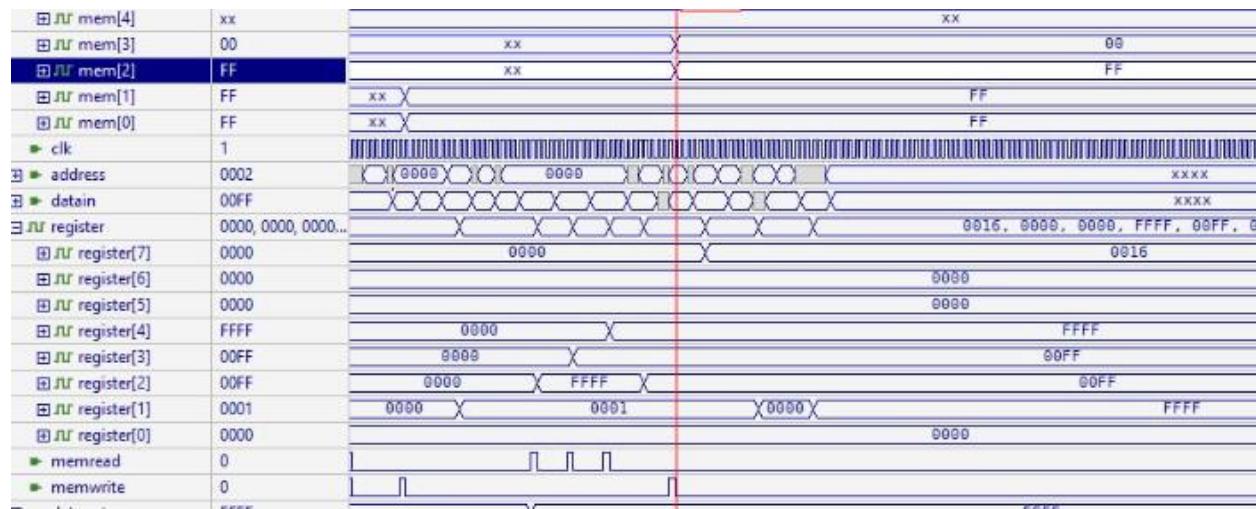


Figure 25: SW test

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};
```

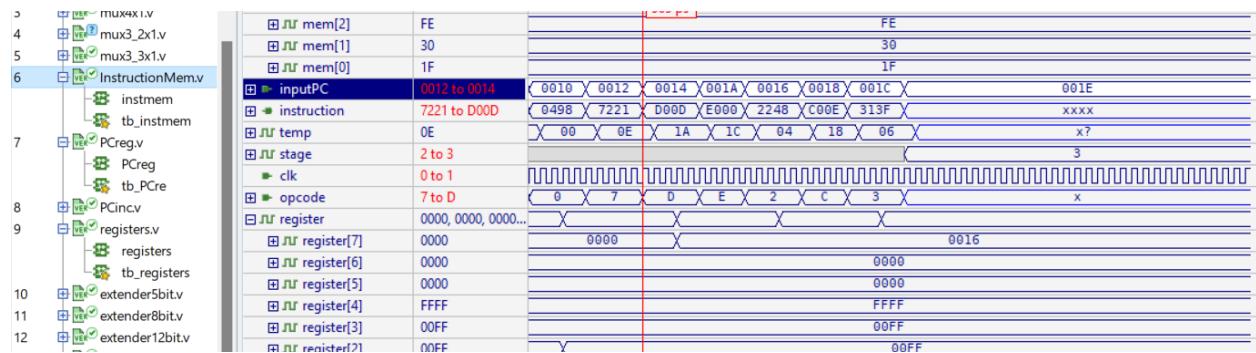


Figure 26: CALL test

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};

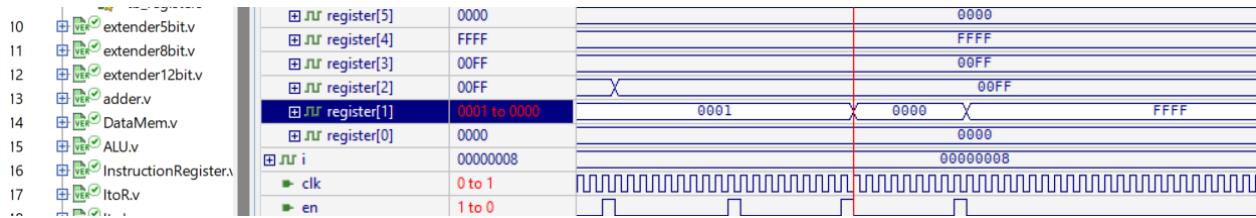


Figure 27: SUB test

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};
```

