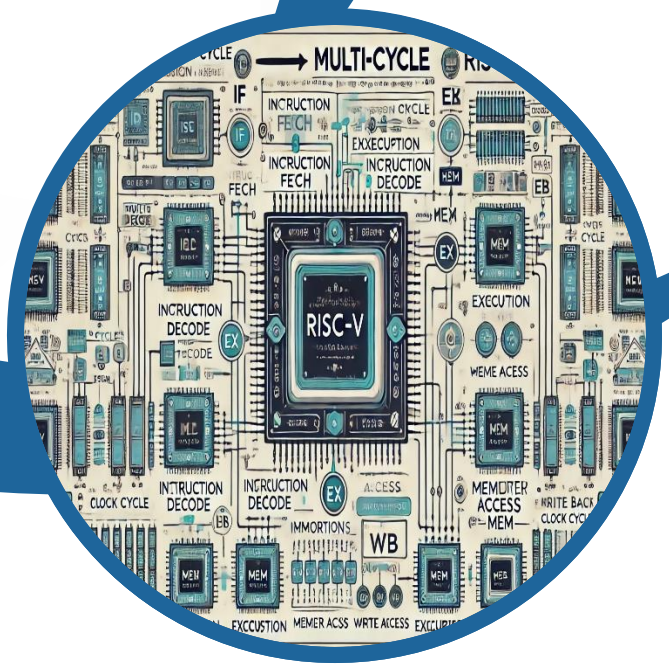
Computer Assignment 2

RISC-V Processor

Multi Cycle Approach

December 12, 2024



Bardia Amirian | 810101605
Amirreza Nadi Chaghadari | 810101538

Digital Systems 2

Dr. Safari

Table of Contents

Table of Figures

Table of Tables

Problem Description

We are to design a simplified version of the RISC-V processor, using multi-cycle approach. It means that different instructions will take different numbers of clock cycles to execute, but the clock itself can become faster. Here is the list of instructions we are expected to implement:

R-Type[1]: add, sub, and, or, slt

I-Type[2]: lw, addi, xori, ori, slti, jalr

S-Type[3]: sw

J-Type[4]: jal

B-Type[5]: beq, bne

U-Type[6]: lui

Each of these instructions have their own format, thus requiring the datapath to include appropriate paths for the data to flow accordingly.

Once we designed the processor, we are to test it by running a code on it which searches for the minimum of an array with 10 elements.

[1] Register Type
[2] Immediate Type
[3] Store Type
[4] Jump Type
[5] Branch Type
[6] Upper Type

Instruction Formats

R-Type Instructions

Instruction format

Luckily, all R-Type instructions we are to implement are well-defined and are almost the same. The only thing that differs between them is what the ALU[7] does with its inputs. the overall form of these instructions looks like this:

Table 1. T-Type Instruction Format

31 25	24 20	19 15	14 12	11 7	6 0
f_7	r_{s2}	r_{s1}	f_3	r_d	OPC

Instruction Execution Table

Table 2. R-Type Instruction Execution Table

add	$r_d = r_{s1} + r_{s2}$
sub	$r_d = r_{s1} - r_{s2}$
and	$r_d = r_{s1}$ & r_{s2} (bitwise)
or	$r_d = r_{s1} \mid r_{s2}$ (bitwise)
slt	$r_d = r_{s1} < r_{s2}\ ?\ 1 : 0$

[7] Arithmetic-Logic Unit (ALU): The part that handles arithmetic and logical calculations.

I-Type Instructions

Instruction format

Immediate type instructions of this computer assignments have 3 types:

1. The arithmetic type which includes addi, ori, xori, and slti
2. Loading type which includes lw
3. Jumping type which includes jalr

Table 3. I-Type Instruction Format

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
$Imm^{8}[11:0]$	r_{s1}	f_3	r_d	OPC

Instruction Execution Table

Table 4. I-Type Instruction Execution Table

addi	$r_d = r_{s1} + Imm$ (Sign-Extended)
ori	$r_d = r_{s1} \mid Imm$ (Sign-Extended)
xori	$r_d = r_{s1} \oplus Imm$ (Sign-Extended)
slti	$r_d = r_{s1} < Imm \; ? \; 1 : 0$ (Sign-Extended)
lw	$r_d = Mem^{9}[r_{s1} + Imm]$ (Signed-Extended)
jalr	$r_d = pc + 4$ $pc = r_s + Imm$ (Signed-Extended)

[8] Immediate
[9] Memory

S-Type Instructions

Instruction format

This assignment includes only one S-Type instruction, sw.

