University of Minho

Master in Industrial Electronic and Computers Engineering

Embedded Systems

Dedicated Processors Project

RISC-V Processor

Authors - Group 5
Bruno Silva A88289
Rafael Pedro A88271
Silviane Correia A90037
Sofia Alves A90263

Professors:
Tiago Gomes
Rui Machado
Sérgio Pereira

May 31, 2022

Contents

List of Figures

Acronyms

RV32I RISC-V 32-bit integer instruction set

ISA Instruction Set Architecture

PC Program Counter

1 Introduction

In the context of the dedicated processor project subject of the embedded systems course, the group was challenged to add features to a given basic single-cycle processor. Some of the possible features that could be implemented in this project are in the following list:

- ALL RV32I instructions (except fence, ecall, ebreak, csrr-based)
- Type (single-cycle, multicycle, pipeline)
- Add a Peripheral at choice
- Add standard/custom buses/interfaces
- Communicate with PS (for code transfers)
- Add custom IP for Processor control (reset)
- Explore different memory options (BRAMs, DDR, logic (LUTs))

The group decided to focus on three features of the list, which are the implementation of all of the RV32I instructions, adding pipeline and adding peripherals. The following chapters of this report describe the implementation of this features.

2 RV32I Instructions

The RISC-V 32-bit integer instruction set (RV32I) forms the core of RISC-V instruction set. In the Instruction Set Architecture (ISA), there are six instruction types:

- R-type (register-type);
- I-type (immediate);
- S/B-type (store/branch);
- U/J-type (upper immediate/jump);

These instructions operate on operands, that can be stored in registers or memory, or they can be constants/immediates stored in the instruction itself.

Since retrieving operands from memory takes a long time and the instructions must run fast, the architecture provides 32 registers to store commonly used operands. These are kept in a multiported memory called a register file. There are 31 general-purpose registers x1–x31, which hold integer values and the register x0 is hardwired to the constant 0.

As mentioned before the RISC-V instructions can use immediate operands, they are called immediates because their values are immediately available from the instruction and do not require a register or memory access.

Data can also be stored in memory to have more than 32 variables (the 32 registers). The register file is small and fast and the memory is larger and slower. In this architecture, instructions operate exclusively on registers, so data stored in memory must be moved to a register before it can be processed. With this combination the program can access a large amount of data fairly quickly. RISC-V uses a byte-addressable memory, each byte in memory has a unique address, and a 32-bit word consists of four 8-bit bytes, so each word address is a multiple of 4.

Each instruction is 32 bits (4 bytes) long, because of this the address of subsequent instructions increases by four. The Program Counter (PC), maintains track of the current instruction. After each instruction completes, the PC holds the memory address of the current instruction and increments by four so that the processor can read or fetch the next instruction from memory.

2.1 RV32I instruction implementation

The assignment started from the basis of a single-cycle processor with some instructions already implemented, being the cases of the instructions:

JAL BEQ LW SW ADDI ADD SUB SLT
OR AND

The task here is to implement the rest of the instructions:

LUI	AUIPC	JALR	SRA	BNE	BLT	BGE	BLTU
BGEU	LB	LH	LBU	LHU	SB	SH	SLTI
SLTIU	XORI	ORI	ANDI	SLLI	SRLI	SRAI	SLL
SLTU	XOR	SRL					

For the implementation of the remaining instructions it is necessary to take into account the RISC-V 32-bit instruction formats, since every instruction is classified in a specific type and accordingly to this the format of its instruction is different, so it is mandatory to consult Figures 1 and 2. Note that the RISC-V ISA keeps the source (rs1 and rs2) and destination (rd) registers at the same position in all formats to simplify decoding.

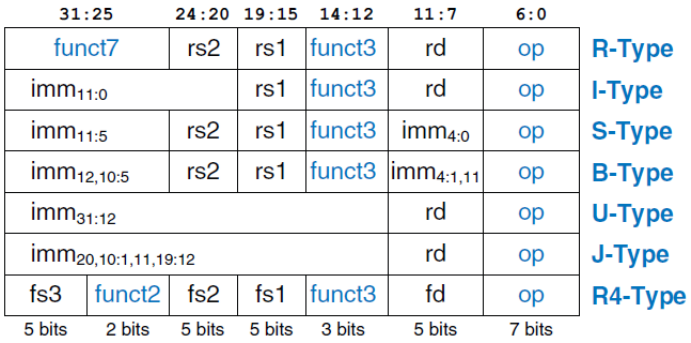

Figure 1: RISC-V 32-bit instruction formats

op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	000	–	I	lb rd, imm(rs1)	load byte	rd = SignExt([Address]$_{7:0}$)
0000011 (3)	001	–	I	lh rd, imm(rs1)	load half	rd = SignExt([Address]$_{15:0}$)
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address]$_{31:0}$
0000011 (3)	100	–	I	lbu rd, imm(rs1)	load byte unsigned	rd = ZeroExt([Address]$_{7:0}$)
0000011 (3)	101	–	I	lhu rd, imm(rs1)	load half unsigned	rd = ZeroExt([Address]$_{15:0}$)
0010011 (19)	000	–	I	addi rd, rs1, imm	add immediate	rd = rs1 + SignExt(imm)
0010011 (19)	001	0000000*	I	slli rd, rs1, uimm	shift left logical immediate	rd = rs1 << uimm
0010011 (19)	010	–	I	slti rd, rs1, imm	set less than immediate	rd = (rs1 < SignExt(imm))
0010011 (19)	011	–	I	sltiu rd, rs1, imm	set less than imm. unsigned	rd = (rs1 < SignExt(imm))
0010011 (19)	100	–	I	xori rd, rs1, imm	xor immediate	rd = rs1 ^ SignExt(imm)
0010011 (19)	101	0000000*	I	srli rd, rs1, uimm	shift right logical immediate	rd = rs1 >> uimm
0010011 (19)	101	0100000*	I	srai rd, rs1, uimm	shift right arithmetic imm.	rd = rs1 >>> uimm
0010011 (19)	110	–	I	ori rd, rs1, imm	or immediate	rd = rs1 | SignExt(imm)
0010011 (19)	111	–	I	andi rd, rs1, imm	and immediate	rd = rs1 & SignExt(imm)
0010111 (23)	–	–	U	auipc rd, upimm	add upper immediate to PC	rd = {upimm, 12'b0} + PC
0100011 (35)	000	–	S	sb rs2, imm(rs1)	store byte	[Address]$_{7:0}$ = rs2$_{7:0}$
0100011 (35)	001	–	S	sh rs2, imm(rs1)	store half	[Address]$_{15:0}$= rs2$_{15:0}$
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	[Address]$_{31:0}$= rs2
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2
0110011 (51)	000	0100000	R	sub rd, rs1, rs2	sub	rd = rs1 − rs2
0110011 (51)	001	0000000	R	sll rd, rs1, rs2	shift left logical	rd = rs1 << rs2$_{4:0}$
0110011 (51)	010	0000000	R	slt rd, rs1, rs2	set less than	rd = (rs1 < rs2)
0110011 (51)	011	0000000	R	sltu rd, rs1, rs2	set less than unsigned	rd = (rs1 < rs2)
0110011 (51)	100	0000000	R	xor rd, rs1, rs2	xor	rd = rs1 ^ rs2
0110011 (51)	101	0000000	R	srl rd, rs1, rs2	shift right logical	rd = rs1 >> rs2$_{4:0}$
0110011 (51)	101	0100000	R	sra rd, rs1, rs2	shift right arithmetic	rd = rs1 >>> rs2$_{4:0}$
0110011 (51)	110	0000000	R	or rd, rs1, rs2	or	rd = rs1 | rs2
0110011 (51)	111	0000000	R	and rd, rs1, rs2	and	rd = rs1 & rs2
0110111 (55)	–	–	U	lui rd, upimm	load upper immediate	rd = {upimm, 12'b0}
1100011 (99)	000	–	B	beq rs1, rs2, label	branch if =	if (rs1 == rs2) PC = BTA
1100011 (99)	001	–	B	bne rs1, rs2, label	branch if ≠	if (rs1 ≠ rs2) PC = BTA
1100011 (99)	100	–	B	blt rs1, rs2, label	branch if <	if (rs1 < rs2) PC = BTA
1100011 (99)	101	–	B	bge rs1, rs2, label	branch if ≥	if (rs1 ≥ rs2) PC = BTA
1100011 (99)	110	–	B	bltu rs1, rs2, label	branch if < unsigned	if (rs1 < rs2) PC = BTA
1100011 (99)	111	–	B	bgeu rs1, rs2, label	branch if ≥ unsigned	if (rs1 ≥ rs2) PC = BTA
1100111 (103)	000	–	I	jalr rd, rs1, imm	jump and link register	PC = rs1 + SignExt(imm), rd = PC + 4
1101111 (111)	–	–	J	jal rd, label	jump and link	PC = JTA, rd = PC + 4

Figure 2: RISC-V 32-bit instructions

Based on the table in the Figure 2 and the single-cycle processor, the alterations started in the main decoder. It was necessary to add the opcodes that were left and also to increase the size of some of the control signals to support all the RV32I instructions:

- the *ImmSrc* was changed to 3 bits;
- the *ResultSrc* was changed to 3 bits;
- the *Jump* was changed to 2 bits;

These alterations were also made throughout the project and the current main decoder is the presented below.

```
module maindec (
  op,
  ResultSrc,
  MemWrite,
  Branch,
  ALUSrc,
  RegWrite,
  Jump,
  ImmSrc,
  ALUOp
);
  input wire [6:0] op;
  output wire [2:0] ResultSrc;
  output wire MemWrite;
  output wire Branch;
  output wire ALUSrc;
  output wire RegWrite;
  output wire [1:0] Jump;
  output wire [2:0] ImmSrc;
  output wire [1:0] ALUOp;
  reg [13:0] controls;

  assign {RegWrite, ImmSrc, ALUSrc, MemWrite, ResultSrc, Branch,
   ALUOp, Jump} = controls;
  always @(*)
    case (op)
      7'b0000011: controls = 14'b1_000_1_0_001_0_00_00;
      7'b0100011: controls = 14'b0_001_1_1_000_0_00_00;
      7'b0110011: controls = 14'b1_xxx_0_0_000_0_10_00;
      7'b1100011: controls = 14'b0_010_0_0_000_1_01_00;
      7'b1100111: controls = 14'b1_100_0_0_010_0_00_11;
      7'b0010011: controls = 14'b1_000_1_0_000_0_10_00;
      7'b1101111: controls = 14'b1_011_0_0_010_0_00_01;
      7'b0110111: controls = 14'b1_101_1_0_011_0_00_00;
      7'b0010111: controls = 14'b1_101_1_0_100_0_00_00;
      default: controls = 14'b0_000_0_0_000_0_00_00;
    endcase
endmodule
```

Listing 1: maindec.v

2.1.1 Implementation of immediate and register instructions

The immediate and register instructions are very similar with regard to the operations they perform, the main difference is that the second source of immediate type instructions is an immediate and in the register type instructions it is a register.

Both of these types support: logical instructions, shift instructions and set less than instructions. The set less than instructions can be signed or unsigned. The instructions operations are all treated as signed unless the set less than unsigned

instructions.

To implement the I-type and R-type instructions it was necessary to change the implementation of the ALU decoder. The number of bits of the *ALUControl* was increased by one, the updated ALUControl has now 4 bits, to support all of the operations that are needed and this alteration was also made throughout the project.

```
output reg [3:0] ALUControl;
```

The logical instructions that are present in the RV32I are *or*, *xor* and *and*. The shift instructions that are present in the RV32I are *sll*(shift left logical), *slli*(shift left logical immediate),*srl*(shift right logical),*srli*(shift right logical immediate),*sra*(shift right arithmetic) and *srai*(shift right arithmetic immediate). The set less then instructions that are present in the RV32I are *sltu*(set less then unsigned), *stli*(set less than immediate) and *stliu*(set less than immediate unsigned).

In the ALU decoder according to the *funct3* of these instructions the ALUControl sign associated with each of them was assigned. It should be noted that the same type of instructions(shift, logical and set less then) on the I-types and R-types have the same *funct3* and are viewed in the same way by the ALU.

It should also be noted that the *add* and *sub* have the same *funct3* and the *shift right arithmetic* and *shift right logical* also have the same *funct3*.

To distinguish the *add* and *sub* it was implemented this, that when it is a *sub* the *RtypeSub* is set.

```
assign RtypeSub = funct7b5 & opb5;
```

To distinguish the *shift right arithmetic* and *shift right logical* it was implemented this, that when it is a *shift right arithmetic* the *funct7b5* is set.

```
if (funct7b5)
ALUControl = 4'b1000;  // shift right arithmetic
else
ALUControl = 4'b0111;  // shift right logical
```

The updated implementation of the ALU decoder is presented below.

```
  always @(*)
    case (ALUOp)
            2'b00: ALUControl = 4'b0000;
            2'b01: ALUControl = 4'b0001;
            default:
                case (funct3)
                    3'b000:
                        if (RtypeSub)
                            ALUControl = 4'b0001; // sub
                        else
                            ALUControl = 4'b0000; // add
                    3'b001: ALUControl = 4'b0110; // shift left
   logical
```

```
                    3'b010: ALUControl = 4'b0101; // set less than
                    3'b011: ALUControl = 4'b1001; // set less then
    unsigned
                    3'b100: ALUControl = 4'b0100; // xor
                    3'b101:
                        if (funct7b5)
                            ALUControl = 4'b1000;  // shift right
    arithmetic
                        else
                            ALUControl = 4'b0111;  // shift right
    logical

                    3'b110: ALUControl = 4'b0011; // or
                    3'b111: ALUControl = 4'b0010; // and
                    default: ALUControl = 4'bxxxx;
                endcase
        endcase
```

Listing 2: *aludecoder.v*

The next step was to edit the ALU, since it was unsigned and the instructions should be all treated as signed unless the instructions was unsigned. So it was necessary to add the keyword *signed* to the sources for the ALU.

```
    input wire signed [31:0] SrcA;
    input wire signed [31:0] SrcB;
```

But there are the unsigned instructions and to handle them is also added unsigned sources for the ALU.

```
    wire [31:0] SrcA_u;
    wire [31:0] SrcB_u;
```

The signed value is converted in an unsigned in order to set the unsigned sources when needed.

```
    assign SrcA_u = SrcA;
    assign SrcB_u = SrcB;
```

According to the instruction the sources will be chosen.

Finally according to the ALUControl defined in the ALU decoder the ALU implements the operations.

```
always @(*)
        case (ALUControl)
            4'b0000: ALUResult = SrcA + SrcB;   // add
            4'b0001: ALUResult = SrcA - SrcB;   // sub
            4'b0010: ALUResult = SrcA & SrcB;   // and
            4'b0011: ALUResult = SrcA | SrcB;   // or
            4'b0100: ALUResult = SrcA ^ SrcB;   // xor
            4'b0101: ALUResult = SrcA < SrcB;   // set less then
            4'b0110: ALUResult = SrcA << SrcB;  // shift left
    logical
```

```
            4'b0111: ALUResult = SrcA >> SrcB;   // shift right logical
            4'b1000: ALUResult = SrcA >>> SrcB;  // shift right arithmetic
            4'b1001: ALUResult = SrcA_u < SrcB_u;// set less then unsigned
            default: ALUResult = ALUResult;
        endcase
```

2.1.2 Implementation of branch instructions

The branch instructions are responsible of comparing the value between 2 registers and decide to jump or not based on the outcome of the comparison. In order to implement the branch instructions it was necessary to implement a better ALU first. In the ALU it was added 4 flags that would identify if the result is zero (Zero), if the operation overflowed (Overflow), if there is carry (Carry) and if the result was negative as illustrated in the figure.

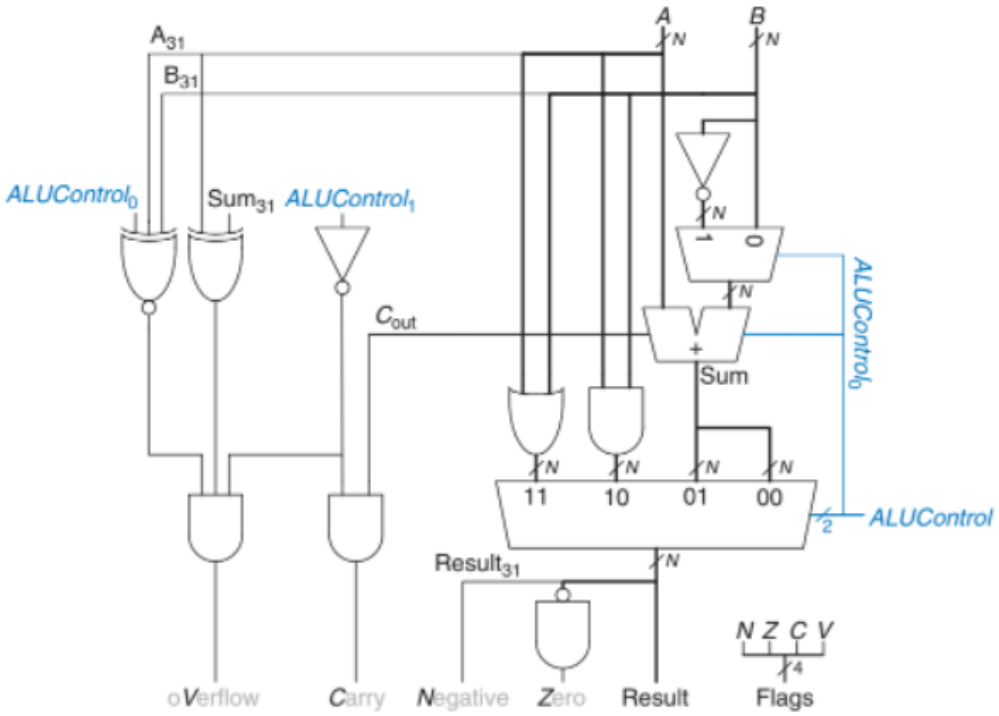

Figure 3: Signed ALU implementation

```
    assign Verflow = ~(SrcB_u[31] ^ SrcA_u[31] ^ ALUControl[0]) & (SrcA_u[31] ^ Sum[31]) & ALUControl[1];
    assign Carry = ~ALUControl[1] & Cout;
```

```
    assign Negative = ALUResult[31];
    assign Zero = ALUResult == 32'b00000000000000000000000000000000;
```

The next step was to create a mux that checks for a condition associated with the branch type. For that the following table was implemented in the control unit.