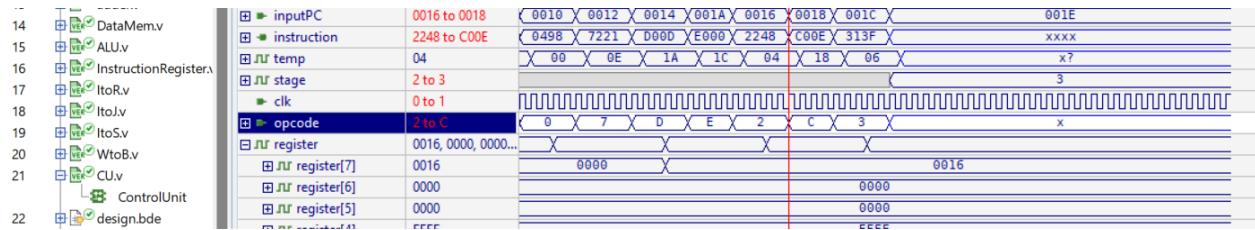


Figure 28: JUMP test

Instruction fetch for jump instruction

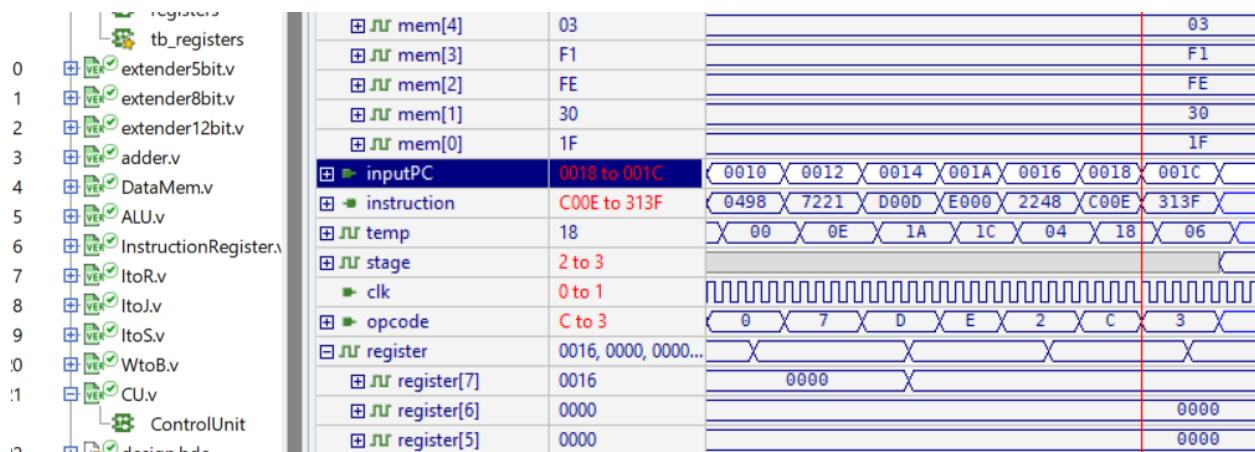


Figure 29: JUMP test

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};
```

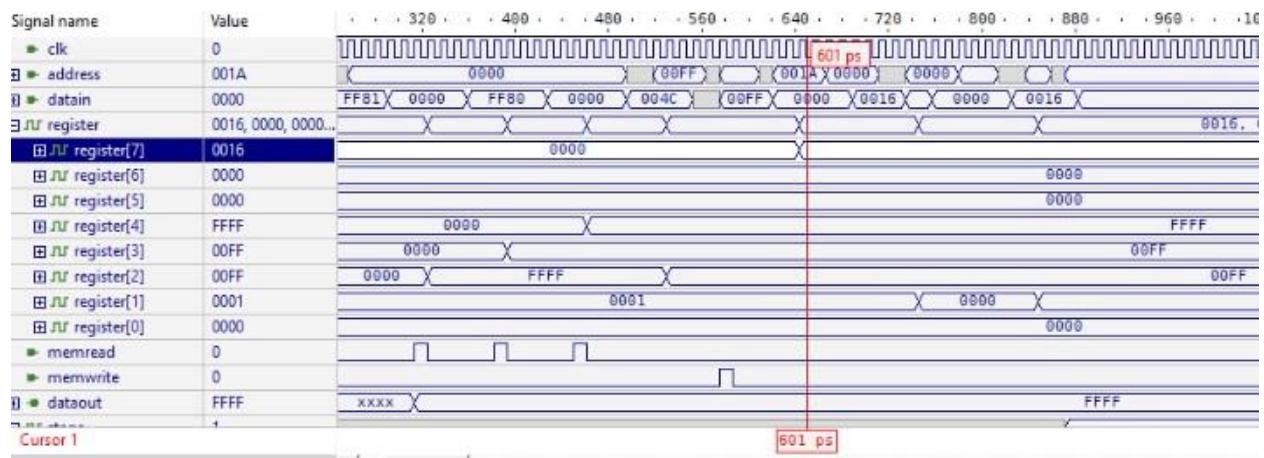


Figure 30: RET test

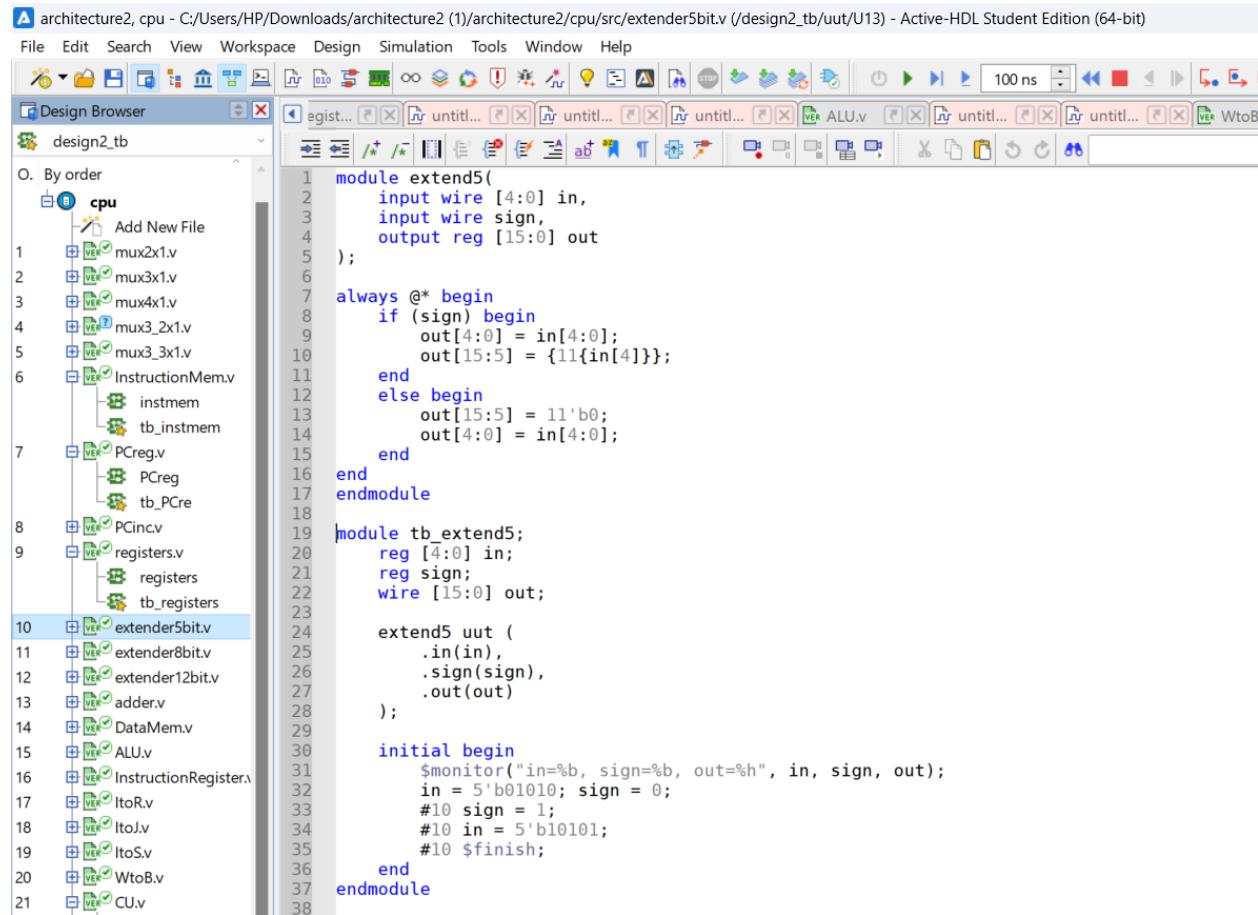
After the Return instruction, the Program Counter PC equals to R7

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



The screenshot shows the Active-HDL Student Edition interface. The top menu bar includes File, Edit, Search, View, Workspace, Design, Simulation, Tools, Window, and Help. The toolbar below has various icons for file operations like Open, Save, and Run. The main window is divided into two panes: the Design Browser on the left and the code editor on the right. The Design Browser lists files in the project tree under 'design2_tb'. The code editor displays the Verilog source code for the 'extender5bit.v' module. The code defines a module 'extend5' with an input wire [4:0] 'in', an input wire 'sign', and an output reg [15:0] 'out'. It contains an always block with an initial begin section for monitoring and stimulus generation.

```
architecture2, cpu - C:/Users/HP/Downloads/architecture2 (1)/architecture2/cpu/src/extender5bit.v (/design2_tb/uut/U13) - Active-HDL Student Edition (64-bit)

File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
design2_tb
O. By order
cpu
  mux2x1.v
  mux3x1.v
  mux4x1.v
  mux3_2x1.v
  mux3_3x1.v
  InstructionMem.v
    instmem
    tb_instmem
  PCReg.v
    PCReg
    tb_PCre
  PCincv
  registers.v
    registers
    tb_registers
extender5bit.v
extender8bit.v
extender12bit.v
adder.v
DataMem.v
ALU.v
InstructionRegister.x
ItoR.v
ItoJ.v
ItoS.v
WtoB.v
CU.v

1 module extend5(
2     input wire [4:0] in,
3     input wire sign,
4     output reg [15:0] out
5 );
6
7 always @* begin
8     if (sign) begin
9         out[4:0] = in[4:0];
10        out[15:5] = {11{in[4]}};
11    end
12    else begin
13        out[15:5] = 11'b0;
14        out[4:0] = in[4:0];
15    end
16 end
17 endmodule
18
19 module tb_extend5;
20     reg [4:0] in;
21     reg sign;
22     wire [15:0] out;
23
24     extend5 uut (
25         .in(in),
26         .sign(sign),
27         .out(out)
28     );
29
30     initial begin
31         $monitor("in=%b, sign=%b, out=%h", in, sign, out);
32         in = 5'b01010; sign = 0;
33         #10 sign = 1;
34         #10 in = 5'b10101;
35         #10 $finish;
36     end
37 endmodule
38
```

Figure 31: extenders

2) 8 bit extender

The screenshot shows the Active-HDL Student Edition interface. The Design Browser on the left lists various files under the project 'design2_tb'. The code editor on the right contains the Verilog code for the 'extend8' module.