Table 5. S-Type Instruction Format

31 25	24 20	19 15	14 12	11 7	6 0
Imm[11:5]	r_{s2}	r_{s1}	f_3	Imm[4:0]	OPC

Instruction Execution Table

Table 6. S-Type Instruction Execution Table

sw	$Mem[r_{s1} + Imm] = r_{s2}$ (Sign-Extended)

J-Type Instructions

Instruction format

This assignment includes only one J-Type instruction: jal

Table 7. J-Type Instruction Format

31 25	24 20	19 15	14 12	11 7	6 0
Imm[20, 10:1, 11, 19:12]				r_d	OPC

The first thing to be noticed here, is how the immediate part contains bits 20:1, and not 19:0. This is because the value must be an integer coefficient of 4, meaning that its least significant 2 bits must be 0. Due to the weird choice made by the developers of RISC-V processor, the first one is ignored, but the second one must be included to avoid address exception! Therefore, the immediate could be interpreted like this:

$$Imm = \{Offset[20:2], 1'b0\}$$

There is also the weird format of giving the immediate, but it was another one of RISC-V processor's developers' choices we need to follow.

There is one more thing to be discussed here. If we do not specify $r_{d,}$ the default value will be ra. But, this is not the concern of this assignment, since we are not implementing an assembler.

Instruction Execution Table

Table 8. J-Type Instruction Execution Table

jal	$r_d = pc + 4$ $pc = pc + Imm$ (Signed-Extended)

B-Type Instructions

Instruction format

There are 2 branch instructions we are expected to implement: bne and beq. They both have this structure

Table 9. B-Type Instruction Format

31 25	24 20	19 15	14 12	11 7	6 0
Imm[12, 10:5]	r_{s2}	r_{s1}	f_3	Imm[4:1, 11]	OPC

What is to be noticed here (again!), is the weird format of the immediate. It is not saved consecutively, and does not end with 0. It has a formula like this:

$$Imm = \{Offset[12:2], 1'b0\}$$

Instruction Execution Table

Table 10. B-Type Instruction Execution Table

beq	$pc = r_{s1} == r_{s2}\ ?\ pc + Imm : pc + 4$
bne	$pc = r_{s1} \sim = r_{s2}\ ?\ pc + Imm : pc + 4$

U-Type Instructions

Instruction format

This assignment includes only one U-Type instruction: lui

Table 11. U-Type Instruction format

31 25	24 20	19 15	14 12	11 7	6 0
Imm[19:0]				r_d	OPC

Instruction Execution Table

Table 12. U-Type Instruction Execution Table

lui	$r_d = \{Imm, 12'H000\}$ (Signed-Extended)

With these tables at hand, we can design the datapath.