Table 5.2 Signed and unsigned comparisons

Comparison	Signed	Unsigned
=	Z	Z
≠	$\overline{Z}$	$\overline{Z}$
<	$N \oplus V$	$\overline{C}$
≤	$Z + (N \oplus V)$	$Z + \overline{C}$
>	$\overline{Z} \bullet (\overline{N \oplus V})$	$\overline{Z} \bullet C$
≥	$(\overline{N \oplus V})$	C

Figure 4: Signed at unsigned comparisons

```
    mux8 #(32) pcSrcMux(
        (Branch & Zero) | Jump[0],
        (Branch & ~Zero) | Jump[0],
        (Branch) | Jump[0],
        (Branch) | Jump[0],
        (Branch & Negative ^ Verflow) | Jump[0],
        (Branch & ~(Negative ^ Verflow)) | Jump[0],
        (Branch & Carry) | Jump[0],
        (Branch & ~Carry) | Jump[0],
        funct3,
        PCSrc[0]
    );
```

After the alu was implemented we can implement the rest of the branch set. For that we must take the comparison result and connect it to the PCSrc so it jumps whenever the condition is true or keeps the program flow if the branch is not taken.

2.1.3 Implementation of store instructions

The store instructions are responsible of taking a value from a register and move it inside the RAM. In order to store a value from a register into the dmem the value must be loaded from the regfile and moved into the dmem. Since there are different types of store the store mode selection should be made as late as possible (inside

the dmem) in order to not interfere with other instructions. With that in mind we had to first distinguish the store type, the parameter that defines the store type is the funct3, but since the name was too gereric for this application we assigned the funct3 BUS to a new BUS named LoadStrType (this BUS will be usefull both for store and load)

```
assign LdStrType=funct3;
```

Next, a switch case was created inside the dmem to select the type of store instruction.

```
always @(posedge clk)
        if (we)
            if(we==1'b1) begin
                  case(LdStrType)
                    3'b000: RAM[a[31:2]] <= {24'b0,wd[7:0]};
                    3'b001: RAM[a[31:2]] <= {16'b0,wd[15:0]};
                    3'b010: RAM[a[31:2]] <= wd[31:0];
                    3'b100: RAM[a[31:2]] <= {RAM[a[31:2]][31:8],wd
    [7:0]};
                    3'b101: RAM[a[31:2]] <= {RAM[a[31:2]][31:16],wd
    [15:0]};
                    default: RAM[a[31:2]] <= 32'b0;
                  endcase
              end else begin
                RAM[a[31:2]] <= wd;
              end
```

With that implemented is now possible to select how the value will be stored inside the dmem.

2.1.4 Implementation of load instructions

The load instructions are responsible of taking a value from the RAM and storing it inside a register in the regfile. As with the store instructions, in the load instructions we will implement the load selection mode in the furthest stage of the instruction (inside the regfile). For that it will be necessary to use the LoadStrType previously mentioned. The regfile switch case is implemented the same way as the one in the dmem.

```
always @(posedge clk)
        if (we3)
          if(ResultSrc==2'b01) begin
              case(LdStrType)
                3'b000: rf[a3] <= {24'b0,wd3[7:0]};
                3'b001: rf[a3] <= {16'b0,wd3[15:0]};
                3'b010: rf[a3] <= wd3[31:0];
                3'b100: rf[a3] <= {rf[a3][31:8],wd3[7:0]};
                3'b101: rf[a3] <= {rf[a3][31:16],wd3[15:0]};
                default: rf[a3] <= 32'b0;
```

```
                endcase
            end else begin
              rf[a3]=wd3;
            end
```

2.1.5 Implementation of jalr instruction

The jarl instruction preforms a jump and saves the sum of the original PC with a value from a register in a given register. In order to implement the instruction its first necessary to implement the sum of the PC with the register value, for that it was used an adder with the imputs of PC and the resister value (from the regfile). The output PCImm is then added as an entry in the resultmux to be able to be selected when the instruction is called. The result is then written back into the regfile.

```
adder PCaddimm(
    PC,
    ImmExt,
    PCImm
);
```

```
mux5 #(32) resultmux(
    ALUResult,
    ReadData,
    PCPlus4,
    ImmExt,
    PCImm,
    ResultSrc,
    Result
);
```

The jump side of the jarl occurs exactly the same way as the already implemented jal

2.1.6 Implementation of lui instruction

The lui instruction is a special type of load that takes a 20bit immediate and appends zeros at the end, then saves the result in a given register. In order to implement this instruction we first added a control state for the extender that extracts the 20bit immediate from the instruction.

```
always @(*)
        case (immsrc)
            3'b000: immext = {{20 {instr[31]}}, instr[31:20]};
            3'b001: immext = {{20 {instr[31]}}, instr[31:25], instr
    [11:7]};
            3'b010: immext = {{20 {instr[31]}}, instr[7], instr
    [30:25], instr[11:8], 1'b0};
            3'b011: immext = {{12 {instr[31]}}, instr[19:12], instr
    [20], instr[30:21], 1'b0};
```

```
            3'b100: immext = {20'b0, instr[31:20]};
            3'b101: immext = {instr[31:12], 12'b0};
            default: immext = 32'bxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx;
        endcase
```

Then we added the option to take the result directly from the immext, allowing to save in the given register.

```
mux5 #(32) resultmux(
    ALUResult,
    ReadData,
    PCPlus4,
    ImmExt,
    PCImm,
    ResultSrc,
    Result
);
```

2.1.7 Implementation of auipc instruction

The auipc instruction is an extension from the lui, instead of saving the value directly in a register the value is added to the PC and then saved in the given register. For that we created an adder that sums the immediate with the PC and then the PCImm was added to the resultmux.

```
adder PCaddimm(
    PC,
    ImmExt,
    PCImm
);
```

```
mux5 #(32) resultmux(
    ALUResult,
    ReadData,
    PCPlus4,
    ImmExt,
    PCImm,
    ResultSrc,
    Result
);
```

with that the result can be stored in the given register.

2.2 Tests

2.2.1 Type R

2.2.1.1 SLL - Shift Left Logical

Instruction	Operation
sll rd,rs1,rs2	rd = rs1 << rs2 (4:0)

In this instruction the test was made of set 2 in register 9, set 3 in register 4, and the respective result of the shift left in register 6. In this case the result should be b00000000000000000000000000010000.

addi x9,x0,2

Expected Data Address: 9
Expected Write Data: 2
Machine Code: 0x00200493

addi x4,x0,3

Expected Data Address: 4
Expected Write Data: 3
Machine Code: 0x00300213

sll x6,x9, x4

Expected Data Address: 6
Expected Write Data: b00000000000000000000000000010000
Machine Code: 0x00449333

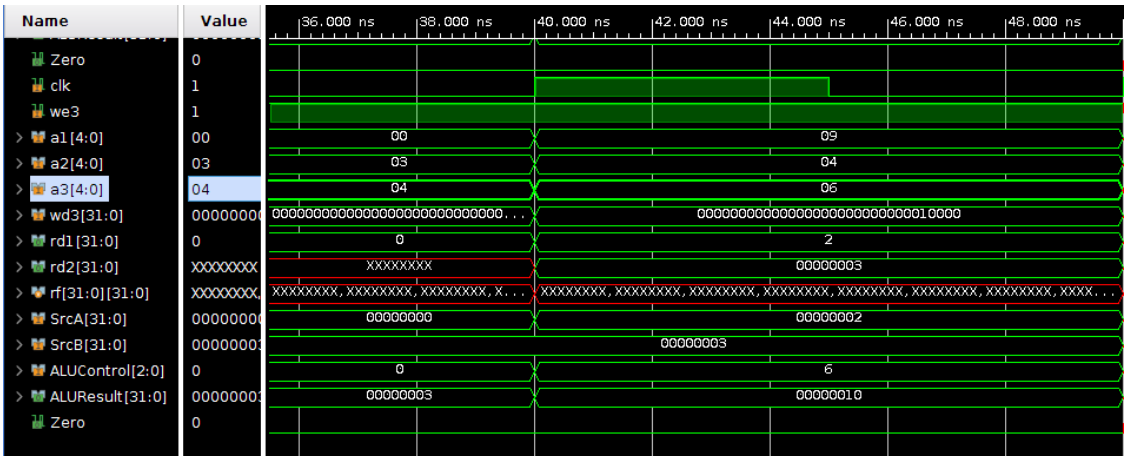

Figure 5: SLL- Instruction Test

2.2.1.2 SLTU - Set Less Then Unsigned

Instruction	Operation
sltu rd, rs1, rs2	rd = (rs1 < rs2)

In this instruction the test was made of set -5 in register 9, set 4 in register 4, and the respective result of the comparison in register 6. In this case the result should be 0. Since sltiu will see the numbers as unsigned in reality -5 will be seen as 4294967291. This will make the result of the comparison 0, since 4294967291 is not less than 4.

addi x9,x0,-5

Expected Data Address: 9
Expected Write Data: -5
Machine Code: 0xFFB00493

addi x4,x0,4

Expected Data Address: 4
Expected Write Data: 4
Machine Code: 0x00400213

sltu x6,x9, x4

Expected Data Address: 6
Expected Write Data: 0
Machine Code: 0x0044B333

Figure 6: SLTU- Instruction Test

2.2.1.3 XOR

Instruction	Operation
xor rd, rs1, rs2	rd = rs1 ∧ rs2

A xor was made with the value of 2 placed in register 9 and a 3 placed in register 4 and the result was stored in the register 6.

addi x9,x0,2

Expected Data Address: 9
Expected Write Data: 2
Machine Code: 0x00200493

addi x4,x0,3

Expected Data Address: 4
Expected Write Data: 3
Machine Code: 0x00300213

xor x6,x9, x4

Expected Data Address: 6
Expected Write Data: 1
Machine Code: 0x0044C333

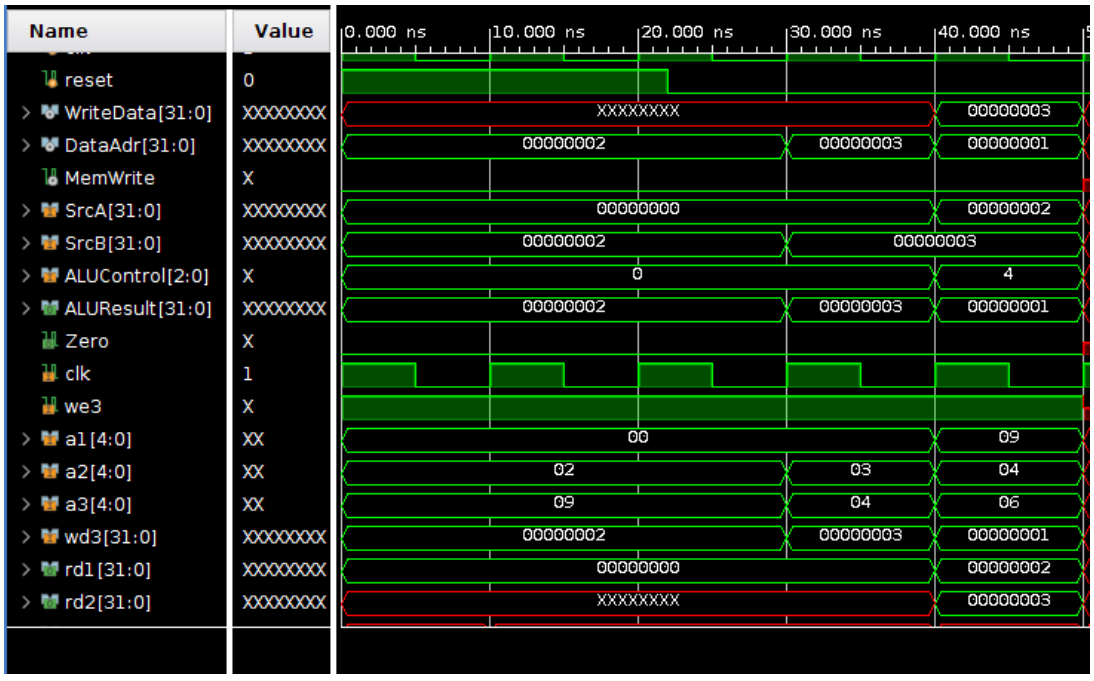

Figure 7: XOR- Instruction Test

2.2.1.4 SRL - Shift Right Logical

Instruction	Operation
srl rd, rs1, rs2	rd = rs1 >> rs2(4:0)

In this instruction the test was made of set 10 in register 9, set 3 in register 4, and the respective result of the shift right in register 6. In this case the result should be b00000000000000000000000000000001.

addi x9,x0,10

Expected Data Address: 9
Expected Write Data: 10
Machine Code: 0x00A00493

addi x4,x0,3

Expected Data Address: 4
Expected Write Data: 3
Machine Code: 0x00300213

srl x6,x9, x4

Expected Data Address: 6
Expected Write Data: b00000000000000000000000000000001
Machine Code: 0x0044D333

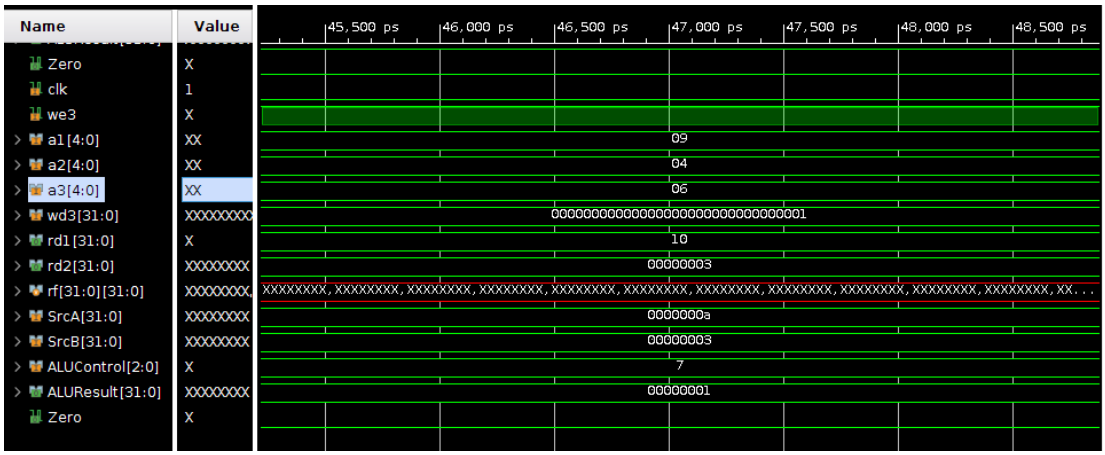

Figure 8: SRL- Instruction Test

2.2.1.5 SRA - Shift Right Arithmetic

Instruction	Operation
sra rd, rs1, rs2	rd = rs1 >>> rs2(4:0)

In this instruction the test was made of set -2 in register 9, set 3 in register 4, and the respective result of the shift right in register 6. In this case the result should be b11111111111111111111111111111111, because the arithmetic shift makes the shift and replace the positions needed according with the signal of the number, so if it is negative it adds 1s and if it is positive it adds 0s.

addi x9,x0,-2

Expected Data Address: 9
Expected Write Data: -2
Machine Code: 0xFFE00493

addi x4,x0,3

Expected Data Address: 4
Expected Write Data: 3
Machine Code: 0x00300213

sra x6,x9, x4

Expected Data Address: 6
Expected Write Data: b11111111111111111111111111111111
Machine Code: 0x4044D333

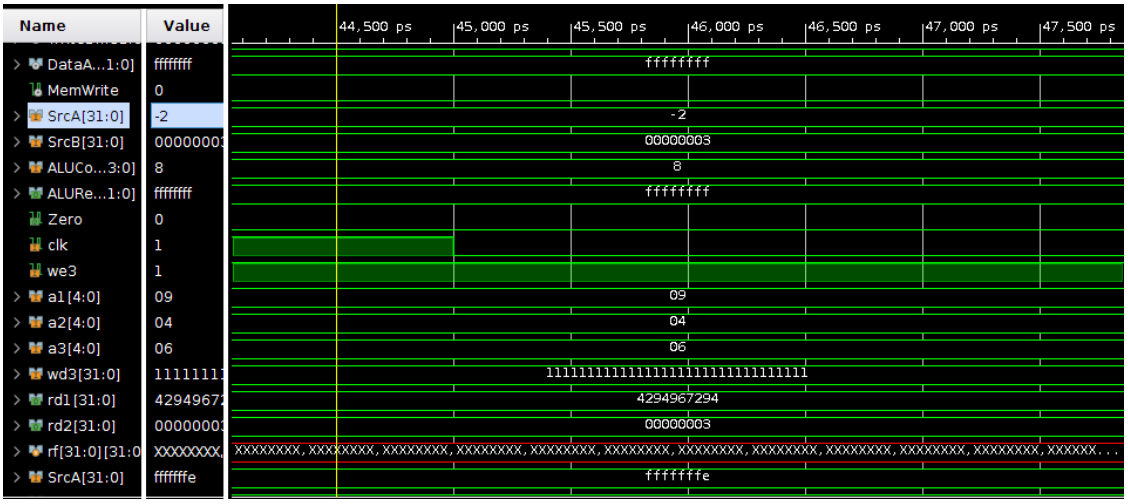

Figure 9: SRA- Instruction Test

2.2.2 Type I

2.2.2.1 SLTI - Set Less Than Immediate

Instruction	Operation
slti rd, rs1, imm	rd = (rs1 < SignExt(imm))

In order to test this instruction, a comparison was made between the value of -5, which was set in register 9 and -4, in the immediate. Since -5 is smaller than -4 the instruction should set the destination register to 1, which in this case was 6.