```
1 module extend8(
2     input wire [7:0] in,
3     input wire sign,
4     output reg [15:0] out
5 );
6
7 always @* begin
8     if (sign) begin
9         out[7:0] = in[7:0];
10        out[15:8] = {8{in[7]}};
11    end
12    else begin
13        out[15:8] = 8'b0;
14        out[7:0] = in[7:0];
15    end
16 endmodule
17
18 module tb_extend8;
19     reg [7:0] in;
20     reg sign;
21     wire [15:0] out;
22
23 extend8 uut (
24     .in(in),
25     .sign(sign),
26     .out(out)
27 );
28
29 initial begin
30     $monitor("in=%b, sign=%b, out=%h", in, sign, out);
31     in = 8'b01010101; sign = 0;
32     #10 sign = 1;
33     #10 in = 8'b10101010;
34     #10 $finish;
35 end
36
37 endmodule
38
```

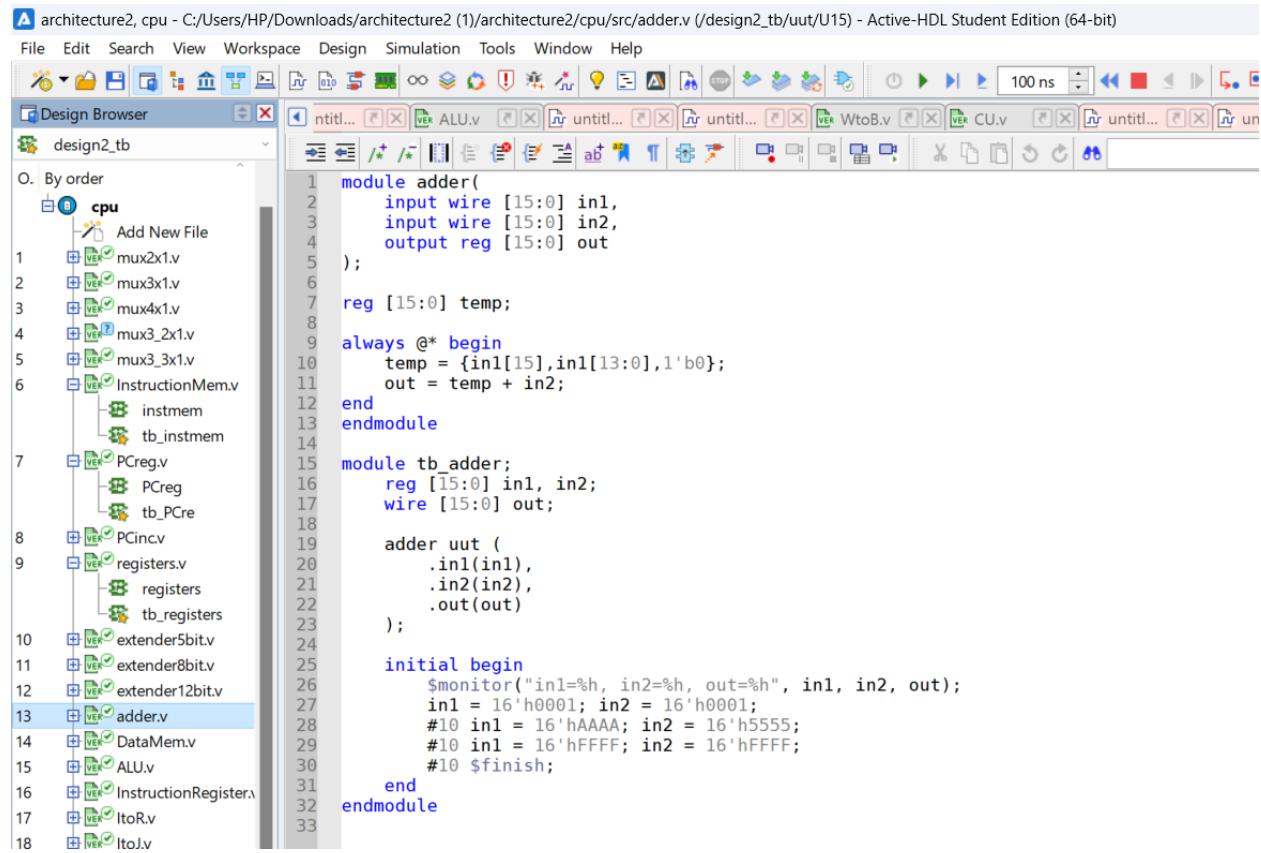
3) 12bit extender:

The screenshot shows the Active-HDL Student Edition interface. The Design Browser on the left lists various files under the project 'design2_tb'. The code editor on the right contains the Verilog code for the 'extend12' module.

```
1 module extend12(
2     input wire [11:0] in,
3     output reg [15:0] out
4 );
5
6 always @* begin
7     out[15:12] = 4'b0;
8     out[11:0] = in[11:0];
9 end
10
11 endmodule
```

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.



The screenshot shows the Active-HDL Student Edition interface. The left pane displays the 'Design Browser' with a tree view of the project structure. The 'adder.v' file is selected in the browser. The right pane shows the Verilog code for the 'adder' module and its testbench.

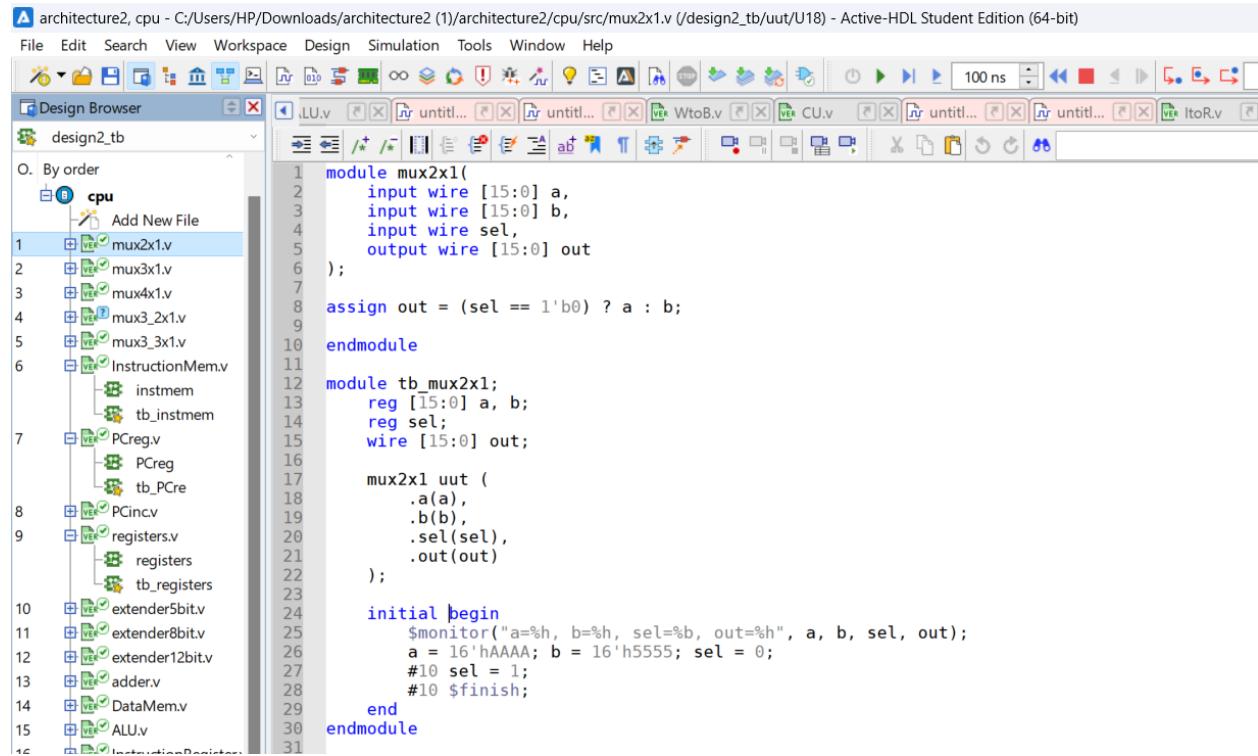
```
1 module adder(
2     input wire [15:0] in1,
3     input wire [15:0] in2,
4     output reg [15:0] out
5 );
6
7     reg [15:0] temp;
8
9     always @* begin
10         temp = {in1[15],in1[13:0],1'b0};
11         out = temp + in2;
12     end
13 endmodule
14
15 module tb_adder;
16     reg [15:0] in1, in2;
17     wire [15:0] out;
18
19     adder uut (
20         .in1(in1),
21         .in2(in2),
22         .out(out)
23     );
24
25     initial begin
26         $monitor("in1=%h, in2=%h, out=%h", in1, in2, out);
27         in1 = 16'h0001; in2 = 16'h0001;
28         #10 in1 = 16'hAAAA; in2 = 16'h5555;
29         #10 in1 = 16'hFFFF; in2 = 16'hFFFF;
30         #10 $finish;
31     end
32 endmodule
33
```

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



The screenshot shows the Active-HDL Student Edition interface. The top menu bar includes File, Edit, Search, View, Workspace, Design, Simulation, Tools, Window, and Help. The main window has two panes: the left pane is the Design Browser showing a hierarchical list of files under 'design2_tb' (cpu, mux2x1.v, mux3x1.v, mux4x1.v, mux3_2x1.v, mux3_3x1.v, InstructionMem.v, PCreg.v, PCincv.v, PCincv, registers.v, extender5bit.v, extender8bit.v, extender12bit.v, adder.v, DataMem.v, ALU.v), and the right pane is a code editor displaying Verilog code for a MUX 2x1 module and its testbench.