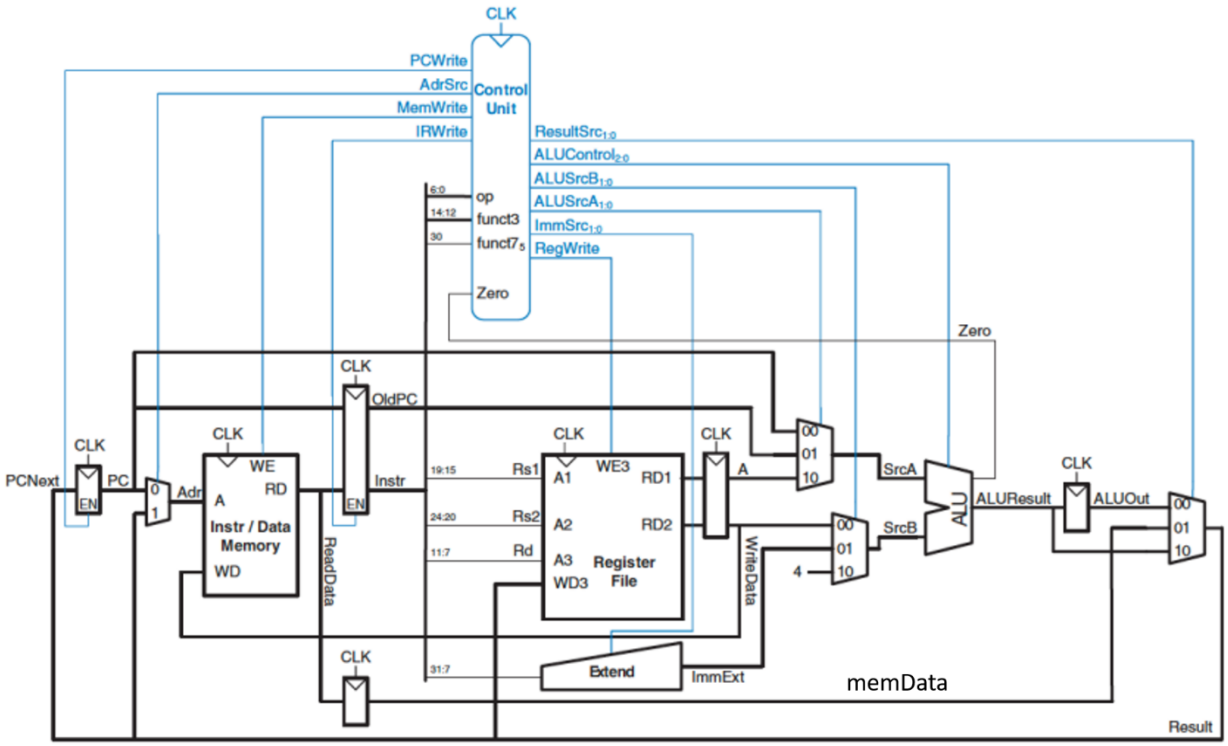
Datapath Design

Abstract Block Diagram

The idea here is to reuse the same hardware to handle different things in the cycle of executing each instruction. For example, when "LW" is being executed, the ALU can be used to calculate the next value of program counter, aka. PC, and calculate the address where the data is. In order to achieve this, we need to save anything that may be of use, before altering it.

Here's how the datapath would look like:

Figure 1. Datapath Abstract Block Diagram



The "clocked" rectangles are the registers mentioned earlier[10]. Some of them have direct impact on logic of multi-cycle approach, meaning that removing them would result in design failure. Others are just there to shorten the critical path delay, allowing higher clock frequencies.

[10] The program counter is also a simple register in theory, but we decided to give it a separate identity due to its importance.

Designing Components

Comparing with Previous Assignment

The components used in this computer assignments are the same as the previous one. The only thing we altered was the ALU.

ALU

The new ALU operation table looks like this:

Table 13. ALU Operation Table

Operation Code Parameter	Operation Code	Operation
ADD	000	$S = A + B$
SUB	001	$S = A - B$
AND	010	$S = A\&B$
OR	011	$S = A\|B$
SLT	100	$S = A < B\ ?\ 1 : 0$
XOR	101	$S = A\hat{}B$
PASS	110	$S = B$

Adding “PASS” operation makes the “LUI” instruction easily executable. As for the implementation, the Verilog description of this new ALU looks like this:

Figure 2. ALU Verilog Description

```
40  module ALU #(parameter WL = 32) (OpCode, operand1, operand2, result, ZERO);
41
42      parameter[2:0]  ADD = 3'b000,
43                      SUB = 3'b001,
44                      AND = 3'b010,
45                      OR =  3'b011,
46                      SLT = 3'b100,
47                      XOR = 3'b101,
48                      PASS= 3'b110;
49
50      input [2:0] OpCode;
51      input signed [WL-1:0] operand1, operand2;
52      output signed [WL-1:0] result;
53      output ZERO;
54
55      reg signed [WL-1:0] result_temp;
56      assign result = result_temp;
57
58      always @(OpCode, operand1, operand2) begin
59          result_temp = {(WL){1'b0}};
60          case(OpCode)
61              ADD: begin result_temp = operand1 + operand2; end
62              SUB: begin result_temp = operand1 - operand2; end
63              AND: begin result_temp = operand1 & operand2; end
64              OR:  begin result_temp = operand1 | operand2; end
65              SLT: begin result_temp = (operand1 < operand2) ? {{(WL-1){1'b0}}, 1'b1} : {(WL){1'b0}}; end
66              XOR: begin result_temp = operand1 ^ operand2; end
67              PASS:begin result_temp = operand2; end
68              default: result_temp = operand2;
69          endcase
70      end
71
72      assign ZERO = ~|{result};
73  endmodule
```

Control Unit

We decided to change the control unit entirely, since we had the chance to redesign the whole thing. We partitioned the controller into 3 major parts:

- ALU Controller: Decides what the ALU does, based on what the Main Controller tells it.
- PC Controller: Decides when to update the program counter, based on what the Main Controller tells it.
- Main Controller: Handles the overall course of actions, including what the other two parts do

The detailed explanation of each partition is given in the next pages.

ALU Controller

We use the ALU to handle every arithmetic and logic operation we do. Sometimes, the operation is defined by “f3” and “f7”. But, sometimes it has no dependency on them. For example, we use the ALU to calculate the next value of PC. This happens in the flow of each and every instruction’s execution, and has no dependency on anything. So, the ALU’s function is predetermined in some situations, but depends on the “f3” and “f7” in other ones.