To be able to test this instruction, it was necessary to use a sum instruction to set register 9 to a value of -5.

addi x9,x0,-5

Machine Code: 0xFFB00493
Expected Data Address: 9
Expected Write Data: -5

slti x6, x9, -4

Machine Code: 0xFFC4A313
Expected Data Address: 6
Expected Write Data: 1

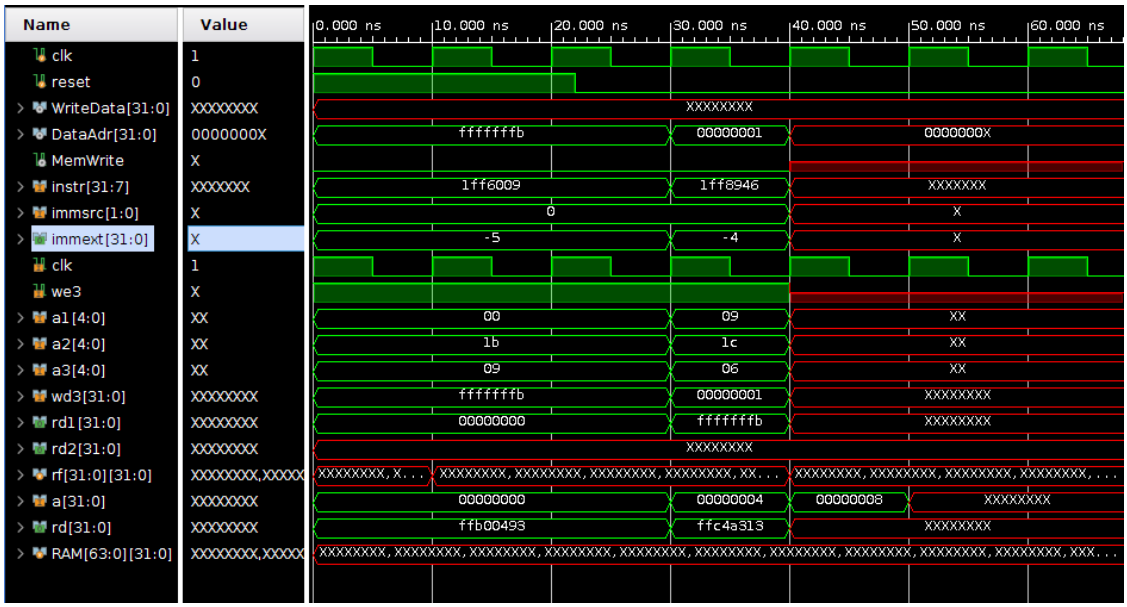

Figure 10: SLTI- Instruction Test

2.2.2.2 SLTIU - Set Less Than Immediate Unsigned

Instruction	Operation
sltiu rd, rs1, imm	rd = (rs1 < SignExt(imm))

This test was performed between the value of -5, which was set in register 9, and 4, in the immediate. Since sltiu will see the numbers as unsigned in reality -5 will be seen as 4294967291. This will make the result of the comparison 0, since 4294967291 is not less than 4.

addi x9,x0,-5

Expected Data Address: 9
Expected Write Data: -5
Machine Code: 0xFFB00493

sltiu x6, x9, 4

Expected Data Address: 6
Expected Write Data: 0
Machine Code: 0x0044B313

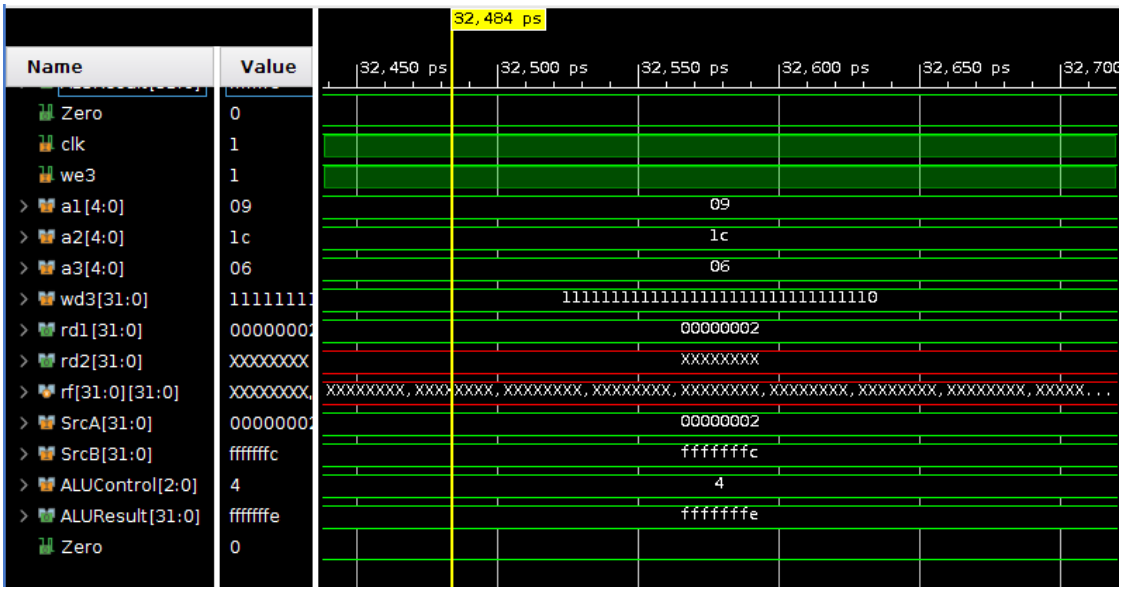

Figure 11: SLTIU- Instruction Test

2.2.2.3 XORI - Xor Immediate

A xor was made with the value of 2 placed in register 9 and an immediate with the value -4 and the result was stored in the register 6.

Instruction	Operation
xori rd, rs1, imm	rd = (rs1 ∧ SignExt(imm))

addi x9,x0,2

Expected Data Address: 9
Expected Write Data: 2
Machine Code: 0x00200493

xori x6, x9, -4

Expected Data Address: 6
Expected Write Data: 11111111111111111111111111110
Machine Code: 0xFFC4C313

Figure 12: SLTIU- Instruction Test

2.2.2.4 ORI - Or Immediate

Instruction	Operation
ori rd, rs1, imm	rd = rs1 | SignExt(imm)

An or was made between the value 2 in register 9 and the immediate value -4 and the result was stored in the register 6.

addi x9,x0,2

Expected Data Address: 9
Expected Write Data: 2
Machine Code: 0x00200493

ori x6, x9, -4

Expected Data Address: 6
Expected Write Data: 11111111111111111111111111111110
Machine Code: 0xFFC4E313

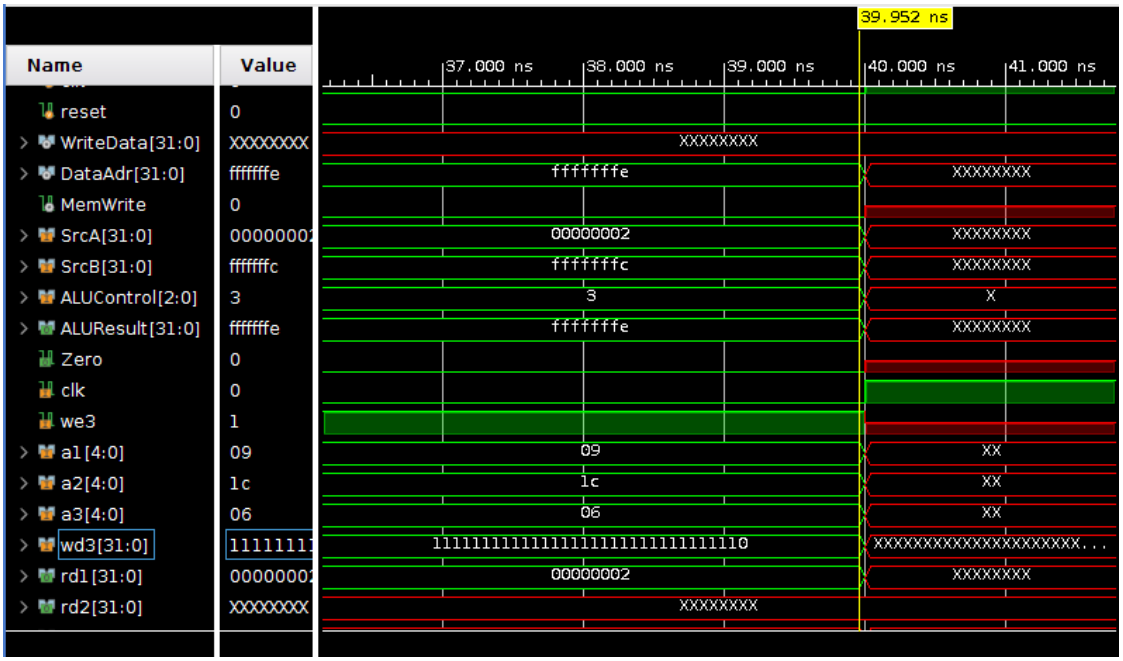

Figure 13: ORI- Instruction Test

2.2.2.5 ANDI - And Immediate

Instruction	Operation
andi rd, rs1, imm	rd = (rs1 & SignExt(imm))

An and was made between the value 2 in register 9 and the immediate value -4 and the result was stored in the register 6.

addi x9,x0,2

Expected Data Address: 9
Expected Write Data: 2
Machine Code: 0x00200493

andi x6, x9, -4

Expected Data Address: 6
Expected Write Data: 0
Machine Code: 0xFFC4F313

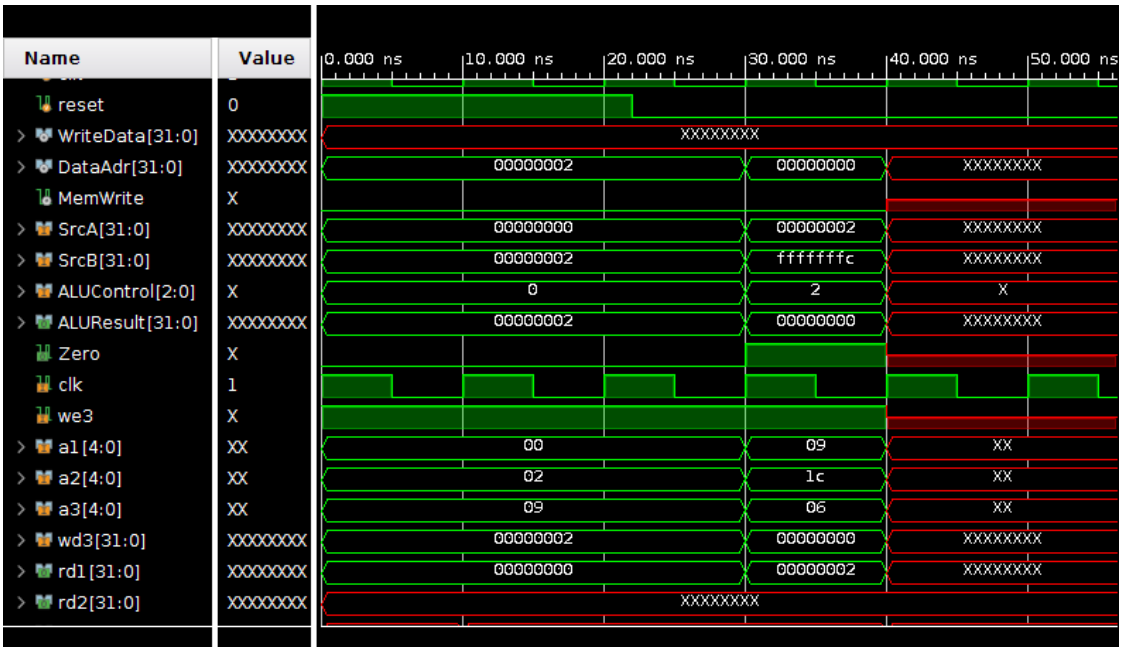

Figure 14: ANDI- Instruction Test

2.2.2.6 SLLI - Shift Left Logical Immediate

Instruction	Operation
slli rd, rs1, uimm	rd = (rs1 << uimm))

In this instruction the test was made of set 2 in register 9, set 4 in immediate, and the respective result of the shift left in register 6. In this case the result should be b00000000000000000000000000100000.

addi x9,x0,2

Expected Data Address: 9
Expected Write Data: 2
Machine Code: 0x00200493

slli x6, x9, 4

Expected Data Address: 6
Expected Write Data: 00000000000000000000000000100000
Machine Code: 0x00449313

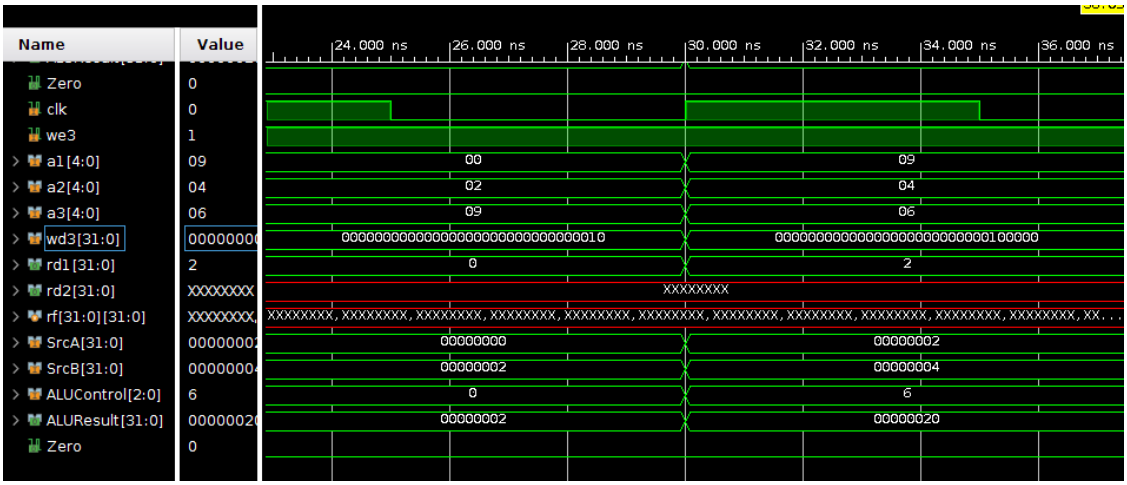

Figure 15: SLLI - Instruction Test

2.2.2.7 SRLI - Shift Right Logical Immediate

Instruction	Operation
srli rd, rs1, uimm	rd = (rs1 >> uimm))

In this instruction the test was made of set 2 in register 9, set 4 in immediate, and the respective result of the shift right in register 6. In this case the result should be 0.

addi x9,x0,2

Expected Data Address: 9
Expected Write Data: 2
Machine Code: 0x00200493

srli x6, x9, 4

Expected Data Address: 6
Expected Write Data: 000000000000
Machine Code: 0x0044D313

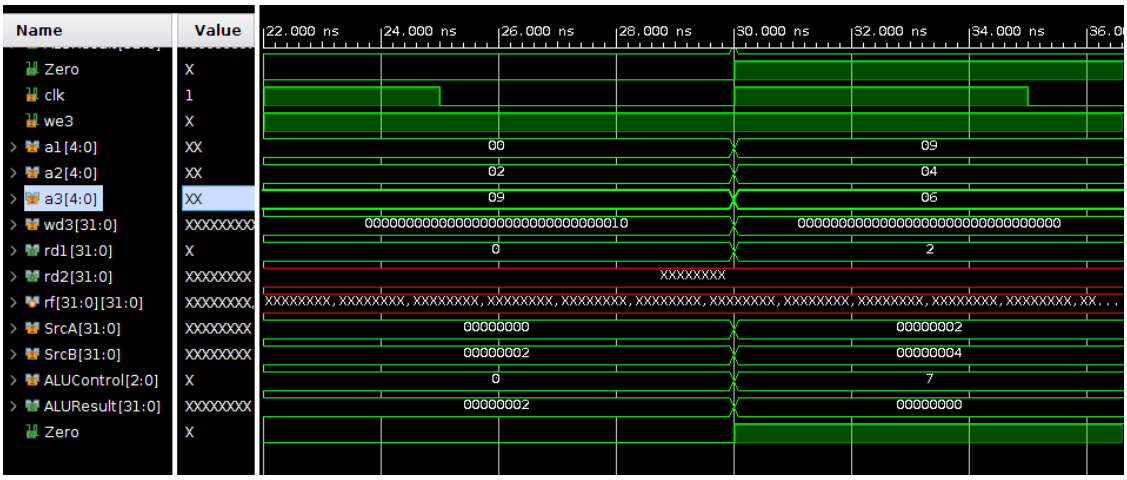

Figure 16: SRLI- Instruction Test

2.2.2.8 SRAI - Shift Right Logical Arithmetic

Instruction	Operation
srai rd, rs1, uimm	rd = (rs1 >>> uimm))

In this instruction the test was made of set -2 in register 9, set 4 in immediate, and the respective result of the shift right in register 6. In this case the result should be b11111111111, because the arithmetic shift makes the shift and replace the positions needed according with the signal of the number, so if it is negative it adds 1s and if it is positive it adds 0s.

addi x9,x0,-2

Expected Data Address: 9
Expected Write Data: -2
Machine Code: 0xFFE00493

srai x6, x9, 4

Expected Data Address: 6
Expected Write Data: 111111111111
Machine Code: 0x4044D313

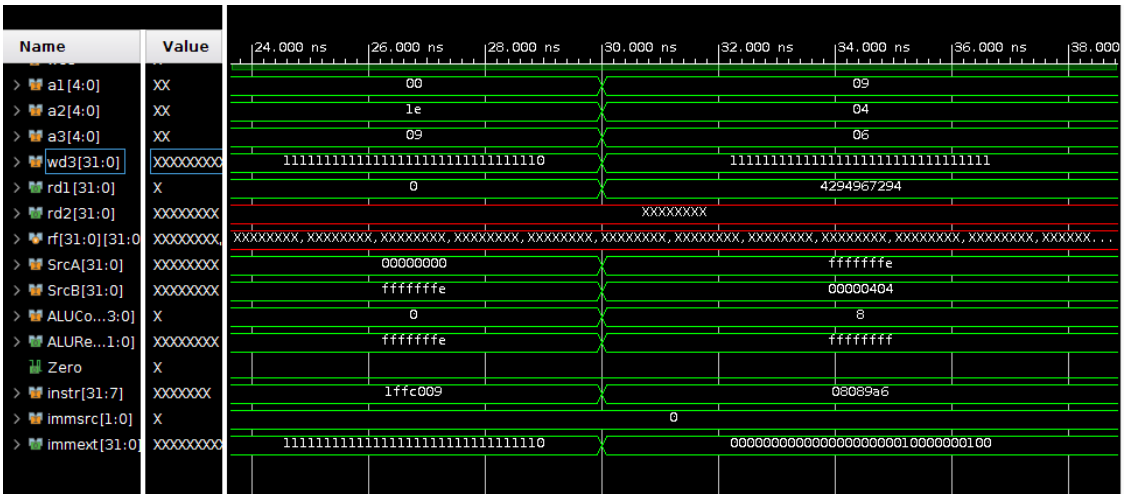

Figure 17: SRAI - Instruction Test

2.2.2.9 LB - Load Byte

Instruction	Operation
lb rd, imm(rs1)	rd = SignExt([Address] 7:0)

lb x5, 2(x0)

Machine Code: 0x00220283
Expected Register Value: 0x00000123 → 0x000000BC

2.2.2.10 LH - Load Half

Instruction	Operation
lh rd, imm(rs1)	rd = SignExt([Address] 15:0)

lh x9, 2(x0)

Machine Code: 0x00221303
Expected Register Value: 0x00000123 → 0x0000FABC

2.2.2.11 LW - Load Byte Unsigned

Instruction	Operation
lw rd, imm(rs1)	rd = [Address] 31:0

lw x7, 2(x0)

Machine Code: 0x00222383
Expected Register Value: 0x00000123 → 0xFFFFFABC

2.2.2.12 LBU - Load Byte Unsigned

Instruction	Operation
lbu rd, imm(rs1)	rd = ZeroExt([Address] 7:0)

lbu x8, 2(x0)

Machine Code: 0x00224403
Expected Register Value: 0x00000123 → 0x000001BC

2.2.2.13 LHU - Load Half Unsigned

Instruction	Operation
lhu rd, imm(rs1)	rd = ZeroExt([Address] 15:0)

lhu x6, 2(x0)