```
architecture2, cpu - C:/Users/HP/Downloads/architecture2 (1)/architecture2/cpu/src/mux2x1.v (/design2_tb/uut/U18) - Active-HDL Student Edition (64-bit)

File Edit Search View Workspace Design Simulation Tools Window Help

Design Browser
design2_tb
O. By order
cpu
  Add New File
  mux2x1.v
  mux3x1.v
  mux4x1.v
  mux3_2x1.v
  mux3_3x1.v
  InstructionMem.v
    instmem
    tb_instmem
  PCreg.v
    PCreg
    tb_PCre
  PCincv.v
  PCincv
  registers.v
    registers
    tb_registers
  extender5bit.v
  extender8bit.v
  extender12bit.v
  adder.v
  DataMem.v
  ALU.v
  InstructionRegisters.v

1 module mux2x1(
2   input wire [15:0] a,
3   input wire [15:0] b,
4   input wire sel,
5   output wire [15:0] out
6 );
7
8 assign out = (sel == 1'b0) ? a : b;
9
10 endmodule
11
12 module tb_mux2x1;
13   reg [15:0] a, b;
14   reg sel;
15   wire [15:0] out;
16
17   mux2x1 uut (
18     .a(a),
19     .b(b),
20     .sel(sel),
21     .out(out)
22   );
23
24   initial begin
25     $monitor("a=%h, b=%h, sel=%b, out=%h", a, b, sel, out);
26     a = 16'hAAAA; b = 16'h5555; sel = 0;
27     #10 sel = 1;
28     #10 $finish;
29   end
30 endmodule
31
```

Figure 35: MUX

Mux 3x1

architecture2, cpu - C:/Users/HP/Downloads/architecture2 (1)/architecture2/cpu/src/mux3x1.v (/design2_tb/uut/U20) - Active-HDL Student Edition (64-bit)

```

File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
design2_tb
O. By order
cpu
  mux2x1.v
  mux3x1.v
  mux4x1.v
  mux3_2x1.v
  mux3_3x1.v
  InstructionMem.v
    instmem
    tb_instmem
  PCreg.v
    PCreg
    tb_PCre
  PCincv
  registers.v
    registers
    tb_registers
  extender5bit.v
  extender8bit.v
  extender12bit.v
  adder.v
  DataMem.v
  ALU.v
  InstructionRegisters.v
  ItoR.v
  ItoJ.v
  ItoS.v
  WtoB.v
  CU.v
  ControlUnit

1 module mux3x1(
2   input wire [15:0] a,
3   input wire [15:0] b,
4   input wire [15:0] c,
5   input wire [1:0] sel,
6   output wire [15:0] out
7 );
8
9 assign out = (sel == 2'b00) ? a :
10   (sel == 2'b01) ? b :
11   c;
12
13 endmodule
14
15 module tb_mux3x1;
16   reg [15:0] a, b, c;
17   reg [1:0] sel;
18   wire [15:0] out;
19
20   mux3x1 uut (
21     .a(a),
22     .b(b),
23     .c(c),
24     .sel(sel),
25     .out(out)
26   );
27
28 initial begin
29   $monitor("a=%h, b=%h, c=%h, sel=%b, out=%h", a, b, c, sel, out);
30   a = 16'hAAAA; b = 16'h5555; c = 16'hFFFF;
31   sel = 2'b00; #10;
32   sel = 2'b01; #10;
33   sel = 2'b10; #10;
34   $finish;
35 end
36
37 endmodule

```

MUX 4x1

architecture2, cpu - C:/Users/HP/Downloads/architecture2 (1)/architecture2/cpu/src/mux4x1.v (/design2_tb/uut/U14) - Active-HDL Student Edition (64-bit)