The Main Control unit handles these situations. It determines when the ALU must do an operation without checking “f3” and “f7”, and when the operation depends on them. But, it does not handle the dependency of the operation on “f3” and “f7”. For example, if an R-Type instruction is being executed, the ALU must act based of “f3” and “f7”, but the Main Controller only tells that it must check these signals and does not specify which operation to perform. That’s where the ALU Controller comes in. The ALU Controller checks the Main Controller’s output. If it determines the operation, the ALU controller follows. Otherwise, it decides the operation based of “f3” and “f7”.

The table of ALU Controller’s functionality looks like this:

Table 14. ALU Controller Functionality Table

ALUOp	f3, f7	ALU Operation
00	-	ADD
01	-	SUB
10	ADD	ADD
	SUB	SUB
	AND	AND
	OR	OR
	SLT	SLT
	XOR	XOR
11	-	PASS

Here’ ALUOp is the output of the Main Controller, it’s how it controls ALU Controller’s actions. ALU Operation is what the ALU does, and must be controlled by the ALU Controller.

The Verilog Description of the ALU Controller is shown in the next page.

Figure 3. ALU Controller Verilog Description

```
module ALU_Controller(ALUOp, OpCode, f3, f7, Al
    input [1:0] ALUOp;
    input [2:0] f3;
    input [6:0] f7;
    input [6:0] OpCode;
    output [2:0] ALUControl;
    reg    [2:0] ALUControl_temp;
    assign ALUControl = ALUControl_temp;

    parameter [2:0] ADD = 3'b000,
                    SUB = 3'b001,
                    AND = 3'b010,
                    OR =  3'b011,
                    SLT = 3'b100,
                    XOR = 3'b101,
                    PASS= 3'b110;

    parameter [1:0] ADD_OP     = 2'b00,
                    SUB_OP     = 2'b01,
                    CHECK_F_OP = 2'b10,
                    PASS_OP    = 2'b11;

    parameter [6:0] ADD_OPC =   7'd51,
                    SUB_OPC =   7'd51,
                    AND_OPC =   7'd51,
                    OR_OPC =    7'd51,
                    SLT_OPC =   7'd51,
                    LW_OPC =    7'd3,
                    ADDI_OPC =  7'd19,
                    XORI_OPC =  7'd19,
                    ORI_OPC =   7'd19,
                    SLTI_OPC =  7'd19,
                    JALR_OPC =  7'd103,
                    SW_OPC =    7'd35,
                    JAL_OPC =   7'd111,
                    BEQ_OPC =   7'd99,
                    BNE_OPC =   7'd99,
                    LUI_OPC =   7'd55;

    parameter [2:0] ADD_F3 =    3'd0,
                    SUB_F3 =    3'd0,
                    AND_F3 =    3'd7,
                    OR_F3 =     3'd6,
                    SLT_F3 =    3'd2,
                    LW_F3 =     3'd2,
                    ADDI_F3 =   3'd0,
                    XORI_F3 =   3'd4,
                    ORI_F3 =    3'd6,
                    SLTI_F3 =   3'd2,
                    JALR_F3 =   3'd0,
                    SW_F3 =     3'd2,
                    BEQ_F3 =    3'd0,
                    BNE_F3 =    3'd1;

    parameter [6:0] ADD_F7 =    7'd0,
                    SUB_F7 =    7'd32,
                    AND_F7 =    7'd0,
                    OR_F7 =     7'd0,
                    SLT_F7 =    7'd0;

    always @(ALUOp, f3, f7) begin
        ALUControl_temp = ADD;
        case(ALUOp)
            ADD_OP: begin ALUControl_temp = ADD; end
            SUB_OP: begin ALUControl_temp = SUB; end
            PASS_OP:begin ALUControl_temp = PASS; end
            CHECK_F_OP: begin
                if ( (OpCode ==  ADD_OPC) & (f3 == ADD_F3) & (f7 == ADD_F7) ) begin ALUControl_temp = ADD ; end
                else if ( (OpCode ==  SUB_OPC) & (f3 == SUB_F3) & (f7 == SUB_F7) ) begin ALUControl_temp = SUB ; end
                else if ( (OpCode ==  AND_OPC) & (f3 == AND_F3) & (f7 == AND_F7) ) begin ALUControl_temp = AND ; end
                else if ( (OpCode ==  OR_OPC) & (f3 == OR_F3) & (f7 == OR_F7) ) begin ALUControl_temp = OR ; end
                else if ( (OpCode ==  SLT_OPC) & (f3 == SLT_F3) & (f7 == SLT_F7) ) begin ALUControl_temp = SLT ; end
                else if ( (OpCode ==  LW_OPC) & (f3 == LW_F3) ) begin ALUControl_temp = ADD ; end
                else if ( (OpCode ==  ADDI_OPC) & (f3 == ADDI_F3) ) begin ALUControl_temp = ADD ; end
                else if ( (OpCode ==  XORI_OPC) & (f3 == XORI_F3) ) begin ALUControl_temp = XOR ; end
                else if ( (OpCode ==  ORI_OPC) & (f3 == ORI_F3) ) begin ALUControl_temp = OR ; end
                else if ( (OpCode ==  SLTI_OPC) & (f3 == SLTI_F3) ) begin ALUControl_temp = SLT ; end
                else if ( (OpCode ==  JALR_OPC) & (f3 == JALR_F3) ) begin ALUControl_temp = ADD ; end
                else if ( (OpCode ==  SW_OPC) & (f3 == SW_F3) ) begin ALUControl_temp = ADD ; end
                else if ( (OpCode ==  JAL_OPC) ) begin ALUControl_temp = ADD ; end
                else if ( (OpCode ==  BEQ_OPC) & (f3 == BEQ_F3) ) begin ALUControl_temp = SUB ; end
                else if ( (OpCode ==  BNE_OPC) & (f3 == BNE_F3) ) begin ALUControl_temp = SUB ; end
                else if ( (OpCode ==  LUI_OPC) ) begin ALUControl_temp = PASS ; end
            end
        endcase
    end
endmodule
```