Machine Code: 0x00225483
Expected Register Value: 0x00000123 → 0x00000ABC

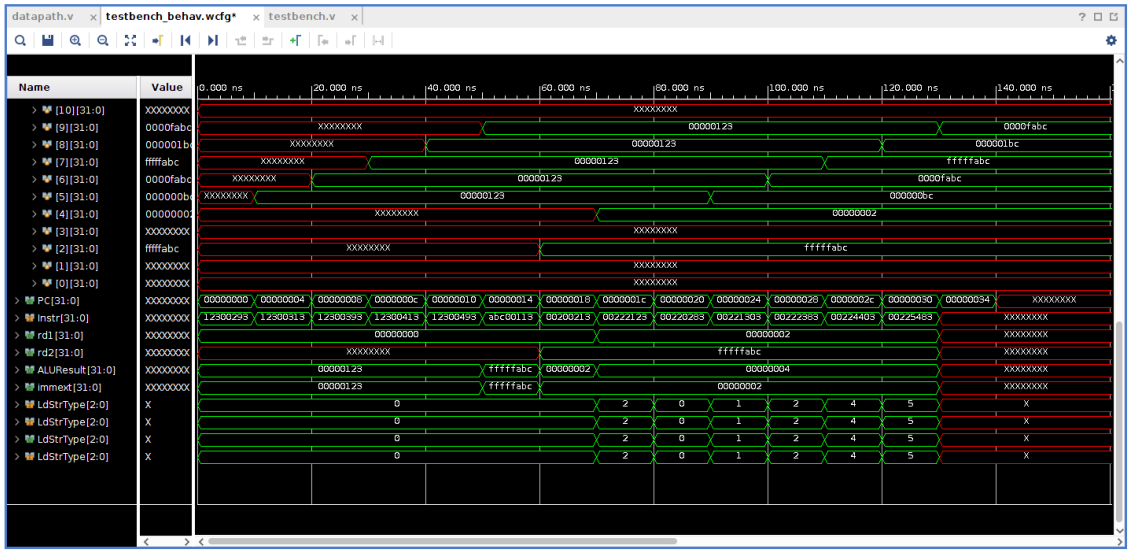

Figure 18: LOADS - Instructions Test

2.2.2.14 JALR - Jump And Link Register

Instruction	Operation
jalr rd, rs1, imm	PC = rs1 + SignExt(imm), rd = PC + 4

jalr x6, x1, 0x8

Expected Data Address: 0x6
Expected Write Data: 0x8
Expected PC: 0xC
Machine Code: 0x00808367

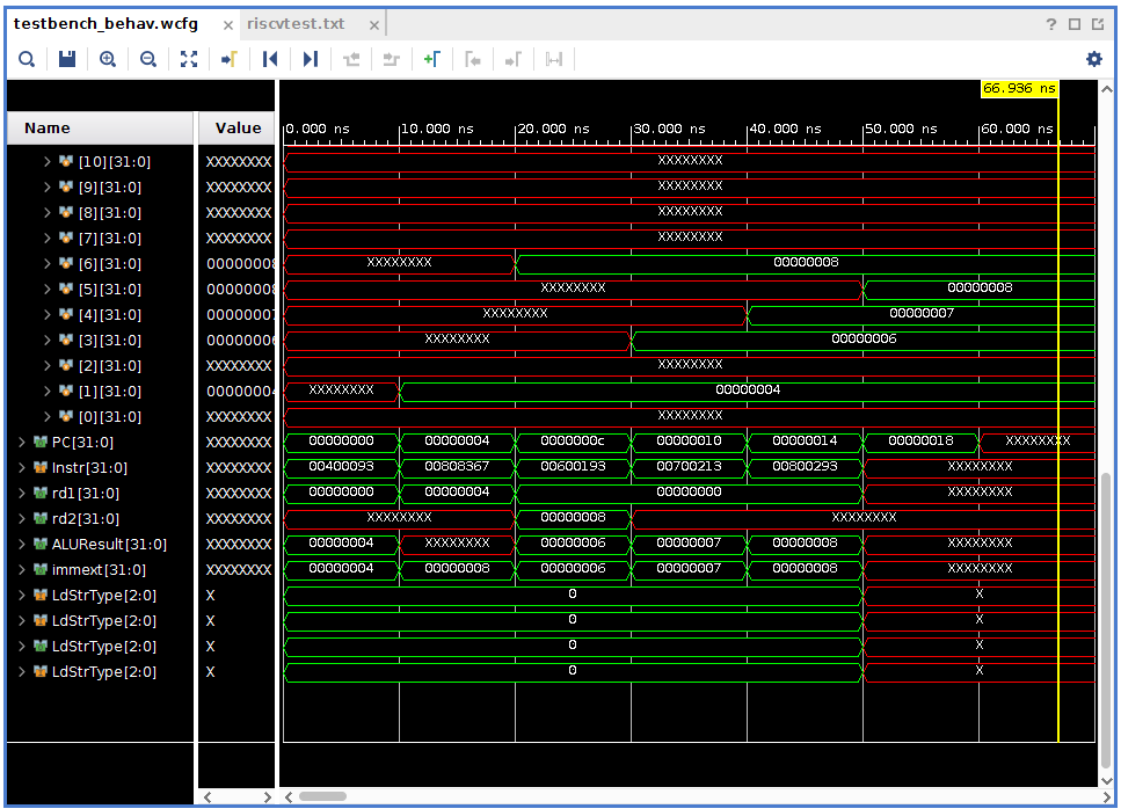

Figure 19: JARL - Instruction Test

2.2.3 Type S

2.2.3.1 SB - Store Byte

Instruction	Operation
sb rs2, imm(rs1)	[Address]((rs1+imm)(7:0)) = rs2 (7:0)

sb x1, 2(x0)

Machine Code: 0x00220123
Expected Register Value: 0x000000BC

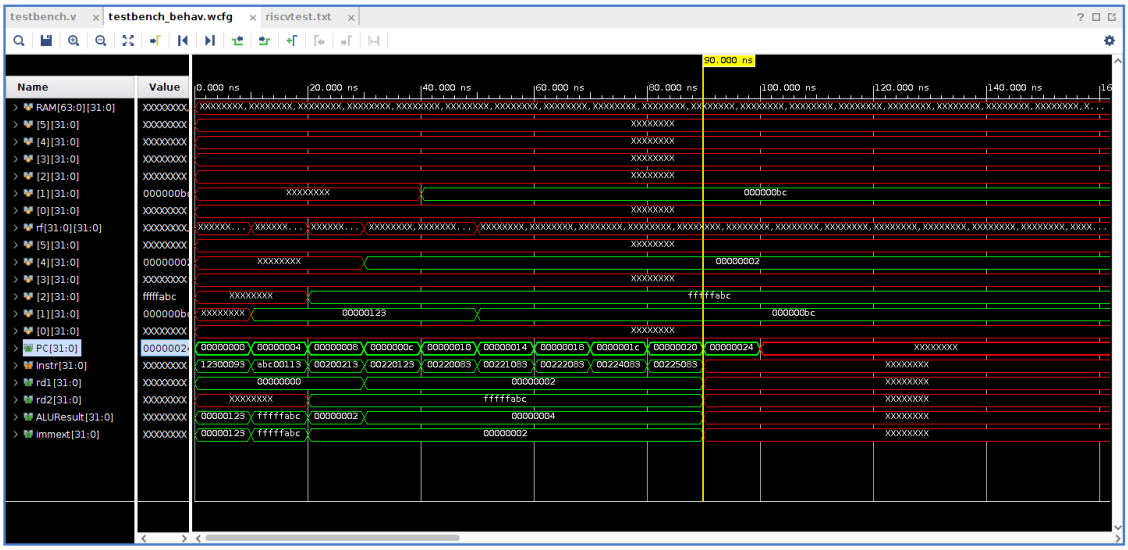

Figure 20: SB - Instruction Test

2.2.3.2 SH - Store Half

Instruction	Operation
sh rs2, imm(rs1)	[Address]((rs1+imm)(15:0)) = rs2 (15:0)

sh x1, 2(x0)

Machine Code: 0x00220123
Expected Register Value: 0x0000FABC

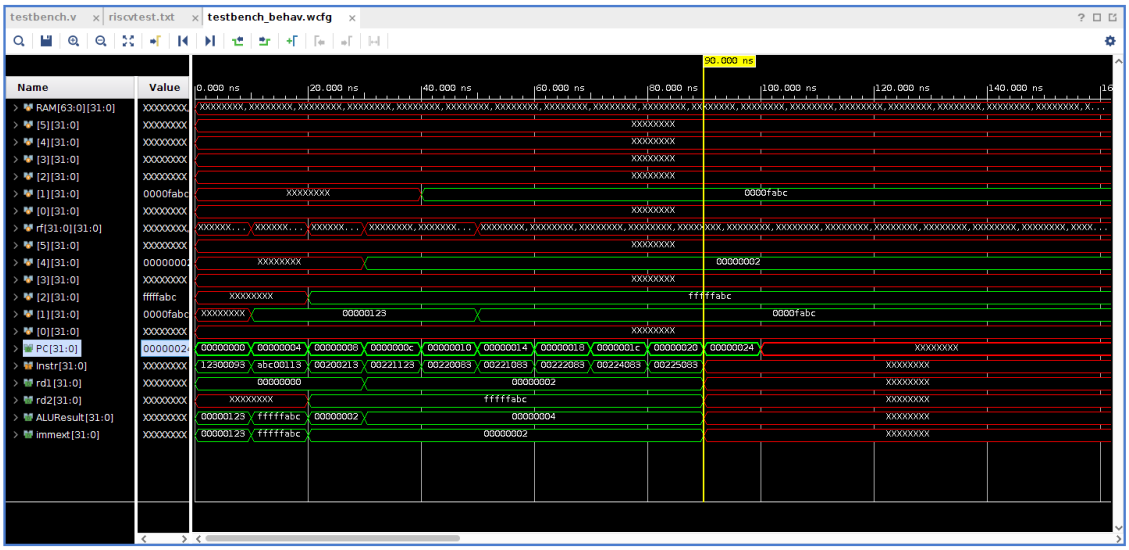

Figure 21: SH - Instruction Test

2.2.4 Type B

To test the branch instructions it was decided to make 2 test benches for each instruction. The test benches were made following the typology presented below.

```
Test Case 1:

ADDI X2, X0, #005       #Machine Code: 0x00500113
ADDI X3, X0, #00C       #Machine Code: 0x00C00193
## Bench Instruction
ADDI X1, X0, #000       #Machine Code: 0x00000093
ADDI X4, X0, #009       #Machine Code: 0x00900213

Test Case 2:

ADDI X2, X0, #FFB       #Machine Code: 0xFFB00113
ADDI X3, X0, #00C       #Machine Code: 0x00C00193
## Bench Instruction
ADDI X1, X0, #000       #Machine Code: 0x00000093
ADDI X4, X0, #009       #Machine Code: 0x00900213
```