```

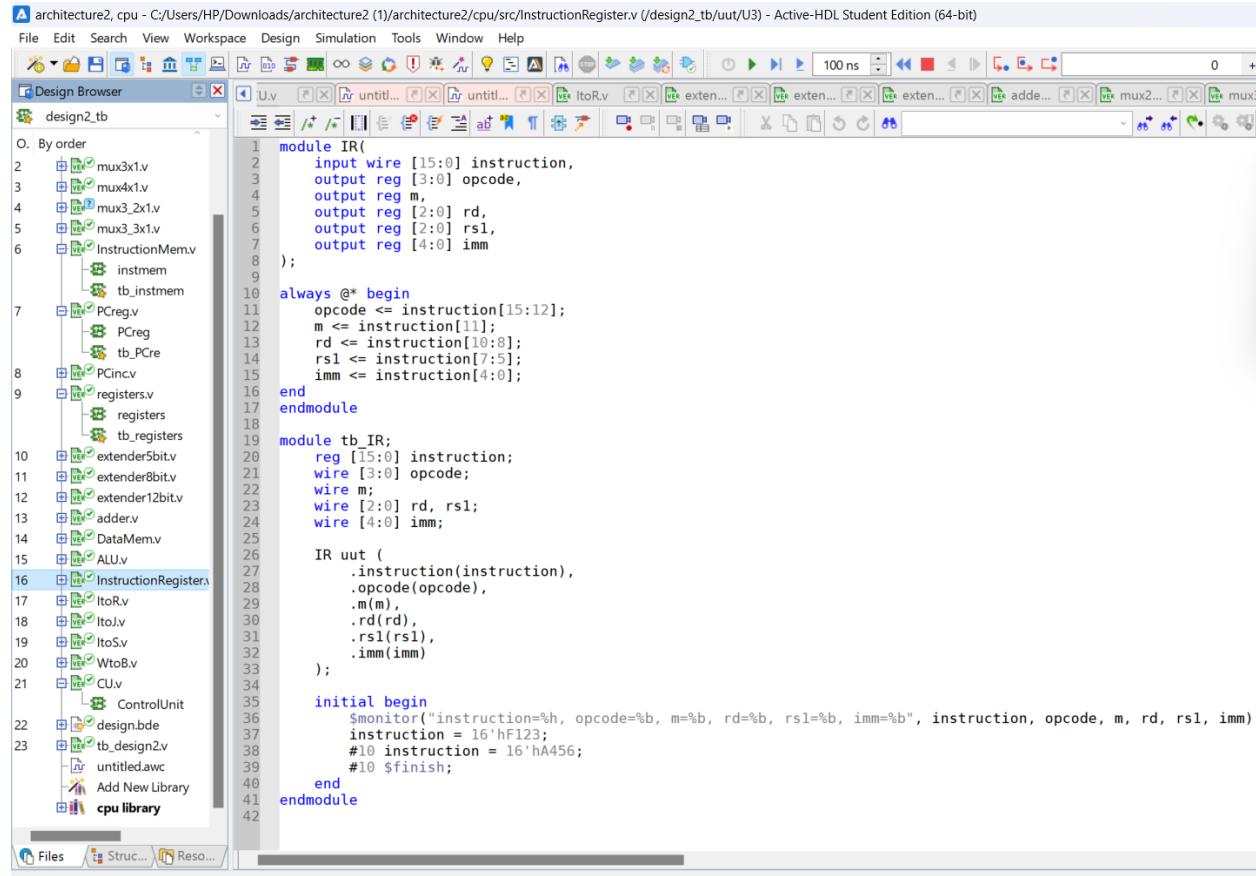
File Edit Search View Workspace Design Simulation Tools Window Help
Design Browser
design2_tb
O. By order
cpu
  mux2x1.v
  mux3x1.v
  mux4x1.v
  mux3_2x1.v
  mux3_3x1.v
  InstructionMem.v
    instmem
    tb_instmem
  PCreg.v
    PCreg
    tb_PCre
  PCincv
  registers.v
    registers
    tb_registers
  extender5bit.v
  extender8bit.v
  extender12bit.v
  adder.v
  DataMem.v
  ALU.v
  InstructionRegisters.v
  ItoR.v
  ItoJ.v
  ItoS.v
  WtoB.v
  CU.v
  ControlUnit

1 module mux4x1(
2   input wire [15:0] a,
3   input wire [15:0] b,
4   input wire [15:0] c,
5   input wire [15:0] d,
6   input wire [1:0] sel,
7   output wire [15:0] out
8 );
9
10 assign out = (sel == 2'b00) ? a :
11   (sel == 2'b01) ? b :
12   (sel == 2'b10) ? c :
13   d;
14
15 endmodule
16
17 module tb_mux4x1;
18   reg [15:0] a, b, c, d;
19   reg [1:0] sel;
20   wire [15:0] out;
21
22   mux4x1 uut (
23     .a(a),
24     .b(b),
25     .c(c),
26     .d(d),
27     .sel(sel),
28     .out(out)
29   );
30
31 initial begin
32   $monitor("a=%h, b=%h, c=%h, d=%h, sel=%b, out=%h", a, b, c, d, sel, out);
33   a = 16'hAAAA; b = 16'h5555; c = 16'hFFFF; d = 16'h0000;
34   sel = 2'b00; #10;
35   sel = 2'b01; #10;
36   sel = 2'b10; #10;
37   sel = 2'b11; #10;
38   $finish;
39 end
40
41 endmodule

```

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.



The screenshot shows the Active-HDL Student Edition interface. The left pane is the 'Design Browser' showing a hierarchical tree of design components. The right pane is a code editor displaying Verilog HDL code for the 'IR' module and its testbench 'tb_IR'. The code defines the module's inputs (instruction [15:0]), outputs (opcode [3:0], m [1:0], rd [2:0], rs1 [2:0], imm [4:0]), and internal logic. It also includes an 'IR uut' instantiation and a testbench 'initial begin' block for monitoring and stimulus.

```
architecture2, cpu - C:/Users/HP/Downloads/architecture2 (1)/architecture2/cpu/src/InstructionRegister.v (/design2_tb/uut/U3) - Active-HDL Student Edition (64-bit)

File Edit Search View Workspace Design Simulation Tools Window Help

Design Browser
design2_tb
O. By order
1 mux3x1.v
2 mux4x1.v
3 mux3_2x1.v
4 mux3_3x1.v
5 InstructionMem.v
6 instmem
7 tb_instmem
8 PCReg.v
9 PCReg
10 tb_PCre
11 PCincv
12 registers.v
13 tb_registers
14 extender5bit.v
15 extender8bit.v
16 extender12bit.v
17 adder.v
18 DataMem.v
19 ALU.v
20 InstructionRegister.v
21 ItoR.v
22 ItoJ.v
23 ItoS.v
24 WtoB.v
25 CU.v
26 ControlUnit
27 design.bde
28 tb_design2.v
29 untitled.awc
30 Add New Library
31 cpu library

1 module IR(
2     input wire [15:0] instruction,
3     output reg [3:0] opcode,
4     output reg m,
5     output reg [2:0] rd,
6     output reg [2:0] rs1,
7     output reg [4:0] imm
8 );
9
10 always @* begin
11     opcode <= instruction[15:12];
12     m <= instruction[11];
13     rd <= instruction[10:8];
14     rs1 <= instruction[7:5];
15     imm <= instruction[4:0];
16 end
17 endmodule

18 module tb_IR;
19     reg [15:0] instruction;
20     wire [3:0] opcode;
21     wire m;
22     wire [2:0] rd, rs1;
23     wire [4:0] imm;
24
25 IR uut (
26     .instruction(instruction),
27     .opcode(opcode),
28     .m(m),
29     .rd(rd),
30     .rs1(rs1),
31     .imm(imm)
32 );
33
34 initial begin
35     $monitor("instruction=%h, opcode=%b, m=%b, rd=%b, rs1=%b, imm=%b", instruction, opcode, m, rd, rs1, imm);
36     instruction = 16'hF123;
37     #10 instruction = 16'hA456;
38     #10 $finish;
39 end
40
41 endmodule
```

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

```
1  module datamem(
2    input wire clk,
3    input wire [15:0] address,
4    input wire [15:0] datain,
5    input wire memread,
6    input wire memwrite,
7    input wire wrByte,
8    output reg [15:0] dataout
9  );
10
11 reg [7:0] mem[31:0];
12
13 initial begin
14   for (int i = 0; i < 32; i = i + 1) begin
15     mem[i] = 8'd0;
16   end
17 end
18
19 always @ (posedge clk) begin
20   if (memwrite) begin
21     if (wrByte) begin
22       mem[address] <= datain[7:0];
23     end
24     else begin
25       mem[address] <= datain[7:0];
26       mem[address + 1] <= datain[15:8];
27     end
28   end
29 end
30
31
32 always @* begin
33   if (memread) begin
34     dataout <= mem[address + 1], mem[address];
35   end
36 end
37 endmodule
38
39 module tb_datamem;
40
41 // inputs
42 reg clk;
43 reg [15:0] address;
44 reg [15:0] datain;
45 reg memread;
46 reg memwrite;
47 reg wrByte;
48
49 // outputs
50 wire [15:0] dataout;
```

Instruction Mem:

```
1 module instmem(
2     input wire [15:0] inputPC,
3     output reg [15:0] instruction
4 );
5
6 reg [7:0] mem[31:0];
7
8 parameter AND = 4'b0000;
9 parameter ADD = 4'b0001;
10 parameter SUB = 4'b0010;
11
12 //I type
13 parameter ADDI = 4'b0011;
14 parameter ANDI = 4'b0100;
15 parameter LW = 4'b0101;
16 parameter LBu = 4'b0110;
17 parameter LBs = 4'b0110;
18 parameter SW = 4'b0111;
19
20 //branch
21 parameter BGT = 4'b1000;
22 parameter BGTZ = 4'b1000;
23 parameter BLT = 4'b1001;
24 parameter BLTZ = 4'b1001;
25 parameter BEQ = 4'b1010;
26 parameter BEQZ = 4'b1010;
27 parameter BNE = 4'b1011;
28 parameter BNEZ = 4'b1011;
29
30 //J type
31 parameter JMP = 4'b1100;
32 parameter CALL = 4'b1101;
33 parameter RET = 4'b1110;
34
35 //S type
36 parameter Sv = 4'b1111;
37
38 initial begin
39     // initialize the memory with the program |
40
41     // mem[0] = ADDI R0, R0, -1; R0 = 0 since it cannot be modified
42     mem[0] = {3'b000, 5'b11111};
43     mem[1] = {ADDI, 1'b0, 3'b000};
44
45     // mem[2] = Sv R0, 16; mem[0] = 16'hFFFF
46     mem[2] = {7'b1111111, 1'b0};
47     mem[3] = {Sv, 3'b000, 1'b1};
48
49     // mem[4] = BGTZ R1, 5; PC = PC + 3 = 5 x 2 = 10
50     mem[4] = {3'b000, 5'b000011};
51     mem[5] = {BGTZ, 1'b1, 3'b001};
52
```

```

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

```
1 module IR(
2     input wire [15:0] instruction,
3     output reg [3:0] opcode,
4     output reg m,
5     output reg [2:0] rd,
6     output reg [2:0] rsl,
7     output reg [4:0] imm
8 );
9
10 always @* begin
11     opcode <= instruction[15:12];
12     m <= instruction[11];
13     rd <= instruction[10:8];
14     rsl <= instruction[7:5];
15     imm <= instruction[4:0];
16 end
17 endmodule
18
19 module tb_IR;
20     reg [15:0] instruction;
21     wire [3:0] opcode;
22     wire m;
23     wire [2:0] rd, rsl;
24     wire [4:0] imm;
25
26     IR uut (
27         .instruction(instruction),
28         .opcode(opcode),
29         .m(m),
30         .rd(rd),
31         .rsl(rsl),
32         .imm(imm)
33     );
34
35 initial begin
36     $monitor("instruction=%h, opcode=%b, m=%b, rd=%b, rsl=%b, imm=%b", instruction, opcode, m, rd, rsl, imm);
37     instruction = 16'hF123;
38     #10 instruction = 16'hA456;
39     #10 $finish;
40 end
41 endmodule
42
```

Control unit

```
1 module ControlUnit(
2     input wire clk,
3     input wire [3:0] opcode,
4     input wire m,
5     //input wire takeBranch,
6     output reg PCen,
7     output reg memWrite,
8     output reg memRead,
9     output reg REN,
10    output reg sign5,
11    output reg sign8,
12    output reg signW2B,
13    output reg [1:0] RA,
14    output reg [1:0] RB,
15    output reg [1:0] RW,
16    output reg [1:0] PLsrc,
17    output reg [1:0] BUSWsrc,
18    output reg opAsrc,
19    output reg [1:0] opBsrc,
20    output reg dataInsrc,
21    output reg [1:0] Function,
22    output wire [2:0] R7
23 );
24
25 assign R7 = 3'b111;
26
27 reg [4:0] temp;
28
29 //R type
30 parameter AND = 5'b00000;
31 parameter ADD = 5'b00010;
32 parameter SUB = 5'b00100;
33
34 //I type
35 parameter ADDI = 5'b00110;
36 parameter ANDI = 5'b01000;
37 parameter LW = 5'b01010;
38 parameter LBu = 5'b01100;
39 parameter LBs = 5'b01101;
40 parameter SW = 5'b01110;
41
42 //branch
43 parameter BGT = 5'b10000;
44 parameter BG TZ = 5'b10001;
45 parameter BLT = 5'b10010;
46 parameter BLTZ = 5'b10011;
47 parameter BEQ = 5'b10100;
48 parameter BEQZ = 5'b10101;
49 parameter BNE = 5'b10110;
50 parameter BNZ = 5'b10111;
```

```

51 //J type
52 parameter JMP = 5'b11000;
53 parameter CALL = 5'b11010;
54 parameter RET = 5'b11100;
55
56 //S type
57 parameter Sv = 5'b11110;
58
59 //stages
60 parameter start = 0;
61 parameter RS = 1;
62 parameter IF = 2;
63 parameter ID = 3;
64 parameter EX = 4;
65 parameter MEM = 5;
66 parameter WB = 6;
67 reg [2:0] stage = start;
68
69 always @(posedge clk) begin
70     case (stage)
71         start: begin
72             PCen <= 0;
73             memWrite <= 0;
74             memRead <= 0;
75             REGen <= 0;
76             stage <= IF;
77         end
78
79         RS: begin
80             PCen <= 1;
81             memWrite <= 0;
82             memRead <= 0;
83             REGen <= 0;
84
85             //if (takeBranch == 1): PCsrc <= 2'b01;
86             //else: PCsrc <= 2'b00;
87
88             stage <= IF;
89         end
90
91         IF: begin
92             PCen <= 0;
93             memWrite <= 0;
94             memRead <= 0;
95             REGen <= 0;
96             stage <= ID;
97         end
98
99
100        ID: begin
101            PCen <= 0;
102            PCsrc <= 2'b00;
103            memWrite <= 0;
104            memRead <= 0;
105            REGen <= 0;
106
107            temp = {opcode,1'b0};
108
109            if (opcode[3:0] == 4'b0110 || (opcode[3:0] > 4'b0111 && opcode[3:0] < 4'b1100)) begin
110                temp[0] = n;
111            end
112
113            case(temp)
114                AND: begin
115                    REGen <= 0;
116                    RA <= 2'b00;
117                    RB <= 2'b00;
118                    RW <= 2'b00;
119                    BUSWsrc <= 2'b00;
120                    opAsrc <= 0;
121                    opBsrc <= 2'b00;
122                    Function <= temp;
123                    stage <= EX;
124                end
125
126                ADD: begin
127                    REGen <= 0;
128                    RA <= 2'b00;
129                    RB <= 2'b00;
130                    RW <= 2'b00;
131                    BUSWsrc <= 2'b00;
132                    opAsrc <= 0;
133                    opBsrc <= 0;
134                    Function <= temp;
135                    stage <= EX;
136                end
137
138                SUB: begin
139                    REGen <= 0;
140                    RA <= 2'b00;
141                    RB <= 2'b00;
142                    RW <= 2'b00;
143                    BUSWsrc <= 2'b00;
144                    opAsrc <= 0;
145                    opBsrc <= 2'b00;
146                    Function <= temp;
147                    stage <= EX;
148                end

```

```

149
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   signW28 <= 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;
199 ...