PC Controller

There are three scenarios where the program counter needs to be updated:

- Going to the next instruction, located at the next memory word
- Jumping to a given address
- Branching from an instruction to somewhere else

This begs the need for a hardware that governs the PC, but responds to the Main Control unit itself. Here's a functional description for the PC controller:

Program counter gets updated if:

1. The address "PC+4" is calculated
2. The jumping address (for JAL, JAKR, etc.) is calculated
3. The current instruction is branch type, and the branch condition is met

Here's the Verilog description of this hardware:

Figure 4. PC Controller Verilog Description

```
 90  module PCController(PCUpdate, ZERO, BrEQ, BrNE, PCWrite);
 91      input PCUpdate;
 92      input ZERO;
 93      input BrEQ, BrNE;
 94      output PCWrite;
 95      reg PCWrite_temp;
 96      assign PCWrite = PCWrite_temp;
 97
 98      always @(PCUpdate, ZERO, BrEQ, BrNE) begin
 99          PCWrite_temp = 1'b0;
100          if(PCUpdate)     PCWrite_temp = 1'b1;
101          else if(BrEQ)    PCWrite_temp = ZERO ? 1'b1 : 1'b0;
102          else if(BrNE)    PCWrite_temp = ZERO ? 1'b0 : 1'b1;
103      end
104  endmodule
```

Signals "BrEQ", "BrNE", and "PCUpdate" come from the Main Control unit. They tell this hardware when to check the branch condition, when to jump directly, and when not to do anything. "PCWrite" is the module's output, and gets connected to PC's load-enable input.

Main Control Unit

Overall Description

So far, all we have done has been combinational. But, things change from this point on. We need to design a **Finite State Machine**, aka, **FSM**, to handle the execution of different instructions. This state machine must handle the flow of data through the datapath, by governing its control signals, it also must control the other two parts of the controller.

In order to design the controller, we need to have a step-by-step algorithm, fully describing what needs to happen in order for an instruction to execute. Here's that algorithm.

Table 15. Main Control Unit Tabular Explanation

R-Type	Fetch the instruction
	Decode the instruction and get operands
	Calculate the results
	Write back the results
I-Type (LW and JALR excluded)	Fetch the instruction
	Decode the instruction and get operands and the immediate value
	Calculate the results
	Write back the results
S-Type	Fetch the instruction
	Decode the instruction and get operands
	Calculate memory address
	Store the operand in the memory
J-Type	Fetch the instruction
	Decode the instruction and get operands
	update PC and calculate return address
	Write return address back to register file
B-Type	Fetch the instruction
	Decode the instruction and get operands
	update PC if branch condition is met
U-Type	Fetch the instruction
	Decode the instruction and get operands
	Calculate the immediate and "PASS" it through the ALU
	Write the results back at register file