2.2.4.1 BEQ - Branch Equal

Instruction	Operation
beq rs1, rs2, label	if (rs1 == rs2) PC = BTA

beq x2, x3, 0x08

Machine Code: 0x00310463

Case 1: X2 = 5, X3 = 0x0C

Expected Result: Not Taken
Expected PC: 0x08

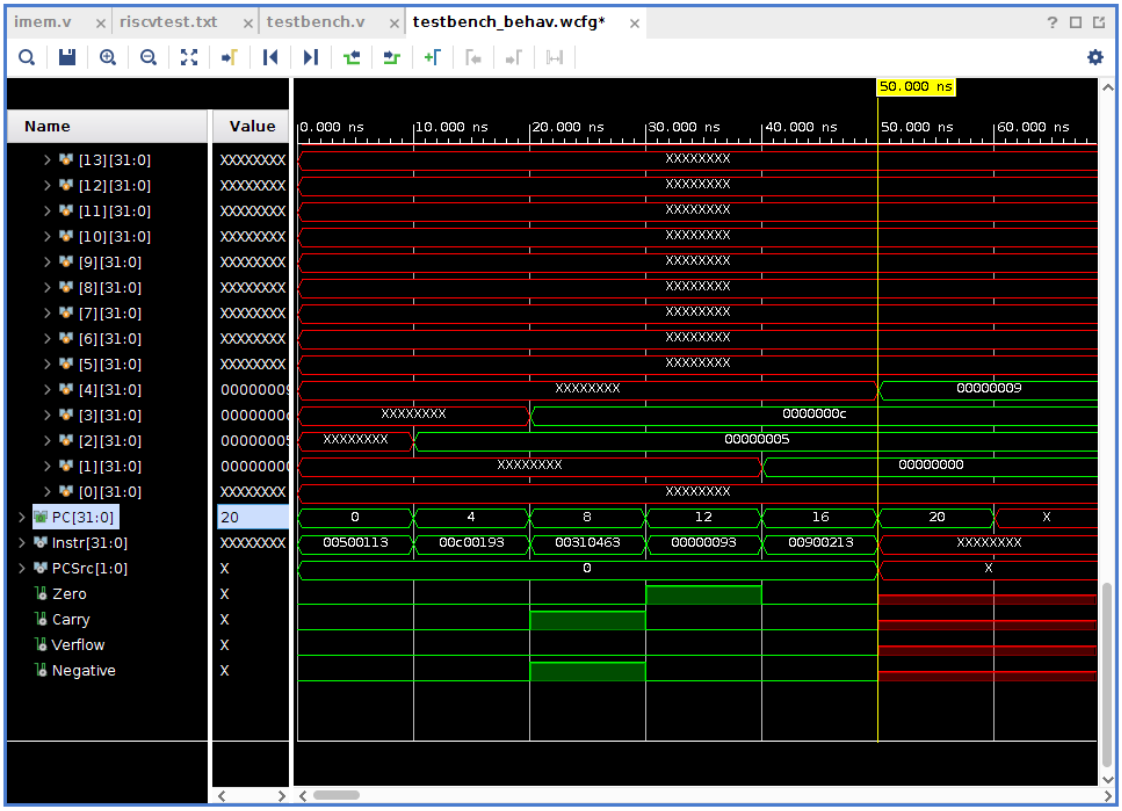

Figure 22: BEQ - Instruction Test Case 1

Case 2: X2 = - 5, X3 = 0x0C

Expected Result: Not Taken
Expected PC: 0x08

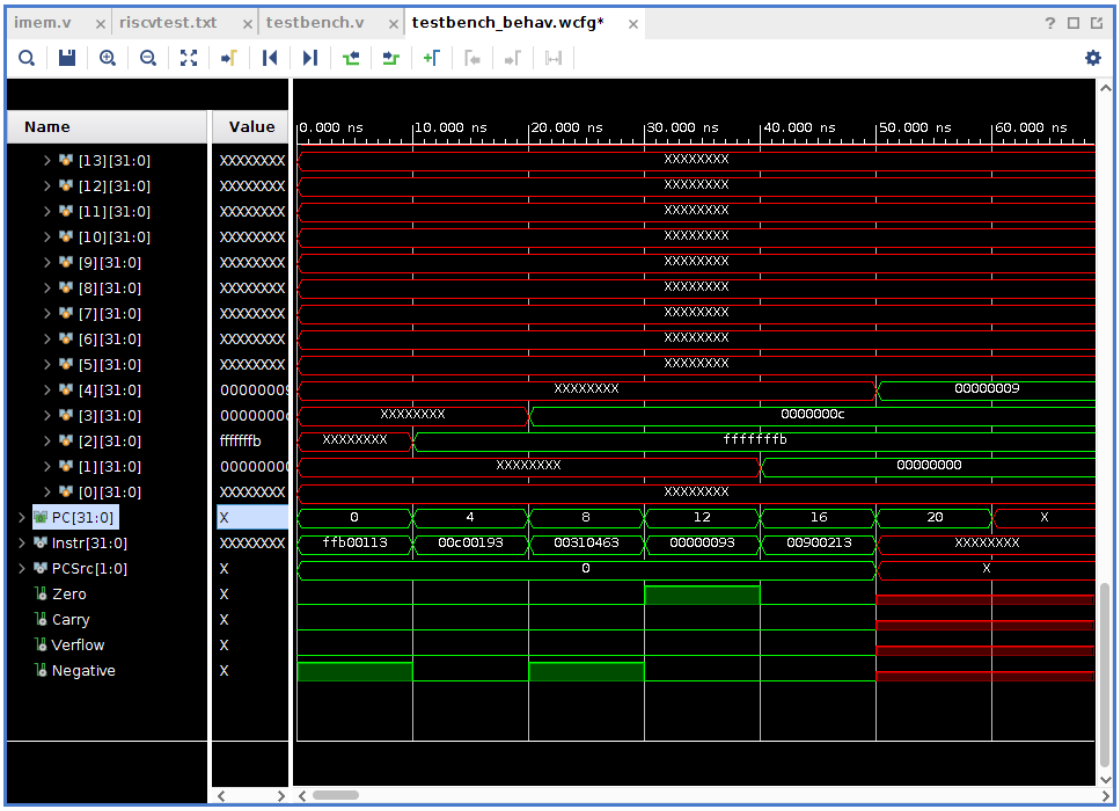

Figure 23: BEQ - Instruction Test Case 2

2.2.4.2 BNE - Branch Not Equal

Instruction	Operation
bne rs1, rs2, label	if (rs1 $\neq$ rs2) PC = BTA

bne x2, x3, 0x08

Machine Code: 0x00311463

Case 1: X2 = 5, X3 = 0x0C

Expected Result: Taken
Expected PC: 0x16

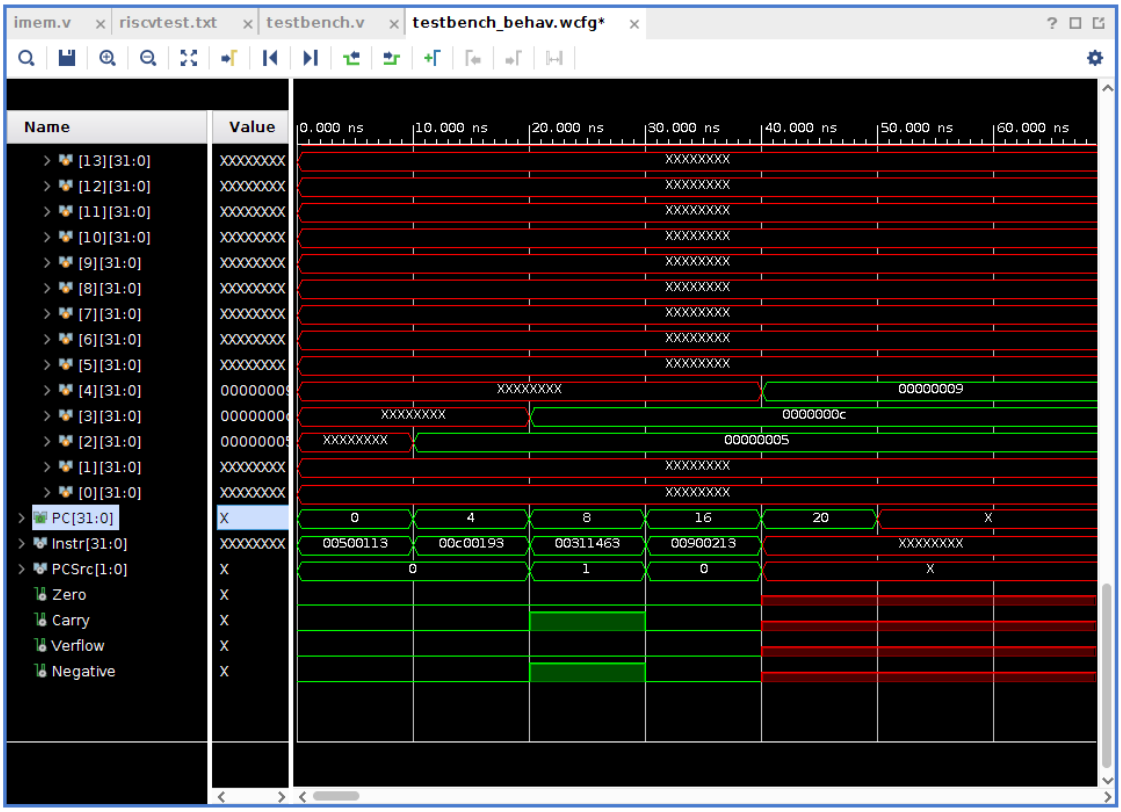

Figure 24: BNE - Instruction Test Case 1

Case 2: X2 = - 5, X3 = 0x0C

Expected Result: Taken
Expected PC: 0x16

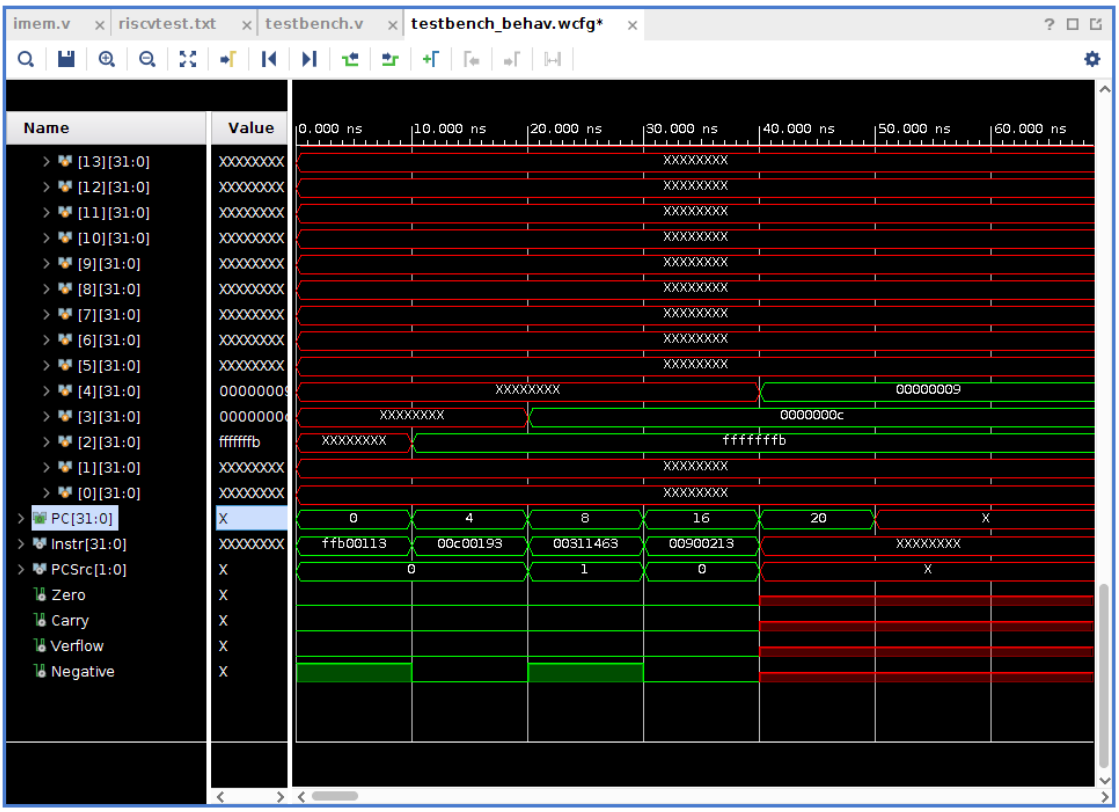

Figure 25: BNE - Instruction Test Case 2

2.2.4.3 BLT - Branch Less Than

Instruction	Operation
blt rs1, rs2, label	if (rs1 < rs2) PC = BTA

blt x2, x3, 0x08

Machine Code: 0x00314463

Case 1: X2 = 5, X3 = 0x0C

Expected Result: Taken
Expected PC: 0x16

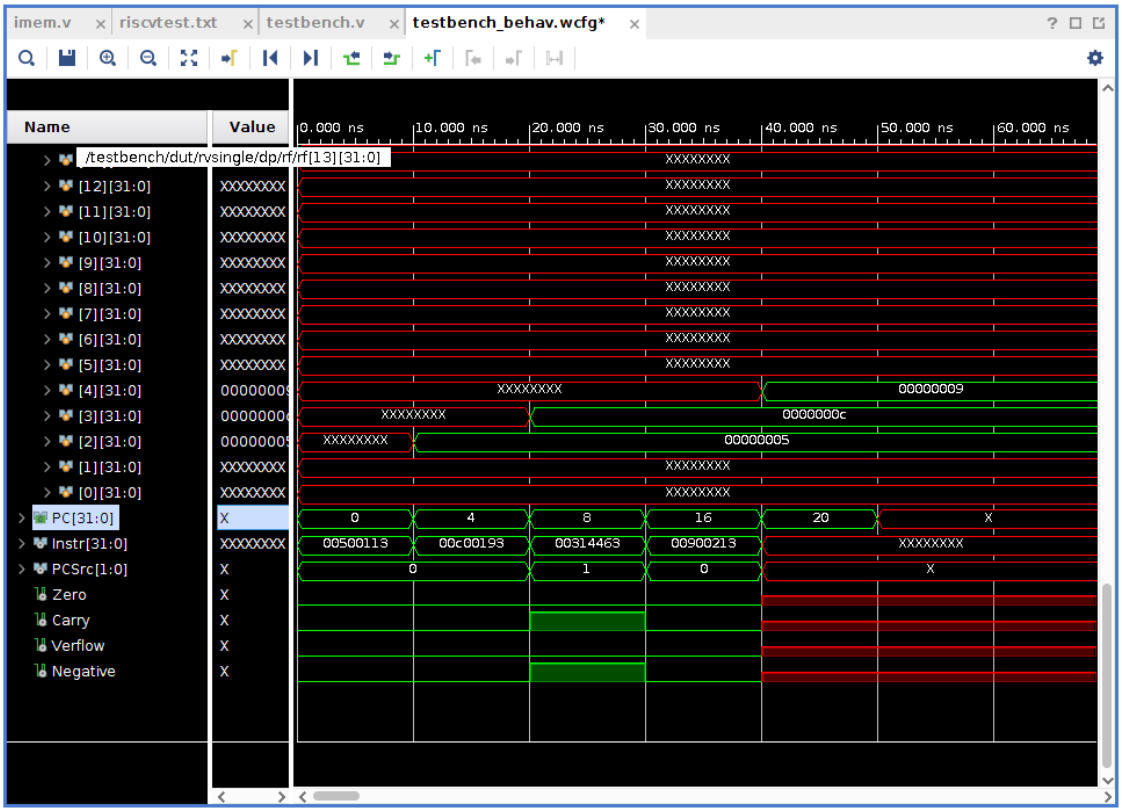

Figure 26: BLT - Instruction Test Case 1

Case 2: X2 = - 5, X3 = 0x0C

Expected Result: Taken
Expected PC: 0x16

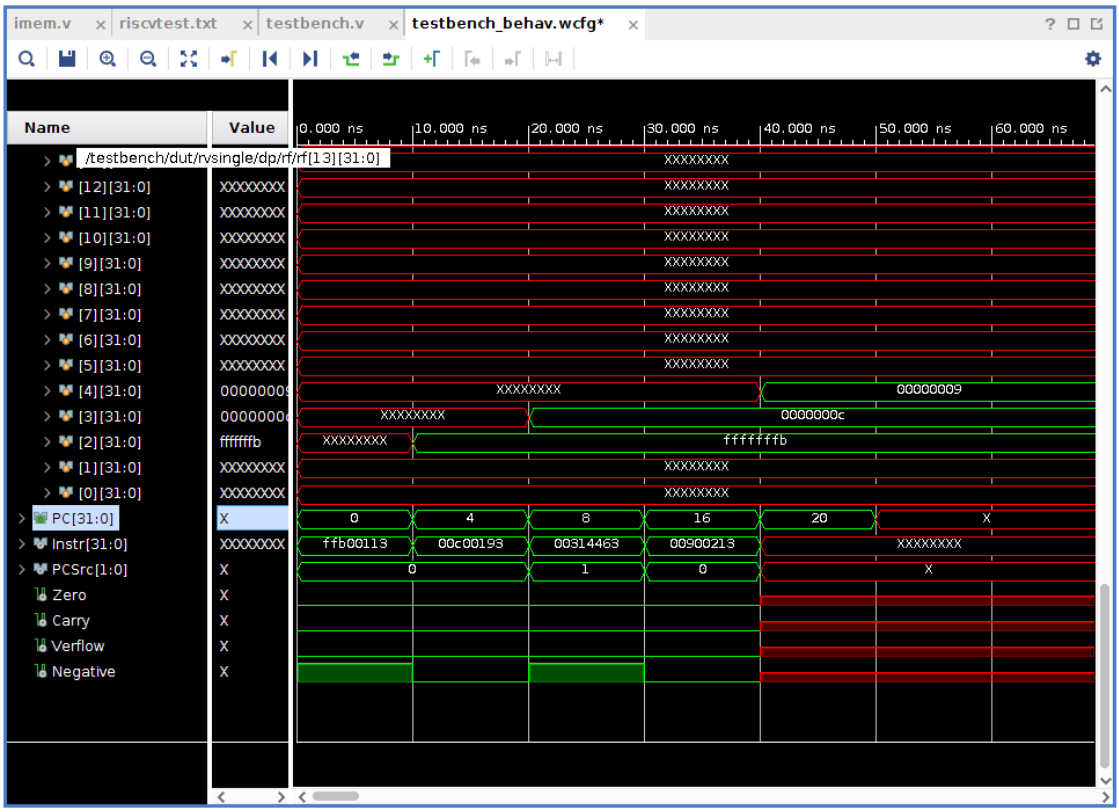

Figure 27: BLT - Instruction Test Case 2

2.2.4.4 BGE - Branch Greater Equal

Instruction	Operation
bge rs1, rs2, label	if (rs1 $\geq$ rs2) PC = BTA

bge x2, x3, 0x08

Machine Code: 0x00315463

Case 1: X2 = 5, X3 = 0x0C

Expected Result: Not Taken
Expected PC: 0x08

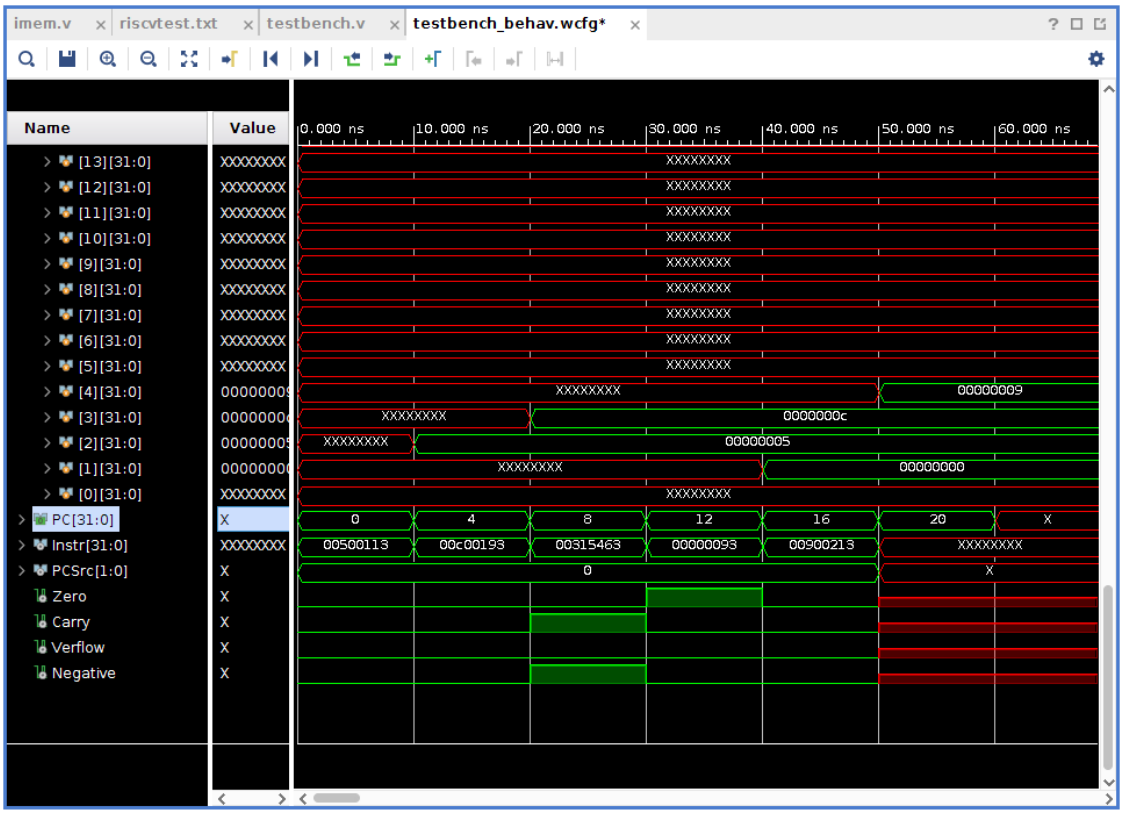

Figure 28: BGE - Instruction Test Case 1

Case 2: X2 = - 5, X3 = 0x0C

Expected Result: Not Taken
Expected PC: 0x08

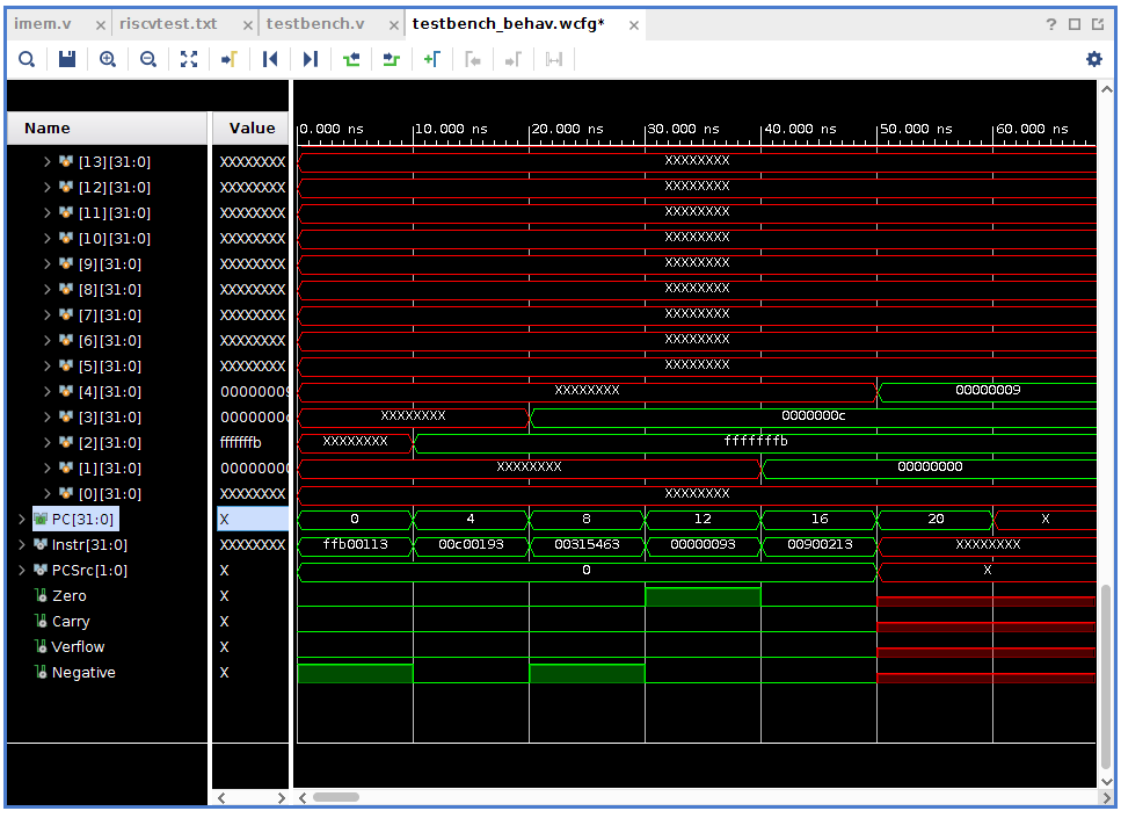

Figure 29: BGE - Instruction Test Case 2

2.2.4.5 BLTU - Branch Less Than Unsigned

Instruction	Operation
bltu rs1, rs2, label	if (rs1 < rs2) PC = BTA

bltu x2, x3, 0x08

Machine Code: 0x00316463

Case 1: X2 = 5, X3 = 0x0C

Expected Result: Taken
Expected PC: 0x16

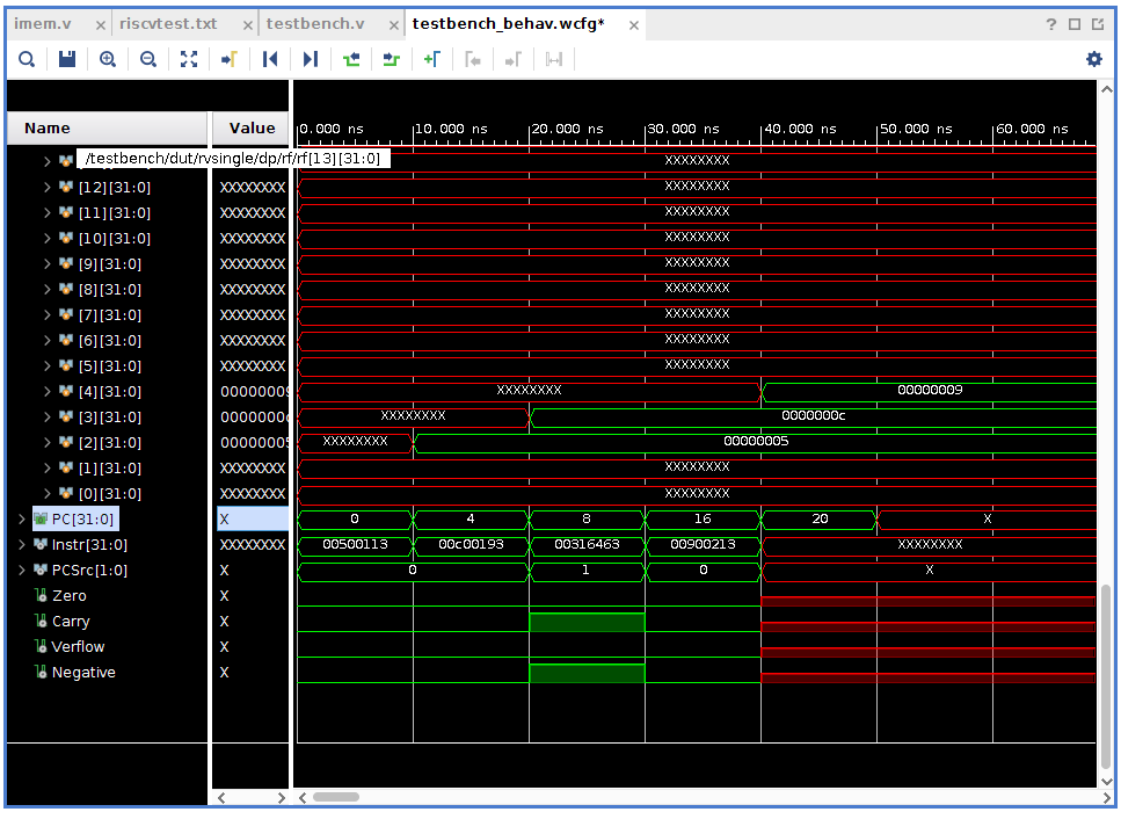

Figure 30: BLTU - Instruction Test Case 1

Case 2: X2 = - 5, X3 = 0x0C

Expected Result: Not Taken
Expected PC: 0x08

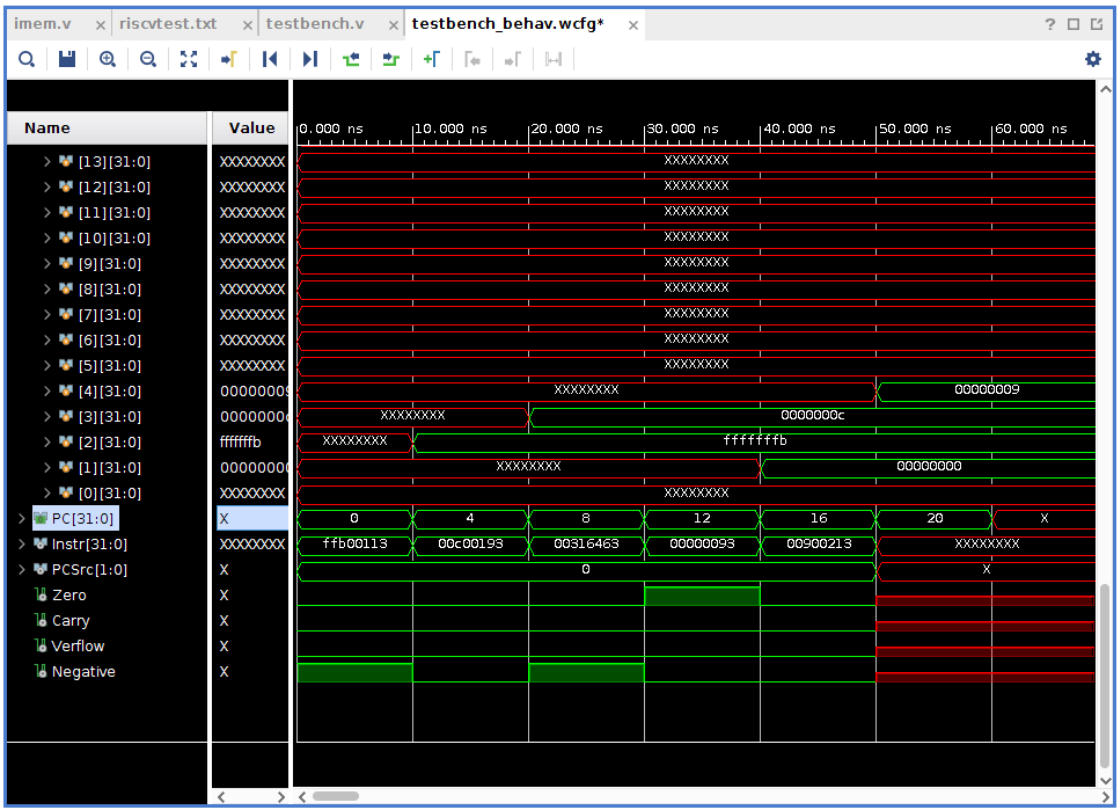

Figure 31: BLTU - Instruction Test Case 2

2.2.4.6 BGEU - Branch Greater Equal Unsigned

Instruction	Operation
bge rs1, rs2, label	if (rs1 $\geq$ rs2) PC = BTA

bgeu x2, x3, 0x08

Machine Code: 0x00317463

Case 1: X2 = 5, X3 = 0x0C

Expected Result: Not Taken
Expected PC: 0x08

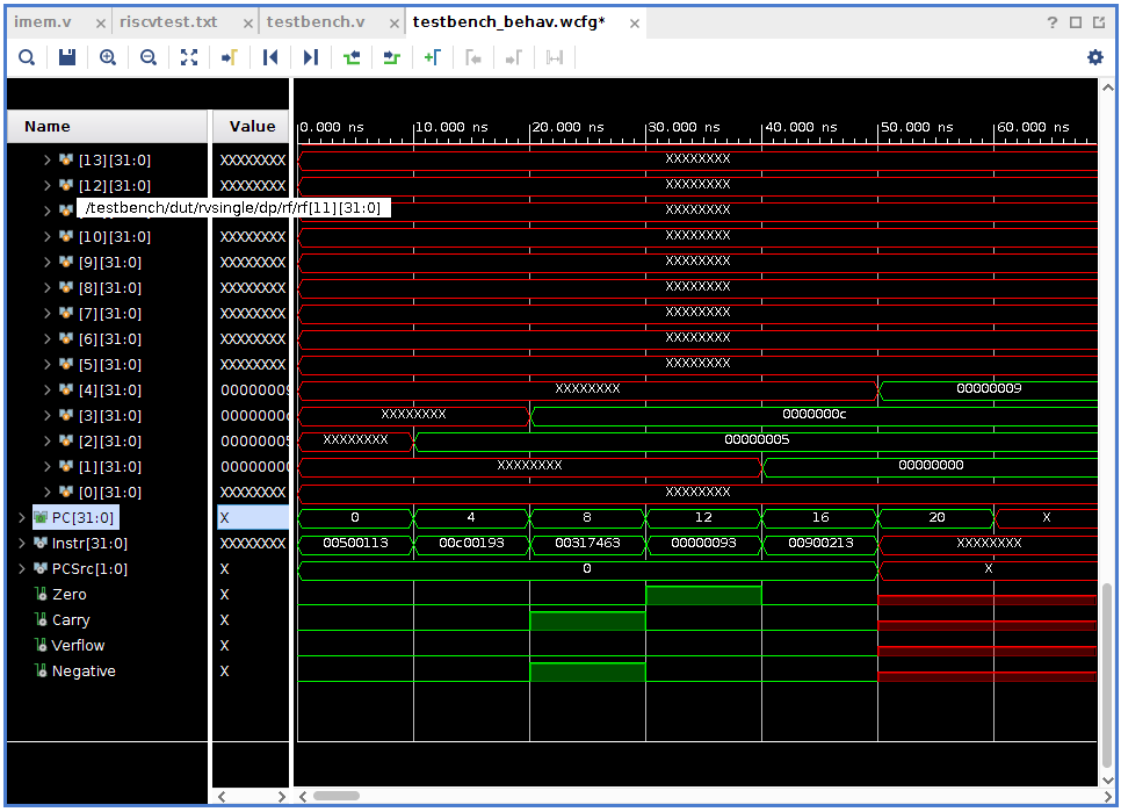

Figure 32: BGEU - Instruction Test Case 1

Case 2: X2 = - 5, X3 = 0x0C

Expected Result: Taken
Expected PC: 0x16

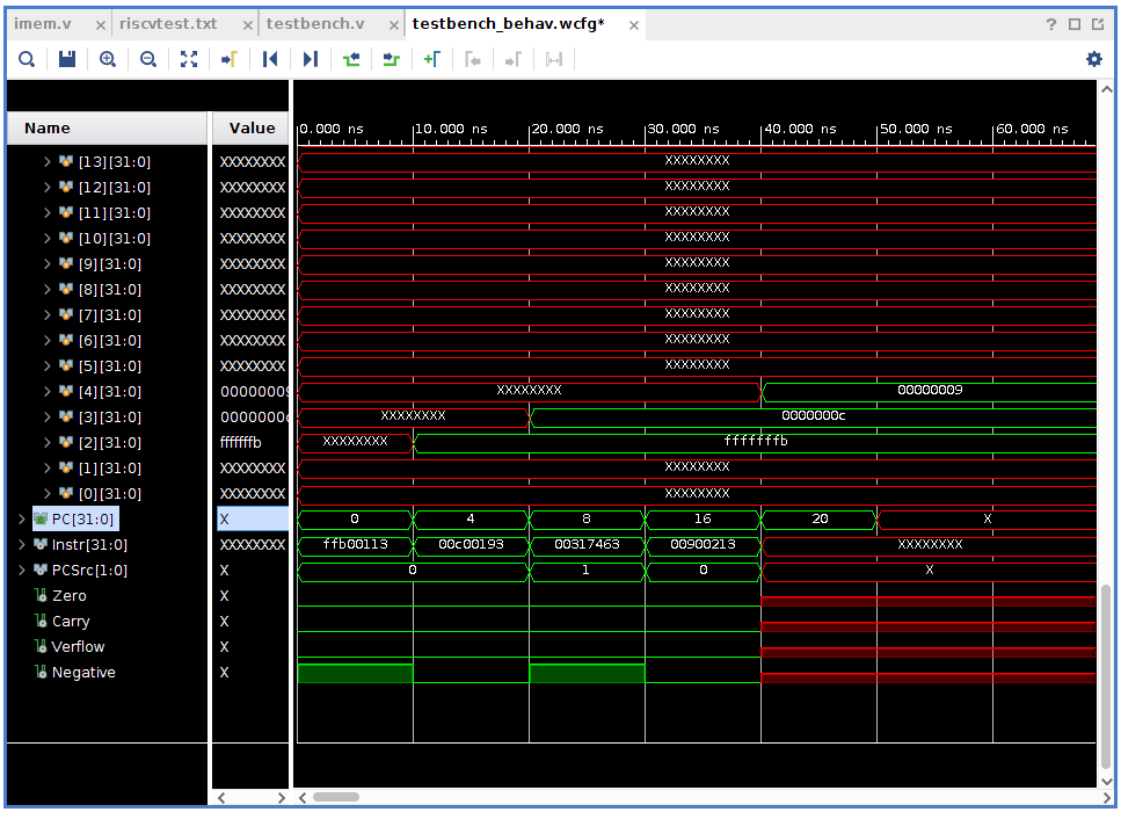

Figure 33: BGEU - Instruction Test Case 2

2.2.5 Type U

2.2.5.1 LUI - Load Upper Immediate

Instruction	Operation
lui rd, upimm	rd = upimm, 12'b0

This instruction adds 12 zeros to the 20 bit immediate, in this test the 20 bit immediate was 0xABCDE and the result was stored in the register x2.

LUI x2, 0xABCDE

Expected Data Address: 0x2
Expected Write Data: 0xABCDE000
Machine Code: 0xABCDE137

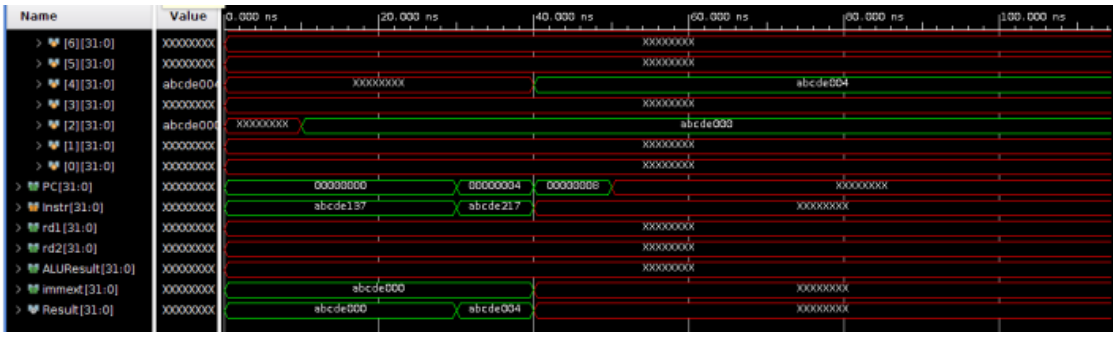

Figure 34: LUI - Instruction Test

2.2.5.2 AUIPC - Add Upper Immediate to PC

Instruction	Operation
auipc rd, upimm	rd = upimm, 12'b0 + PC

This instruction adds 12 zeros to the 20 bit immediate and adds it to the PC, in this test the 20 bit immediate was 0xABCDE and the result was stored in the register x2.

AUIPC x4, 0xABCDE

Expected Data Address: 0x4
Expected Write Data: 0xABCDE004
Machine Code: 0xABCDE217

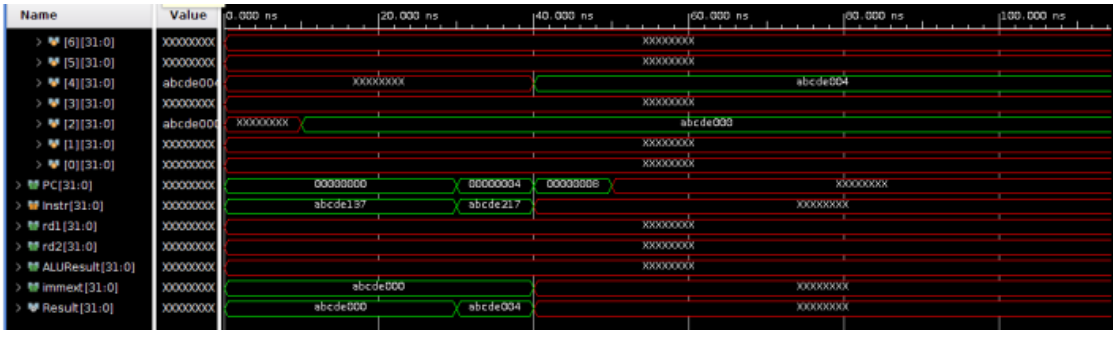

Figure 35: AUIPC - Instruction Test

3 Pipeline Processor

3.1 Introduction to Pipeline

Pipelining is a way to implement temporal parallelism and it is commonly used in modern processors as it significantly increases the processor performance at low cost. Adding pipelining to a processor increases the throughput (instructions executed per second). However, the more stages the pipeline has, the higher the latency (time needed for a given instruction to execute completely) will be. Adding more than 8 pipeline stages is counter-productive to the Risc-V processor when trying to increase performance.