```

```

200
201 LBs: begin
202   memRead <= 0;
203   REGen <= 0;
204   sign5 <= 1;
205   signW28 <= 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

```

```

250          BLT: begin
251              sign5 <= 1;
252              RA <= 2'b01;
253              RB <= 2'b01;
254              PCsrc <= 2'b01;
255              opAsrc <= 0;
256              opBsrc <= 2'b00;
257              Function <= temp;
258              stage <= EX;
259          end
260
261          BLTZ: begin
262              sign5 <= 1;
263              RA <= 2'b01;
264              RB <= 2'b01;
265              PCsrc <= 2'b01;
266              opAsrc <= 0;
267              opBsrc <= 2'b00;
268              Function <= temp;
269              stage <= EX;
270          end
271
272          BEQ: begin
273              sign5 <= 1;
274              RA <= 2'b01;
275              RB <= 2'b01;
276              PCsrc <= 2'b01;
277              opAsrc <= 0;
278              opBsrc <= 2'b00;
279              Function <= temp;
280              stage <= EX;
281          end
282
283          BEQZ: begin
284              sign5 <= 1;
285              RA <= 2'b01;
286              RB <= 2'b01;
287              PCsrc <= 2'b01;
288              opAsrc <= 0;
289              opBsrc <= 2'b00;
290              Function <= temp;
291              stage <= EX;
292          end
293
294          BNE: begin
295              sign5 <= 1;
296              RA <= 2'b01;
297              RB <= 2'b01;
298              PCsrc <= 2'b01;
299              opAsrc <= 0;
300              opBsrc <= 2'b00;
301
302                  Function <= temp;
303                  stage <= EX;
304          end
305
306          BNEZ: begin
307              sign5 <= 1;
308              RA <= 2'b01;
309              RB <= 2'b01;
310              PCsrc <= 2'b01;
311              opAsrc <= 0;
312              opBsrc <= 2'b00;
313              Function <= temp;
314              stage <= EX;
315
316          JMP: begin
317              PCsrc <= 2'b10;
318              opAsrc <= 1;
319              opBsrc <= 2'b10;
320              Function <= temp;
321              stage <= EX;
322
323          CALL: begin
324              REGen <= 0;
325              RW <= 2'b10;
326              PCsrc <= 2'b10;
327              BUSWsrc <= 2'b11;
328              opAsrc <= 1;
329              opBsrc <= 2'b10;
330              Function <= temp;
331              stage <= EX;
332
333          RET: begin
334              RB <= 2'b10;
335              PCsrc <= 2'b11;
336              Function <= temp;
337              stage <= EX;
338
339          Sv: begin
340              memWrite <= 0;
341              sign8 <= 1;
342              RA <= 2'b10;
343              opAsrc <= 0;
344              dataInsrc <= 1;
345              Function <= temp;
346              stage <= EX;
347
348          end
349      endcase
350
351  end
352
353
354  EX: begin

```

```

358         end
359     endcas
360
361 end
362
363 EX: begin
364     case (temp)
365         AND: begin
366             REGen <= 1;
367             stage <= WB;
368         end
369
370         ADD: begin
371             REGen <= 1;
372             stage <= WB;
373         end
374
375         SUB: begin
376             REGen <= 1;
377             stage <= WB;
378         end
379
380         ADDI: begin
381             REGen <= 1;
382             stage <= WB;
383         end
384
385         ANDI: begin
386             REGen <= 1;
387             stage <= WB;
388         end
389
390         LW: begin
391             memRead <= 1;
392             REGen <= 1;
393             stage <= MEM;
394         end
395
396         LBu: begin
397             memRead <= 1;
398             REGen <= 1;
399             stage <= MEM;
400         end
401
402         LBs: begin
403             memRead <= 1;
404             REGen <= 1;
405             stage <= MEM;
406         end
407
408         SW: begin
409             memWrite <= 1;
410             stage <= MEM;
411         end

```

```

412
413         J: begin
414             memWrite <= 1;
415             stage <= MEM;
416         end
417
418         BGT: begin
419             stage <= RS;
420         end
421
422         BGTZ: begin
423             stage <= RS;
424         end
425
426         BLT: begin
427             stage <= RS;
428         end
429
430         BLTZ: begin
431             stage <= RS;
432         end
433
434         BEQ: begin
435             stage <= RS;
436         end
437
438         BEQZ: begin
439             stage <= RS;
440         end
441
442         BNE: begin
443             stage <= RS;
444         end
445
446         BNEZ: begin
447             stage <= RS;
448         end
449
450         JMP: begin
451             stage <= RS;
452         end
453
454         CALL: begin
455             REGen <= 1;
456             stage <= WB;
457         end
458
459         RET: begin
460             stage <= RS;
461         end
462
463         Sv: begin
464             memWrite <= 1;

```

I type to J type Reg

```
1 module I2J(
2     input wire m,
3     input wire [2:0] rdOld,
4     input wire [2:0] rs1Old,
5     input wire [4:0] immOld,
6     output reg [11:0] offset
7 );
8
9 always @* begin
10    offset <= {m,rdOld,rs1Old,immOld};
11 end
12 endmodule
13
14 module tb_I2J;
15     reg m;
16     reg [2:0] rdOld, rs1Old;
17     reg [4:0] immOld;
18     wire [11:0] offset;
19
20     I2J uut (
21         .m(m),
22         .rdOld(rdOld),
23         .rs1Old(rs1Old),
24         .immOld(immOld),
25         .offset(offset)
26     );
27
28 initial begin
29     $monitor("m=%b, rdOld=%b, rs1Old=%b, immOld=%b, offset=%b", m, rdOld, rs1Old, immOld, offset);
30     m = 1; rdOld = 3'b101; rs1Old = 3'b010; immOld = 5'b11001;
31     #10 rdOld = 3'b111; rs1Old = 3'b011; immOld = 5'b00110;
32     #10 $finish;
33 end
34 endmodule
35
```

I2R

```
1 module I2R(
2     input wire m,
3     input wire [2:0] rdOld,
4     input wire [2:0] rs1Old,
5     input wire [4:0] immOld,
6     output reg [2:0] rd,
7     output reg [2:0] rs1,
8     output reg [2:0] rs2
9 );
10
11 always @* begin
12     rd <= {m,rdOld[2:1]};
13     rs1 <= {rdOld[0],rs1Old[2:1]};
14     rs2 <= {rs1Old[0],immOld[4:3]};
15 end
16 endmodule
17
18 module tb_I2R;
19     reg m;
20     reg [2:0] rdOld, rs1Old;
21     reg [4:0] immOld;
22     wire [2:0] rd, rs1, rs2;
23
24     I2R uut (
25         .m(m),
26         .rdOld(rdOld),
27         .rs1Old(rs1Old),
28         .immOld(immOld),
29         .rd(rd),
30         .rs1(rs1),
31         .rs2(rs2)
32     );
33
34 initial begin
35     $monitor("m=%b, rdOld=%b, rs1Old=%b, immOld=%b, rd=%b, rs1=%b, rs2=%b", m, rdOld, rs1Old, immOld, rd, rs1, rs2);
36     m = 1; rdOld = 3'b101; rs1Old = 3'b010; immOld = 5'b11001;
37     #10 rdOld = 3'b111; rs1Old = 3'b011; immOld = 5'b00110;
38     #10 $finish;
39 end
40 endmodule
41
```

I2S

```
1 module I2S(
2     input wire m,
3     input wire [2:0] rdOld,
4     input wire [2:0] rs1Old,
5     input wire [4:0] immOld,
6     output reg [2:0] rs1,
7     output reg [7:0] imm
8 );
9
10 always @* begin
11     rs1 <= {m,rdOld[2:1]};
12     imm <= {rdOld[0],rs1Old,immOld[4:1]};
13 end
14 endmodule
15
16 module tb_I2S;
17     reg m;
18     reg [2:0] rdOld, rs1Old;
19     reg [4:0] immOld;
20     wire [2:0] rs1;
21     wire [7:0] imm;
22
23     I2S uut (
24         .m(m),
25         .rdOld(rdOld),
26         .rs1Old(rs1Old),
27         .immOld(immOld),
28         .rs1(rs1),
29         .imm(imm)
30     );
31
32     initial begin
33         $monitor("m=%b, rdOld=%b, rs1Old=%b, immOld=%b, rs1=%b, imm=%b", m, rdOld, rs1Old, immOld, rs1, imm);
34         m = 1; rdOld = 3'b101; rs1Old = 3'b010; immOld = 5'b11001;
35         #10 rdOld = 3'b111; rs1Old = 3'b011; immOld = 5'b00110;
36         #10 $finish;
37     end
38 endmodule
39
```