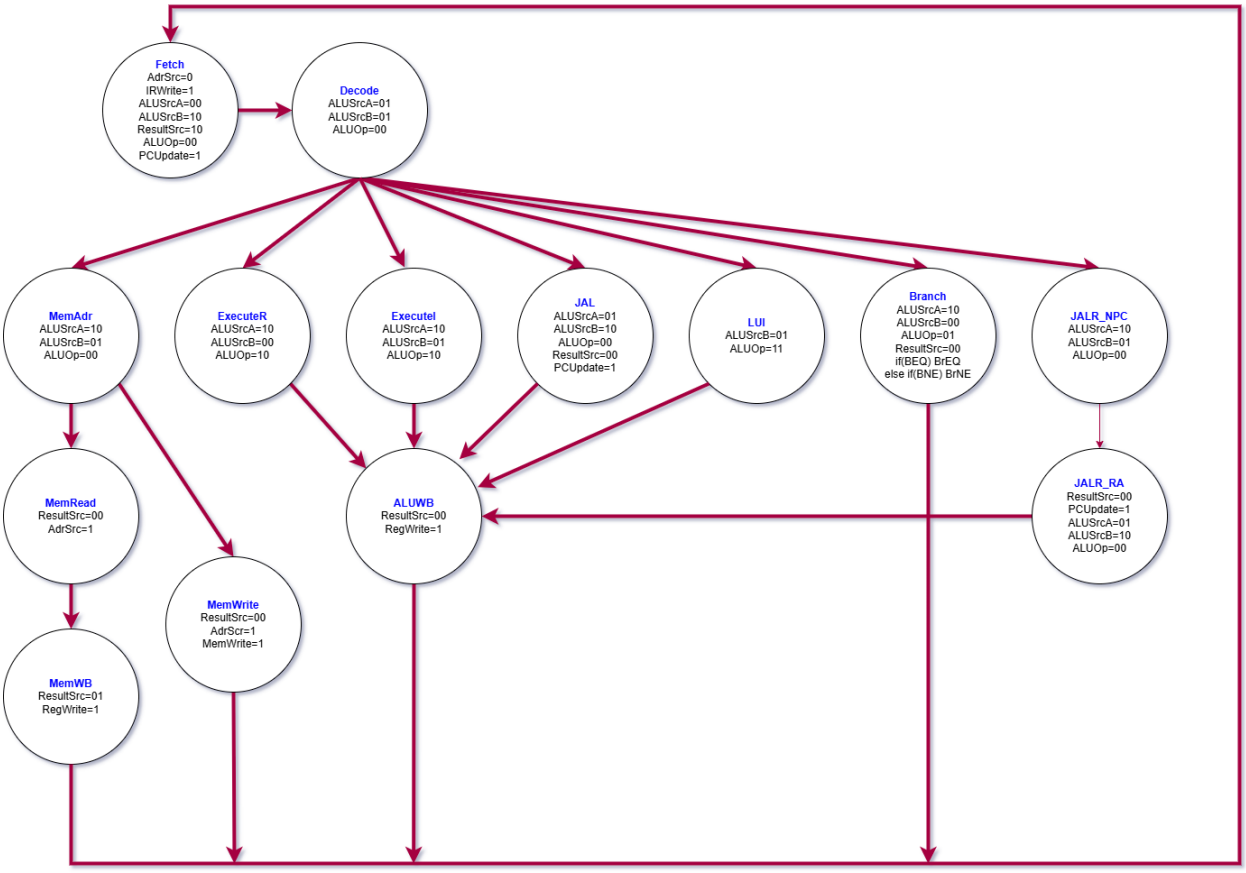
LW	Fetch the instruction
	Decode the instruction and get operands
	Calculate memory address
	Read from the memory
	Write the results back to the register file, from the memory path
JALR	Fetch the instruction
	Decode the instruction and get operands
	Calculate next PC
	Update the PC and calculate return address
	Write the return address back to register file

Now it is easy to implement the FSM. Also, a quick glance at the table shows that some of these steps are common between different instructions, making it possible to share the states between different instructions!

State Diagram

The state diagram for this controller looks like this:

Figure 5. Main Control Unit State Diagram



All there is left at this point, is to describe this in Verilog. Since the description was way too big (240 lines of code, give or take), we did not include it in the report.