To add pipelining to the single-cycle processor that was previously implemented, it was decided to add 5 stages, which will be Fetch, Decode, Execute, Memory Read/Write and Writeback (the same ones considered in single-cycle). The timing diagram for the implemented pipelined processor is represented in Figure 36.

The following sub chapters describe the way the group managed to add pipelining to the single-cycle processor.

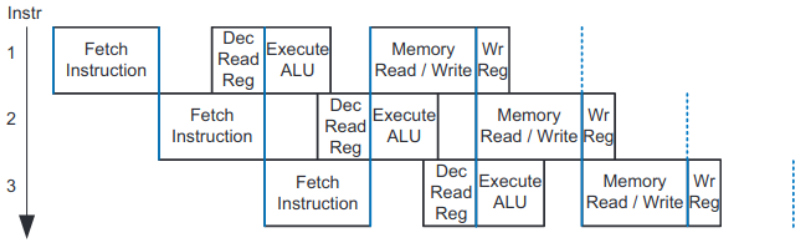

Figure 36: Pipeline Timing Diagram

3.2 Hazard unit

The hazard unit is a sector of a pipeline processor responsible for manage pipeline hazards. Those can occur when a data by one instruction that is still not fully saved is requested by another instruction, or when a branch (or jump) is being processed and the program flow changes (the jump is taken).

To solve these hazards there are 3 run time solutions. It can either data forward a value from a late stage of the pipeline back to an earlier stage in order to fulfill an instruction request without losing clock cycles, it can stall the newest instructions in order to allow the previous ones to calculate a value required in one of the new instructions or it can flush the newest instructions whenever there is a branch that jumps to an unpredicted state (this processor predicts all branches as not taken while

the jump condition is unknown).

According with the pipelined model of a riscv processor it will be required the following inputs and outputs in the hazard unit:

```
module hazardunit(
    clk,
    reset,

    StallF,
    StallD,
    FlushD,
    FlushE,
    ForwardAE,
    ForwardBE,

    Rs1D,
    Rs2D,
    RdE,
    Rs1E,
    Rs2E,
    PCSrcE,
    ResultSrcE,
    RdM,
    RegWriteM,
    RdW,
    RegWriteW
    );
```

Starting from the data forward solution (least time expensive solution) it can be applied to either the srcA or the srcB of the ALU and can be provided either from the result of the ALU (forward 1 cycle) or the result of the operation (forward 2 cycles).

The data forward of 1 cycle occurs when the destiny register from the instruction in the Memory access stage (Rs1E==RdM or Rs2E==RdM), while the data forward of 2 cycles occurs when the destiny register from the instruction in the Write back stage (Rs1E==RdW or Rs2E==RdW).

An aditional condition there was necessary to add is to prevent the data forward from happening when the destiny register is x0 because that address is locked at the const value of 0.

```

assign ForwardAE[1]=(((Rs1E==RdM) & RegWriteM) & ~(Rs1E==5'b00000));

assign ForwardAE[0]=(((Rs1E==RdW) & RegWriteW) & ~(Rs1E==5'b00000));
```

```
5
6 assign ForwardBE[1]=(((Rs2E==RdM) & RegWriteM) & ~(Rs2E==5'b00000));
7
8 assign ForwardBE[0]=(((Rs2E==RdW) & RegWriteW) & ~(Rs2E==5'b00000));
```

The stall solution is used when is not possible to use the data forward to solve a data hazard. This solution stalls early stage instructions in order to allow the old ones to process a value required by the new ones. This solution may be used in coordination with the data forward.

Stalls occur when a load instruction (those take 5 cycles to obtain the loaded value) is still in the Execution stage of the pipeline and the destiny value is requested by an instruction in the Decode stage of the pipeline.

Stalls occur when the destiny register of a load instruction in the Execution stage is the same as a source register of an instruction in the decoding stage.

The stall affects the fetch and decode registers by holding all their values.

```
1
2   assign lwStall=(ResultSrcE==3'b001) & ((Rs1D==RdE) | (Rs2D==RdE));
3
4   assign StallF=lwStall;
5
6   assign StallD=lwStall;
```

The last solution is the flush. That is applied when a jump or a branch is executed and it's function is to clean the instructions currently in the pipeline that aren't suppose to be executed when the jump occurs.