W2B

```
1 module W2B(
2     input wire [15:0] in,
3     input wire sign,
4     output reg [15:0] out
5 );
6
7     always @* begin
8         if (sign) begin
9             out[7:0] = in[7:0];
10            out[15:8] = {8{in[7]}};
11        end
12        else begin
13            out[15:8] = 8'b0;
14            out[7:0] = in[7:0];
15        end
16    end
17 endmodule
```

PcInc

```
1 module PCinc(
2     input wire [15:0] inputPC,
3     output reg [15:0] outPC
4 );
5
6 always @* begin
7     outPC = inputPC + 2;
8 end
9 endmodule
10
11 module tb_PCinc;
12     reg [15:0] inputPC;
13     wire [15:0] outPC;
14
15     PCinc uut (
16         .inputPC(inputPC),
17         .outPC(outPC)
18     );
19
20     initial begin
21         $monitor("inputPC=%h, outPC=%h", inputPC, outPC);
22         inputPC = 16'h0000;
23         #10 inputPC = 16'h0002;
24         #10 inputPC = 16'h0004;
25         #10 $finish;
26     end
27 endmodule
28
```

PCReg

```
1 module PCreg(
2     input clk,
3     input wire en,
4     input wire [15:0] in,
5     output reg [15:0] out
6 );
7
8     reg [15:0] register;
9
10    initial begin
11        register = 16'b0;
12        out = register;
13    end
14
15    always @(posedge clk) begin
16        if (en) begin
17            register = in;
18        end
19    end
20
21    always @* begin
22        out = register;
23    end
24 endmodule
25
26 module tb_PCre;
27     reg clk, en;
28     reg [15:0] in;
29     wire [15:0] out;
30
31     PCreg uut (
32         .clk(clk),
33         .en(en),
34         .in(in),
35         .out(out)
36     );
37
38     initial begin
39         clk = 0;
40         forever #5 clk = ~clk;
41     end
42
43     initial begin
44         $monitor("clk=%b, en=%b, in=%h, out=%h", clk, en, in, out);
45         en = 1; in = 16'h0001;
46         #10 in = 16'h0002;
47         #10 en = 0; in = 16'h0003;
48         #10 en = 1; in = 16'h0004;
49         #10 $finish;
50     end
51 endmodule
52
```

ALU

```

1  module alu(
2      input wire [15:0] operandA,
3      input wire [15:0] operandB,
4      input wire [4:0] Function,
5      output reg [15:0] res,
6      output reg takeBranch
7 );
8
9   reg [15:0] temp;
10
11 //R type
12 parameter AND = 5'b00000;
13 parameter ADD = 5'b00010;
14 parameter SUB = 5'b00100;
15
16 //I type
17 parameter ADDI = 5'b00110;
18 parameter ANDI = 5'b01000;
19 parameter LW = 5'b01010;
20 parameter LBu = 5'b01100;
21 parameter LBs = 5'b01101;
22 parameter SW = 5'b01110;
23
24 //branch
25 parameter BGT = 5'b10000;
26 parameter BGTZ = 5'b10001;
27 parameter BLT = 5'b10010;
28 parameter BLTZ = 5'b10011;
29 parameter BEQ = 5'b10100;
30 parameter BEQZ = 5'b10101;
31 parameter BNE = 5'b10110;
32 parameter BNEZ = 5'b10111;
33
34 //J type
35 parameter JMP = 5'b11000;
36 parameter CALL = 5'b11010;
37 parameter RET = 5'b11100;
38
39 //S type
40 parameter Sv = 5'b11110;
41
42 always @* begin
43     res = 16'b0;
44     takeBranch = 1'b0;
45
46     case(Function)
47         AND: res = operandA & operandB;
48         ADD: res = operandA + operandB;
49         SUB: res = operandA - operandB;
50
51         BGT: takeBranch = (operandB > operandA) ? 1 : 0;
52         BGTZ: takeBranch = (operandB > 0) ? 1 : 0;
53         BLT: takeBranch = (operandB < operandA) ? 1 : 0;
54         BLTZ: takeBranch = (operandB < 0) ? 1 : 0;
55         BEQ: takeBranch = (operandB == operandA) ? 1 : 0;
56         BEQZ: takeBranch = (operandB == 0) ? 1 : 0;
57         BNE: takeBranch = (operandB == operandA) ? 0 : 1;
58         BNEZ: takeBranch = (operandB == 0) ? 0 : 1;
59
60         JMP: begin
61             temp = operandB << 1;
62             res = {operandA[15:12],temp[11:0]};
63         end
64
65         CALL: begin
66             temp = operandB << 1;
67             res = {operandA[15:12],temp[11:0]};
68         end
69
70         RET: res = operandB;
71
72         Sv: res = operandA;
73     endcase
74 end
75
76 endmodule
77
78 module tb_alu;
79     reg [15:0] operandA, operandB;
80     reg [4:0] Function;
81     wire [15:0] res;
82     wire zeroFlag, negativeFlag;
83
84     alu uut (
85         .operandA(operandA),
86         .operandB(operandB),
87         .Function(Function),
88         .res(res),
89         .takeBranch(takeBranch)
90     );
91
92     initial begin
93         $monitor("operandA=%h, operandB=%h, Function=%b, res=%h, takeBranch=%b", operandA, operandB, Function, res, takeBranch);
94         operandA = 16'h0001; operandB = 16'h0001; Function = 5'b00010; // ADD
95         #10 Function = 5'b00100; // SUB
96         #10 Function = 5'b00000; // AND
97         #10 Function = 5'b10100; // BEQ
98         #10 $finish;
99     end
100 endmodule
101
102
103
104
105
106
107

```

Registers

```
module registers(
    input wire clk,
    input wire en,
    input [2:0] RA,
    input [2:0] RB,
    input [2:0] RW,
    input wire [15:0] BUSM,
    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] = 16'00;
    end
end

always @ (posedge clk) begin
    if (~en & (RW == 3'b000)) begin
        register[RW] := BUSM;
    end
    BUSA <- register[RA];
    BUSB <- register[RB];
end
endmodule

module tb_registers;
    reg clk, en;
    reg [2:0] RA, RB, RW;
    wire [15:0] BUSM, BUSA, BUSB;
    registers uut (
        .clk(clk),
        .en(en),
        .RA(RA),
        .RB(RB),
        .RW(RW),
        .BUSM(BUSM),
        .BUSA(BUSA),
        .BUSB(BUSB)
    );
    initial begin
        clk = 0;
        forever #1 clk = ~clk;
    end
    initial begin
        $monitor(`clk=%b, en=%b, RA=%b, RB=%b, RW=%b, BUSM=%h, BUSA=%h, BUSB=%h, reg[RA]=%h, reg[RB]=%h, reg[RW]=%h', clk, en, RA, RB, RW, BUSM, BUSA, BUSB, uut.register[RA], uut.register[RB], uut.register[RW]);
        #10 RA = 3'b001; RB = 3'b000;
        #10 RA = 3'b001; RB = 3'b000;
        #10 RA = 3'b010; RB = 3'b000;
        #10 RA = 3'b010; RB = 3'b001;
        #10 $finish;
    end
endmodule
```