Testing and Verification

Writing Assembly Code

We were asked to test the processor we implemented with a piece of code that searches an array of 10 integers for its minimum value. Here's how it's done in C programming language:

Figure 6. Minimum Finder Code in C language

```
1
2   int i = 0;
3   minimum = A[0];
4   i = i + 1;
5   for(; i < 10 ; i = i + 1){
6       temp = A[i];
7       if(minimum < temp)
8           minimum = temp;
9   }
```

Since we are to implement this in assembly language afterwards, we could change a thing or two to make the conversion easier. Here's the new code, written in a format between assembly and C:

Figure 7. Assembly-Friendly Minimum Finder Code

```
1   int i = 0;
2   minimum = A(i);
3
4   while(1){
5       i = i + 4;
6       if(i >= 40) break;
7       temp = A(i);
8       if(minimum < temp)
9           minimum = temp;
10  }
```

Now this can be easily turned into assembly code. We assumed that the array's first element is located at mem[0x5b0], since that is what we did for this course's second homework to make it compatible with the introduced website. Also, we chose our registers according to the table at the next page.

Register	Purpose	Register Number (X_i)
S_2	Reading data from memory (temp)	X_{18}
S_1	Saving i	X_9
S_0	Saving minimum	X_8
t_1	Saving SLT comparisons' results	X_6

Now we can easily write the assembly code for this.

Figure 8. Assembly Code for Minimum Finder

```
1     lw   s0, 0x5b0(zero)         # minElement = mem[0x5b0] (initialize with the first element)
2     add  s1, zero, zero          # i = 0
3  Loop:
4     addi s1, s1, 4               # i += 4
5     slti t1, s1, 40              # check if 10 elements are traversed (40 = 4 * 10)
6     beq  t1, zero, EndLoop       # if 10 elements are traversed, jump to EndLoop
7     lw   s2, 0x5b0(s1)           # element = mem(i)
8     slt  t1, s2, s0              # check if element is smaller than minElement
9     beq  t1, zero, Loop          # if element is not smaller than minElement, jump to Loop
10    add  s0, s2, zero            # minElement = element
11
12    jal zero,  Loop                    # jump to Loop
13 EndLoop:
14    sw s0, 0x5ac(zero)           # mem[2000] = minElement
15 TRAP:
16    jal zero, TRAP
```

All there is left to do is generate machine code from this and fill the memory with. Note that since the multi-cycle approach is built on hardware reuse and small combinational paths, we have to merge the data memory and the instructions memory, thus having a single memory. This leads to having one memory file, and we need to make sure the data is written in a place that has no overlap with the instructions.

Generating RISC-V Machine Code

Figure 9. Assembly and Machine Code

```
1     lw   s0, 0x5b0(zero)          //0x5b002403
2     add  s1, zero, zero           //0x000004b3
3  Loop:
4     addi s1, s1, 4                //0x00448493
5     slti t1, s1, 40               //0x0284a313
6     beq  t1, zero, EndLoop        //0x00030c63
7     lw   s2, 0x5b0(s1)            //0x5b04a903
8     slt  t1, s2, s0               //0x00892333
9     beq  t1, zero, Loop           //0xfe0306e3
10    add  s0, s2, zero             //0x00090433
11
12    jal zero,  Loop               //0xfe5ff06f
13 EndLoop:
14   sw s0, 0x5ac(zero)             //0x5a802623
15 TRAP:
16   jal zero TRAP                 //0x0000006f
```

The assembly codes are translated to RISC-V machine language in Hex (the related hex instruction of each line of the assembly code is shown in Figure 10).

To be written in the instruction memory, each hex instruction is first separated into four 2-digit hex numbers, and then written upside down (the 2 least significant hex digits first) in the instruction memory.

For example, the hexadecimal instruction 0x5ba40903 is written in the memory file like this:

03

09

A4

5B

This reversing is needed for the instructions to be properly read from the memory (2 most significant hex digits first).

We also did this reversing for data to be stored in the data memory. The data we used is this array:

33, 114, 51, 62, -145, -127, 61, 84, 41, 15

Testbench and Results

We used this testbench to test the code:

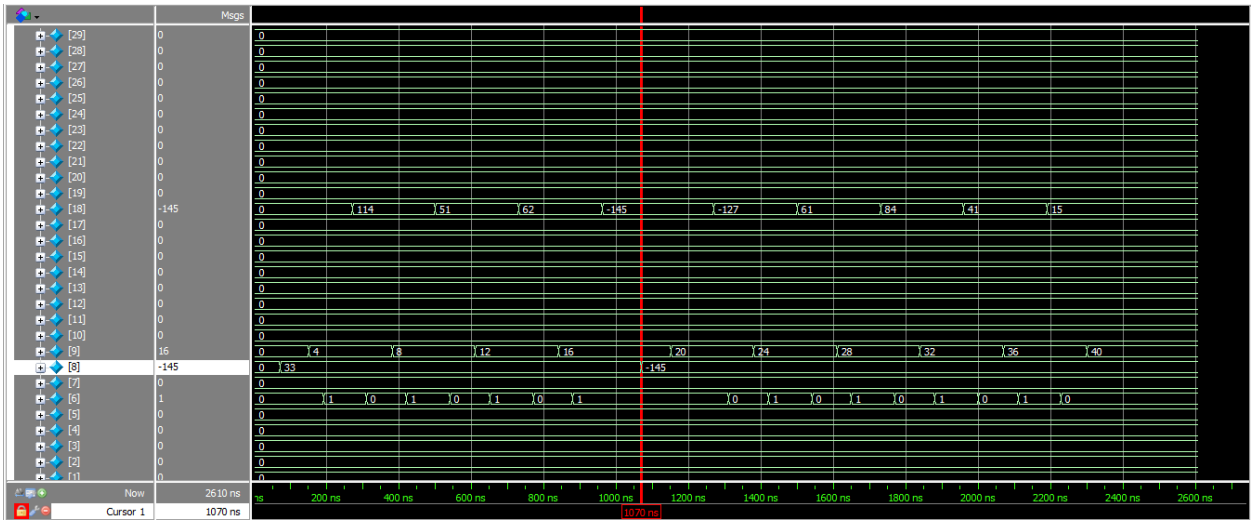
Figure 10. Top Module Testbench

```
115     module RISC_V_TB();
116
117         reg clk,rst;
118
119         RISC_V UUT (clk, rst);
120
121         initial begin clk = 1 ; rst = 1 ;#25 rst = 1'b0; end
122         always #5 clk = ~clk;
123         initial begin #2610 $stop; end
124
125     endmodule
```

What is directly to be objected here, is the fact that we have waited a lot longer than before for the results to get ready. The answer is, we did not design these testbenches[11] to be compared to each other. For the comparison to be possible, we need to synthesize both modules in the same technology boundaries, extract the delay values, and clock them both to their maximum allowed frequency and see which is faster. Anyway, the results of this testbench are represented in the next page.

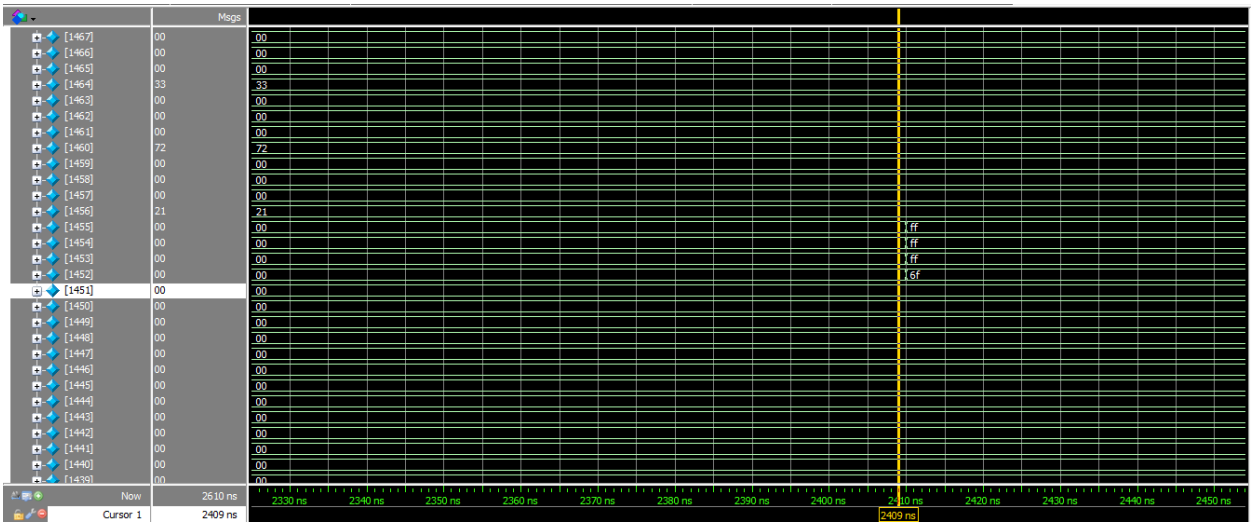
[11] This testbench, and the one for single-cycle processor of the previous computer assignment

Figure 11. Register File Waveforms



The value of the 8th register becomes -145, and stays the same for the rest of the testbench, since this is the minimum value.

Figure 12. Memory Waveforms



The results were also written on the memory, as can be seen in the assembly code. This waveform confirms that aspect as well.

THE END