A flush occurs when the program counter is set to a given value instead of incrementing 4 addresses and when a stall occurs.

```
1
2   assign FlushD=PCSrcE;
3
4   assign FlushE=lwStall | PCSrcE;
```

3.3 Stage Registers

Stage registers are responsible for dividing the processor in stages and save the state of an instruction from stage to stage. They must be able to respond synchronously to the clock cycle and assure the coherence of the instruction order.

Stage registers must be able to be reseted (to a known state) and, in some cases, even flushed (reseted to zero) or stalled (keep the same value from cycle to cycle).

The first register is the Fetch register and it was already half implemented from the single cycle processor, we just had to integrate the StallF control.

The inputs, outputs and control variables for this register are listed bellow:

```

module flopr (
    clk,
    reset,
    StallF,
    d,
    q
);
```

PCF' ⇔ d PCF ⇔ q
The next step was to implement the resister accordingly to it's function.

In the image the q and d represent PCF and PCF' respectively (the default names were kept).

```

always @(posedge clk or posedge reset)
        if (StallF)
            q <= q;
        else if (reset)
            q <= 0;
        else
            q <= d;
```

The second register is the Decode register and it is responsible for dividing the Fetch and Decode stages. It was fully implemented by us.

The inputs, outputs and control variables for this register are listed bellow:

```

module pipelineregD(
clk,
reset,

StallD,
FlushD,

InstrF,
PCF,
```

```
PCPlus4F,

InstrD,
PCD,
PCPlus4D
);
```

The next step was to implement the resister accordingly to it's function.

```

always @(posedge clk)
        if (reset) begin
            InstrD <= 0;
            PCD <= 0;
            PCPlus4D <= 0;
        end else if (StallD) begin
            InstrD <= InstrD;
            PCD <= PCD;
            PCPlus4D <= PCPlus4D;
        end else if (FlushD) begin
            InstrD <= 0;
            PCD <= 0;
            PCPlus4D <= 0;
        end else begin
            InstrD <= InstrF;
            PCD <= PCF;
            PCPlus4D <= PCPlus4F;
        end
```

The third register is the Execute register and it is responsible for dividing the Decode and Execute stages. It was fully implemented by us.
Note that it was important to also pipeline the control unit. The funct3 was implemented out of what is represented in the pipeline diagram because in the single cycle processor we required that BUS to select the type of branch in the branch instructions.

The inputs, outputs and control variables for this register are listed bellow:

```

module pipelineregE(
clk,
reset,

FlushE,

PCPlus4D,
ImmExtD,
RdD,
Rs2D,
Rs1D,
```

```
PCD,
RD2D,
RD1D,

ALUSrcD,
ALUControlD,
BranchD,
JumpD,
MemWriteD,
ResultSrcD,
RegWriteD,
funct3D,

PCPlus4E,
ImmExtE,
RdE,
Rs2E,
Rs1E,
PCE,
RD2E,
RD1E,

ALUSrcE,
ALUControlE,
BranchE,
JumpE,
MemWriteE,
ResultSrcE,
RegWriteE,
funct3E
);
```

The next step was to implement the resister accordingly to it's function.

```

always @(posedge clk)
        if (reset) begin
            PCPlus4E <= 0;
            ImmExtE <= 0;
            RdE <= 0;
            Rs2E <= 0;
            Rs1E <= 0;
            PCE <= 0;
            RD2E <= 0;
            RD1E <= 0;

            ALUSrcE <= 0;
            ALUControlE <= 0;
            BranchE <= 0;
            JumpE <= 0;
            MemWriteE <= 0;
```

```
            ResultSrcE <= 0;
            RegWriteE <= 0;
            funct3E <= 0;
        end else if (FlushE) begin
            PCPlus4E <= 0;
            ImmExtE <= 0;
            RdE <= 0;
            Rs2E <= 0;
            Rs1E <= 0;
            PCE <= 0;
            RD2E <= 0;
            RD1E <= 0;

            ALUSrcE <= 0;
            ALUControlE <= 0;
            BranchE <= 0;
            JumpE <= 0;
            MemWriteE <= 0;
            ResultSrcE <= 0;
            RegWriteE <= 0;
            funct3E <= 0;
        end else begin
            PCPlus4E <= PCPlus4D;
            ImmExtE <= ImmExtD;
            RdE <= RdD;
            Rs2E <= Rs2D;
            Rs1E <= Rs1D;
            PCE <= PCD;
            RD2E <= RD2D;
            RD1E <= RD1D;

            ALUSrcE <= ALUSrcD;
            ALUControlE <= ALUControlD;
            BranchE <= BranchD;
            JumpE <= JumpD;
            MemWriteE <= MemWriteD;
            ResultSrcE <= ResultSrcD;
            RegWriteE <= RegWriteD;
            funct3E <= funct3D;
        end
```

The fourth register is the Memory (access) register and it is responsible for dividing the Execute and Memory stages. It was fully implemented by us.
In this register we also had to pipeline the control unit

The inputs, outputs and control variables for this register are listed bellow:

```

module pipelineregM(
clk,
reset,
```

```
5
6  ALUResultE,
7  WriteDataE,
8  RdE,
9  PCPlus4E,
10
11 RegWriteE,
12 ResultSrcE,
13 MemWriteE,
14
15 ALUResultM,
16 WriteDataM,
17 RdM,
18 PCPlus4M,
19
20 RegWriteM,
21 ResultSrcM,
22 MemWriteM
23 );
```

The next step was to implement the register accordingly to it's function.

```
1
2  always @(posedge clk) begin
3          if (reset) begin
4              ALUResultM <= 0;
5              WriteDataM <= 0;
6              RdM <= 0;
7              PCPlus4M <= 0;
8
9              RegWriteM <= 0;
10             ResultSrcM <= 0;
11             MemWriteM <= 0;
12         end else begin
13             ALUResultM <= ALUResultE;
14             WriteDataM <= WriteDataE;
15             RdM <= RdE;
16             PCPlus4M <= PCPlus4E;
17
18             RegWriteM <= RegWriteE;
19             ResultSrcM <= ResultSrcE;
20             MemWriteM <= MemWriteE;
21         end
22 end
```

The last register is the Write (back) register and it is responsible for dividing the Memory and Write stages. It was fully implemented by us.
In this register we had to pipeline the control unit.

The inputs, outputs and control variables for this register are listed bellow:

```
1
2  module pipelineregW(
3  clk,
4  reset,
5
6  ALUResultM,
7  ReadDataM,
8  RdM,
9  PCPlus4M,
10
11 RegWriteM,
12 ResultSrcM,
13
14 ALUResultW,
15 ReadDataW,
16 RdW,
17 PCPlus4W,
18
19 RegWriteW,
20 ResultSrcW
21 );
```

The next step was to implement the resister accordingly to it's function.

```
1
2  always @(posedge clk) begin
3          if (reset) begin
4              ALUResultW <= 0;
5              ReadDataW <= 0;
6              RdW <= 0;
7              PCPlus4W <= 0;
8              RegWriteW <= 0;
9              ResultSrcW <= 0;
10         end else begin
11             ALUResultW <= ALUResultM;
12             ReadDataW <= ReadDataM;
13             RdW <= RdM;
14             PCPlus4W <= PCPlus4M;
15             RegWriteW <= RegWriteM;
16             ResultSrcW <= ResultSrcM;
17         end
18 end
```

3.4 MUXs

In order to implement the dataforward it's necessary to add MUXs to the sources of the ALU in order to select from where the values are loaded. The possible source options are:

- RD1E/RD2E
- ALUResultM
- ResultW

Taking this in mind the MUXs were created by using the variables ForwardAE and ForwardBE as selectors.

```
1
2     mux3 #(32) AEmux(
3         RD1E,
4         Result,
5         ALUResultM,
6         ForwardAE,
7         SrcA
8     );
```

```
1
2     mux3 #(32) BEmux(
3         RD2E,
4         Result,
5         ALUResultM,
6         ForwardBE,
7         WriteData
8     );
```

Note that SrcB has a control MUX between the hazard MUX and the ALU. For that reason the output of the muxBE is WriteDataE instead of SrcB directly.

3.5 Tests

In order to validate the pipeline implementation it was necessary to run some testbenches with hazards. That way it would be possible to detect if the hazard unit is working as intended and if the instructions are shifting through the pipeline stages without collisions or unintentional delays (stall is an intentional and required delay).

The first test will validate the dataforward solution for an data hazard. In order to test, we'll be using the example from the recomended bibliography.

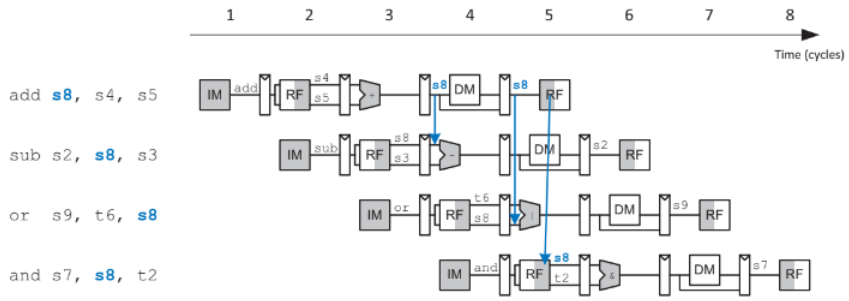

Figure 7.54 **Abstract pipeline diagram illustrating forwarding**

Figure 37: Dataforward example

As can be analysed from the example the instructions "sub", "or" and "and" require the value from a register (x8) that is being actualized in the first instruction (add), however that value requires the full 5 cycles in order to be written back into x8 and be able to be used. this can be solved with dataforward.
In the first situation the instruction "sub" is in the Execution stage and the x8 value is already available in the Memory stage, so the hazard unit enables the mux that allows the ALUResultE to be loaded directly into the SrcA avoiding any hazard.
In the second example the x8 value is in the Write back stage while the or instruction is requesting it in the Execution stage. As with the previous example the hazard unit will load the result directy into the SrcA in order to prevent the hazard.
In the last example the x8 value is still in the Write back stage but is being requested in the regfile, so internally the regfile must be able to dataforward internally the x8 register in order to prevent any time losses. That can be achieved with the following code:

```
1
2 01200113
3 01300193
4 01400213
5 01500293
6 01600313
7 01700393
8 01800413
9 01900493
```

this attribution saves the clock cycle of saving the value in the register and then load it again.

In order to test the implementation first we will set the registers to a known value to analyse the testbench easily. The registers from x2 to x9 will all be set from 0x12 to 0x19 in increments of 1.

Converting to machine code:

```
01200113
01300193
01400213
01500293
01600313
01700393
01800413
01900493
```

Then we convert the example instructions:

```
00520433
40340133
008364B3
002473B3
```

By running the simulation we acquire the following result:

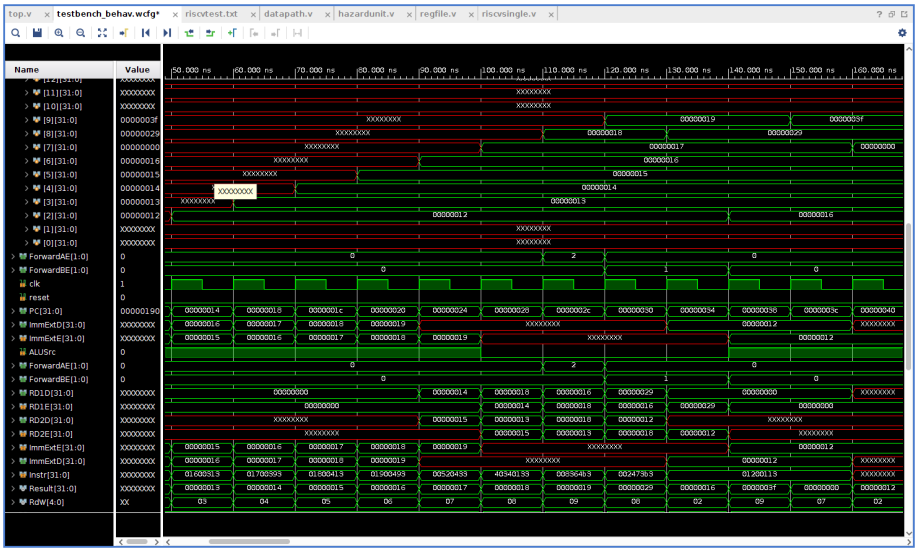

Figure 38: Pipeline processor

As we can observe from the simulation the value calculated in the first instruction (0x14 + 0x15 = 0x29) is data forwarded from the Memory stage back to the Execution stage in order to fulfill the SrcA in the 2nd instruction.

In order to validate the data hazard solving using stalls there was executed a second test. this test was also taken from the recomended bibliography.

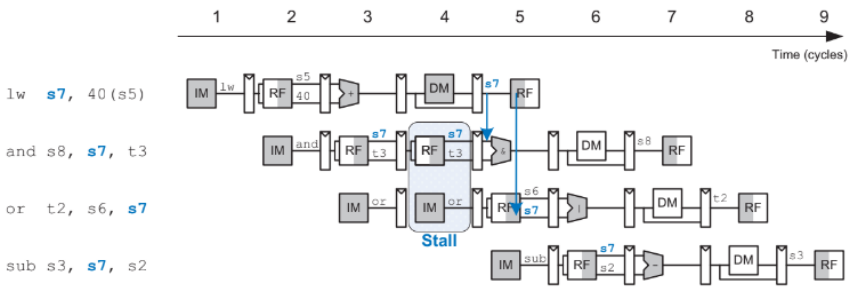

Figure 7.57 **Abstract pipeline diagram illustrating stall to solve hazards**

Figure 39: Dataforward example

The hazards are visible in the "and" and "or" instructions. In order to take the value from x7 and use it in the next instructions we require an extra cycle because the load instruction only disponibilizes the value in the Memory stage when it takes the value from the dmem. Since the datapath shifting occurs to all the stages simultaneously it is necessary to implement a control for this hazard that prevents certain stages from advancing while the elder instructions are still finalizing being executed. When finally the load instruction reaches the Write back stage, then the value can be dataforwarded as seen in the previous test.

For simulation purposes we converted the instructions into machine code

```
0402A383
00338433
00736133
402381B3
```

After the simulation was ran we obtained the following chart.

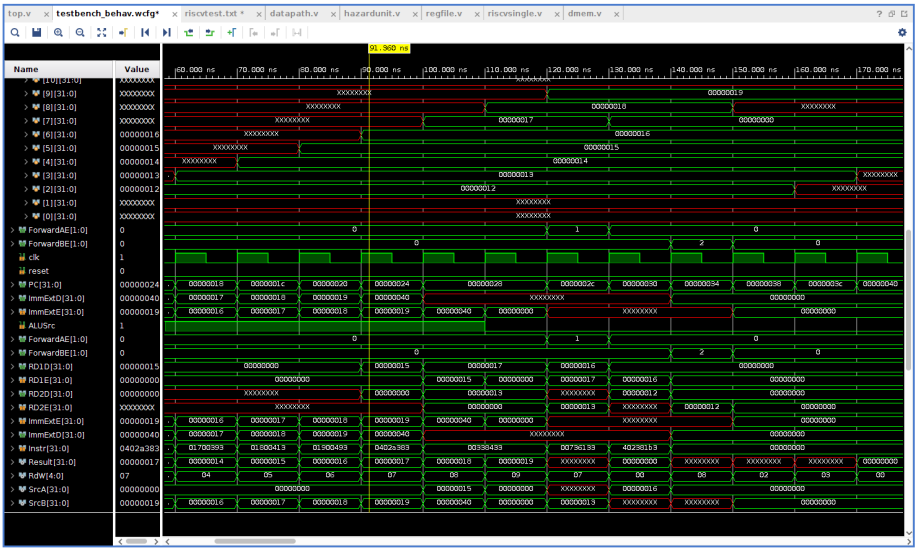

Figure 40: Stall example

As we can observe the instructions in the Decode and Execution stages stall as well as the PC stops, for one cycle, during the 7th cycle (PC=0x28). This allows the result to aquire the value of x8 (in this example: 0x29) and dataforward it to the SrcB.

The third test will validate the flush solution for a control hazard. The test had as reference an example from the recommended bibliography.

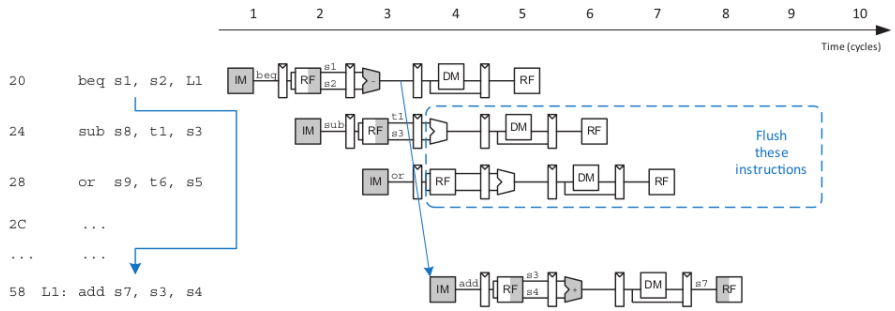

Figure 7.59 Abstract pipeline diagram illustrating flushing when a branch is taken

Figure 41: Flush example

The *beq* instruction represent a control hazard because the branch decision has not been taken by the time the next instruction is retrieved.

Predicting whether the branch will be taken and then executing instructions based on that prediction is an alternative to blocking the pipeline. If the forecast was incorrect, the processor can discard the instructions once the branch decision is ready. The two instructions after the branch must be flushed by clearing the pipeline registers for those instructions if the branch should have been taken. In the test made there was just an instruction made after the *beq* and it should be flushed because the branch will be taken.

For the test the registers x3 and x2 were set to 12 and the machine code for these instructions was the following:

```
00C00113
00C00193
```

Then based on the example,the instructions made were:
beq x3,x2,8
addi x2, x0, 5
addi x9, x0, 14

That in machine code are respectively:

```
00218463
00500113
00E00493
```

By running the simulation we acquire the following result:

Figure 42: Hazard Unit - Flush Test

If the branch was taken it should execute the instruction that sets register x9 to 14. And since x3 and x2 are equal the branch was taken, the instruction that was after the *beq* was flushed and the register x9 was set to 14, proving that the Hazard Unit and the Pipeline are working well.

3.6 Running the processor in the Zybo

After the pipeline processor being implemented we created a constraint file in order to set some constraints in the board. The first constraints were to set the proprieties for all the wires and declare the pin voltage for every wire signal.

```
set_property IOSTANDARD LVCMOS33 [get_ports clk]
set_property IOSTANDARD LVCMOS33 [get_ports MemWrite]
set_property IOSTANDARD LVCMOS33 [get_ports reset]
set_property IOSTANDARD LVCMOS33 [get_ports success]
```

A wire called "success" was added because, in order to check if the processor is running the program successfully we could output the DataAdr and WriteData BUSs, but it would require mapping a lot of ports and it wouldn't as perceptible to see if succeeded. Because of that we implemented a condition for the success pin that sets it true when the 100th position (25*4bytes) of the RAM is 25, that will prove that when we run the given testbench and the success pin is set to true, the processor is running correctly in the zybo.

```
always @(*)
        if (RAM[25]==25)
            successful<=1;
        else if (reset)
            successful<=0;
```

After set the proprieties for every wire it was necessary to assign the respective ports.

```
set_property PACKAGE_PIN K17 [get_ports clk]
set_property PACKAGE_PIN Y16 [get_ports reset]
set_property PACKAGE_PIN D18 [get_ports MemWrite]
set_property PACKAGE_PIN G14 [get_ports success]
```

The clock signal was mapped in the Zybo's clk port while the reset signal was mapped in the first push-button and the last two signals were mapped in the first two wires.

After the hardware was mapped it was necessary to create and set a clock for the processor. With that we created a clock with 10us and 50% duty cycle which has a higher period than the critical path time plus the contamination time.

```
create_clock -period 10000.000 -name clk -waveform {0.000 5000.000}
    [get_ports clk]
```

With that the processor was ready to run in the board.

First we tested with the code that was given with the testbench.

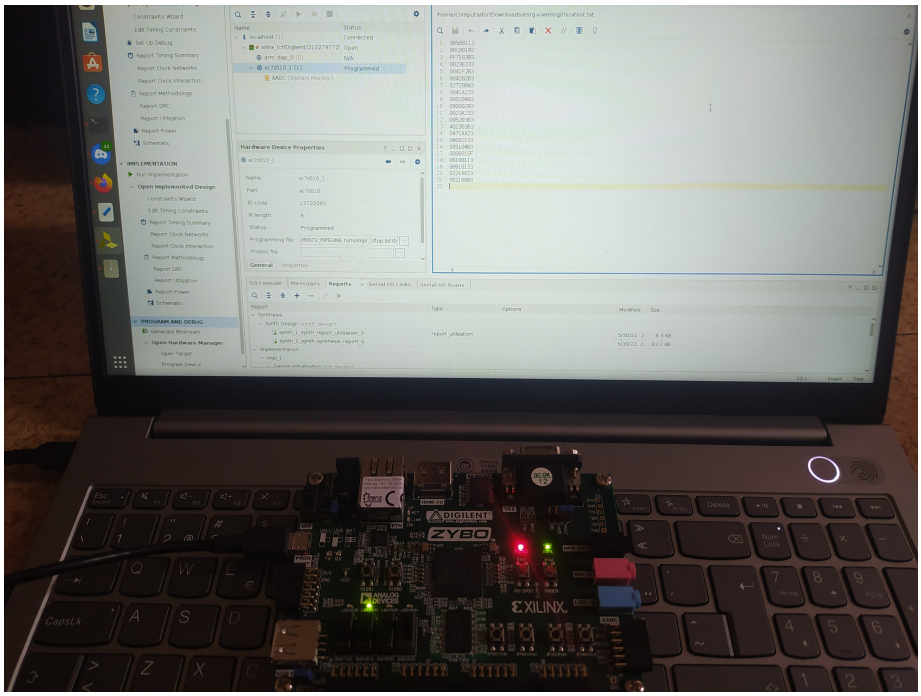

Figure 43: Processor running in the Zybo 1

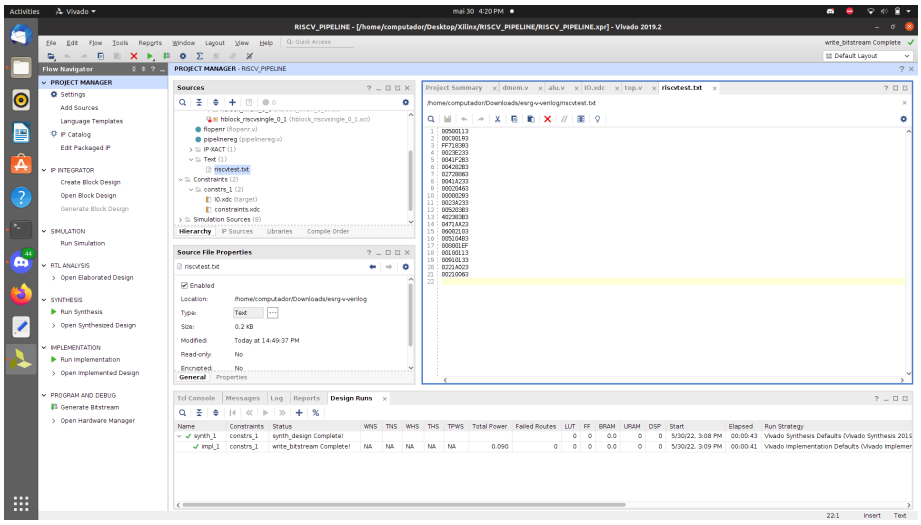

Figure 44: Instructions used in the testbench 1

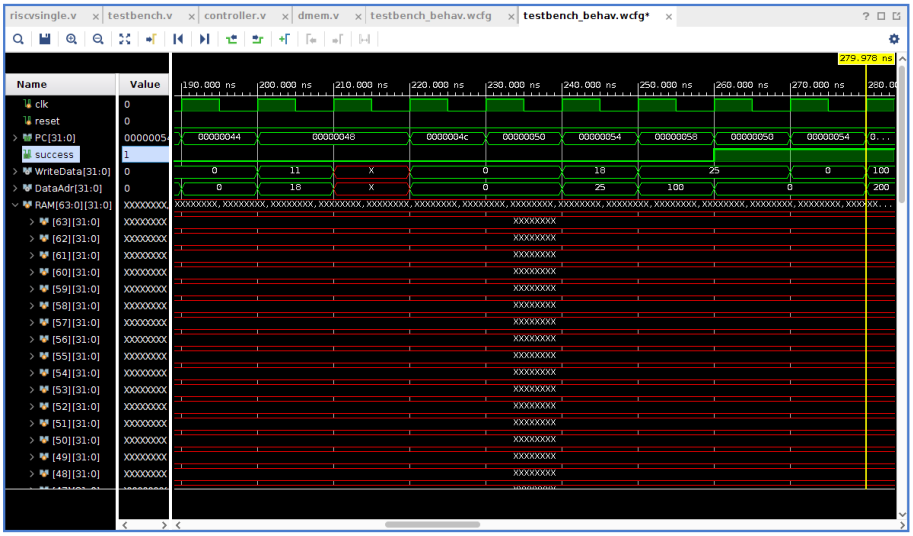

Figure 45: Simulation 1

As it can be seen in the images, the processor is running the code correctly turning on the sucess led at the end.

Next we cleared the instruction responsible for setting the 100th byte of the RAM as 25.

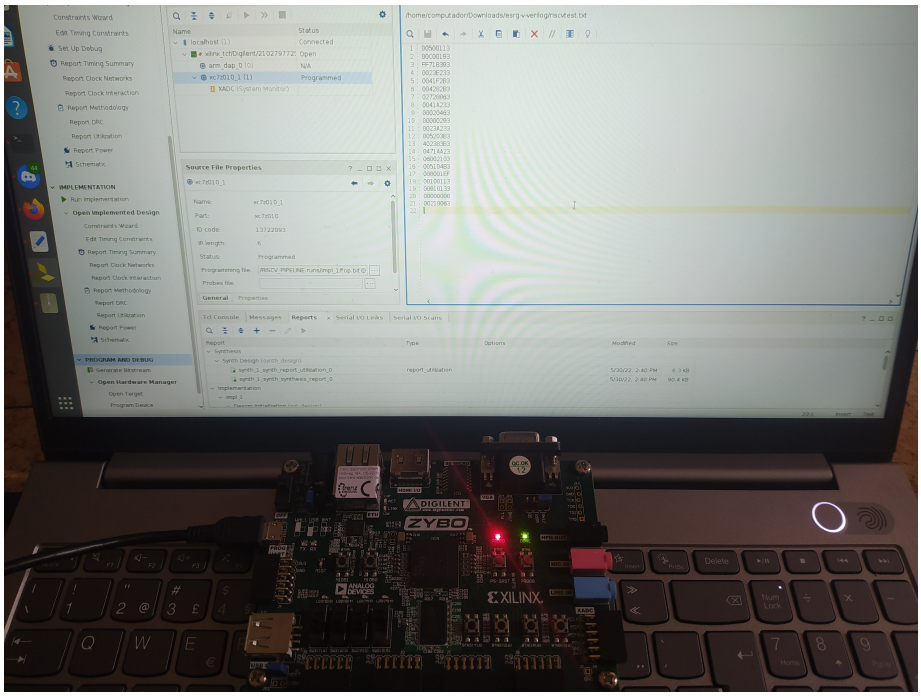

Figure 46: Processor running in the Zybo 2

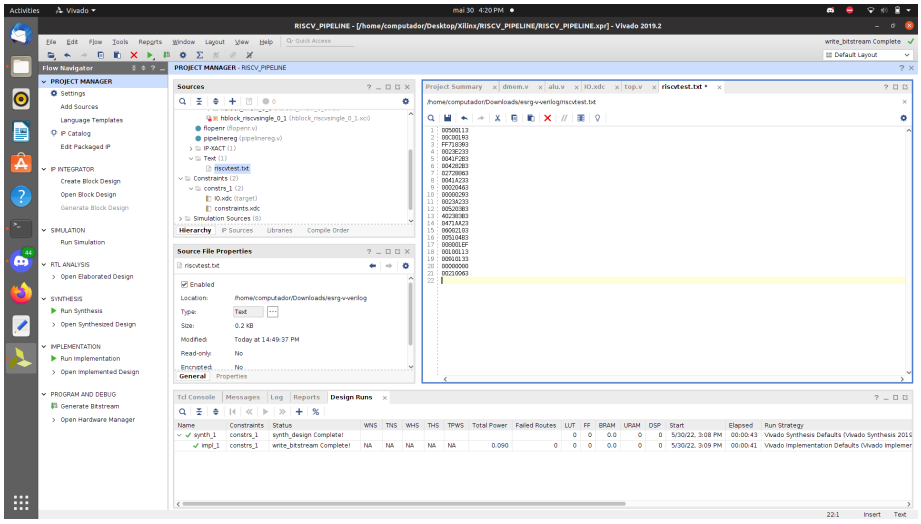

Figure 47: Instructions used in the testbench 2

As it can be seen in the images, the processor did not write the value in the correct position of the RAM, so it wasn't successful.

This proves that the processor can run the instructions as successfully as it can be seen in the simulations.

4 Devices

In order to implement devices a new IP was created. The said IP has the function to interface with peripherals. In this case we will be interfacing with the GPIO peripherals (pushbuttons, switches and LEDs).

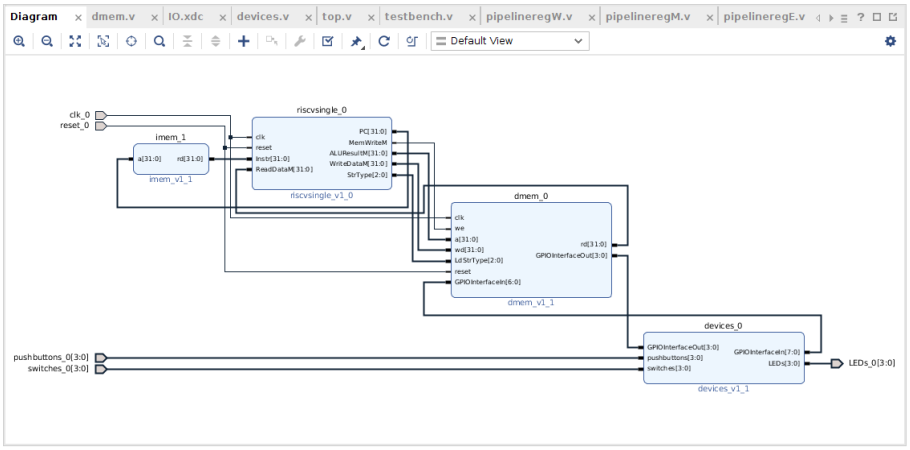

Figure 48: Hardware block

In order to interface with the dmem there are two BUSes, one for the input peripherals (switches and pushbuttons) and one for output peripherals (LEDs), the first one supports 3 pushbuttons (the two buttons mapped in the MultiplexedI/O weren't used as well as the BTN3 that was used for reset) and 4 switches, while the second interface supports 4 LEDs (the fifth is MultiplexedI/O, so we didn't use).

Features

- 650Mhz dual-core Cortex-A9 processor
- DDR3 memory controller with 8 DMA channels
- High-bandwidth peripheral controllers: 1G Ethernet, USB 2.0, SDIO
- Low-bandwidth peripheral controller: SPI, UART, CAN, I2C
- Reprogrammable logic equivalent to Artix-7 FPGA
  - 4,400 logic slices, each with four 6-input LUTs and 8 flip-flops
  - 240 KB of fast block RAM
  - Two clock management tiles, each with a phase-locked loop (PLL) and mixed-mode clock manager (MMCM)
  - 80 DSP slices
  - Internal clock speeds exceeding 450MHz
  - On-chip analog-to-digital converter (XADC)
- ZYNQ XC7Z010-1CLG400C
- 512MB x32 DDR3 w/ 1050Mbps bandwidth
- Dual-role (Source/Sink) HDMI port
- 16-bits per pixel VGA source port
- Trimode (1Gbit/100Mbit/10Mbit) Ethernet PHY
- MicroSD slot (supports Linux file system)
- OTG USB 2.0 PHY (supports host and device)
- External EEPROM (programmed with 48-bit globally unique EUI-48/64™ compatible identifier)
- Audio codec with headphone out, microphone and line in jacks
- 128Mb Serial Flash w/ QSPI interface
- On-board JTAG programming and UART to USB converter
- GPIO: 6 pushbuttons, 4 slide switches, 5 LEDs
- Six Pmod ports (1 processor-dedicated, 1 dual analog/digital, 3 high-speed differential, 1 logic-dedicated)

Figure 49: Zybo features

On the other hand we also have the external ports for the peripherals (pushbuttons, switches and LEDs).

The first step in order to connect the device interface is to reserve a memory address. For that we reserved the address 0x0. The first 4 bits will be connected to the LEDs, the 4 next bits connected to the switches, the 3 following bits connected to the push buttons and lastly 21 bits reserved for future use.

```
    if(a[31:2]==0) begin //GPIO Pheripheral
        //[31:11]-reserved (future use)
        //[10:4]-input peripherals (pushbuttons+switches) read only
        //[3:0]-output peripherals (LEDs) read+write
        RAM[a[31:2]][3:0] <= wd[3:0];
    end
```

As it can be seen by the implementation the input and output pins are allowed to be read, but only the output pins can be written. For that reason, when we try to write to this address only the 4 less significative bits are overwritten.
On the other hand, when we read we can only read the non-reserved pins.

```
assign rd = a[31:2] ? RAM[a[31:2]] : {21'b000000000000000000000, RAM
    [a[31:2]][10:0]};
```

Lastly the device module is responsible to connect to the peripherals.

```
    assign GPIOInterfaceIn={pushbuttons,switches};
    assign LEDs=GPIOInterfaceOut;
```

In order to test the implementation we assigned the LED bus with the number 0x5 and generated the bitstream to test on the board.

```
always @(*)
        RAM[0][3:0]=5;
```

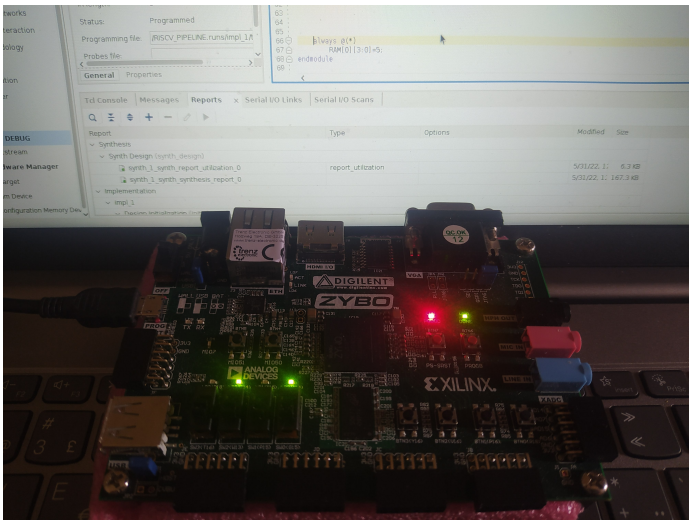

Figure 50: Device connectivity test

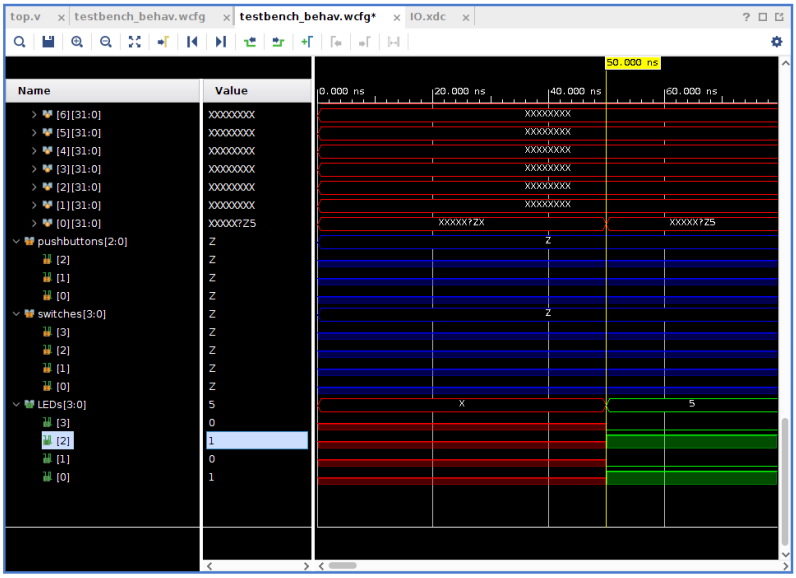

Figure 51: Device connectivity test

As expected the LEDs 0 and 2 turned on and the LEDs 1 and 3 were kept off. That confirms that the peripherals are physically mapped in the memory through the device module.

This feature can be used by programs to communicate with the peripherals and interact with external stimulus.

5 Conclusion

This project represented a landmark in our knowledge about processors. The pipeline represented an interesting challenge, trying to conciliate all the instructions and organizing into stages granting a better throughput.

The integration of peripherals in the project allowed the processor to connect with the external world opening the possibility to integrate in a greater system.

With this project the group faced the opportunity to apply and analyse the flow of programs in an hardware system created from basic components into a processing unit.

Bibliography

[1] Sarah L Harris and David Money Harris. *Digital Design and Computer Architecture*. RISC-V Edition. Katey Birtcher, 2022. ISBN: 9780128200643.

[2] David A. Patterson and John L. Hennessy. *Computer Organization and Design*. RISC-V Edition. Morgan Imprint Publishers, 2018. ISBN: 9780128122754.

[3] David A. Patterson and John L. Hennessy. *Computer Architecture*. 6th edition. Morgan Imprint Publishers, 2019. ISBN: 9780128119051.