Computer Assignment 1

RISC-V Processor

Single Cycle Approach

November 23rd, 2024

Bardia Amirian | 810101605
Amirreza Nadi Chaghadari | 810101538

Digital Systems 2

Dr. Safari

Table of Contents

Table of Contents

Table of Figures

Table of Tables

Problem Description

We are to design a simplified version of the RISC-V processor, using single-cycle approach. It means that each instruction takes exactly one clock cycle to execute. Here are 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)

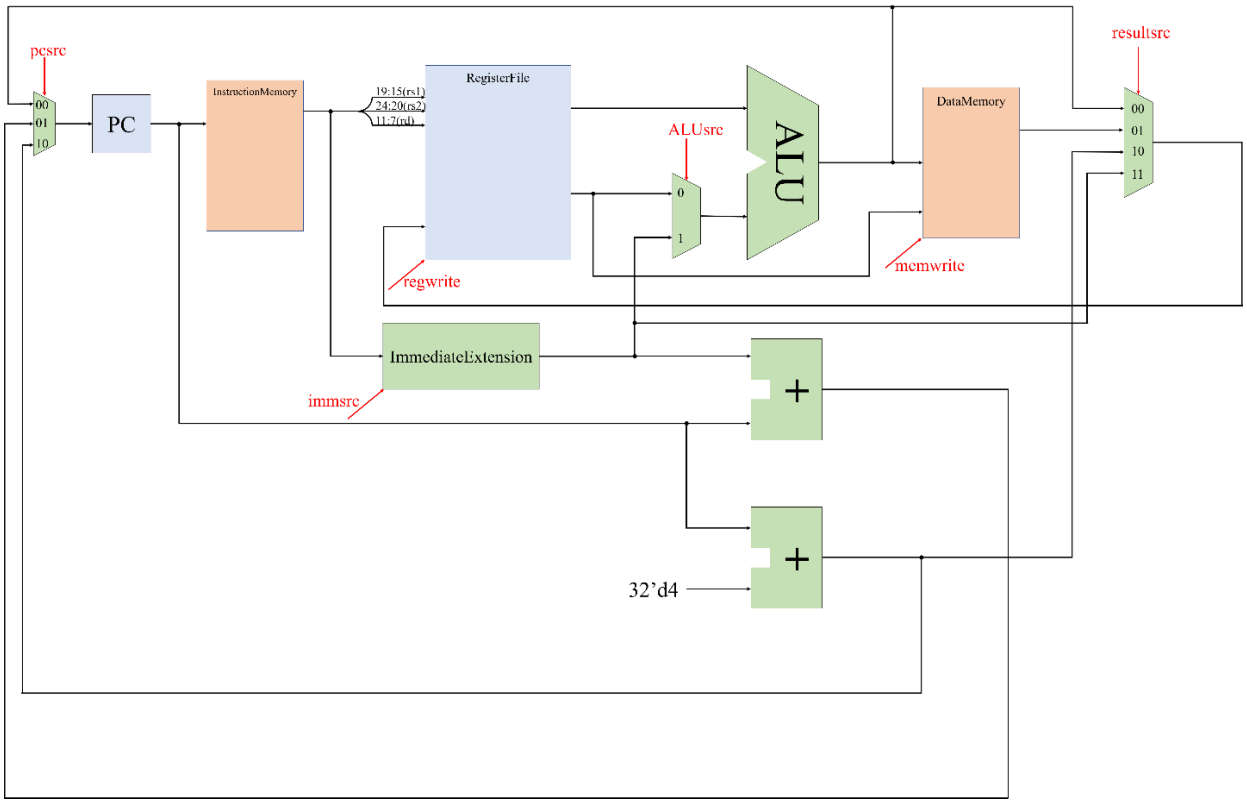
With these tables at hand, we can design the datapath.

Datapath Design

Abstract Block Diagram

The block diagram of the datapath looks like this:

Figure 1. Datapath Block Diagram



Where red signals come from the controller, edged connections (ones that are marked by a dot) mean the entire wire array is given to both branches, and rounded branches (at the input of the Register File) mean that the wire array is split into parts.

Here, the combination of "PC" and "Instruction Memory" handles the flow of the code being executed on the process. Instruction Memory's output is connected to the "Register File" to address the source and destination registers. It is also given to the Immediate Extension block in order to extract immediate values out of it. The ALU has 2 inputs. One comes from the Register File itself, and the other is the output if a MUX choosing between another register or the immediate value. This allows the execution of R-Type and I-Type instructions. By connecting Register File's second output to the input of "Data Memory", the datapath becomes capable of executing "sw" instruction. Also, the output of the memory id connected to the input of the Register File, making "lw" instruction's execution possible. Considering the fact that

the ALU has a “ZERO” flag, B-Type instructions are easily handled by subtracting the operands and using the ZERO flag to set PC’s next value. Also, J-Type instructions are easily done because of the two extra adders. As for the “lui” instruction, we can direct Immediate Extension’s output to the Register File’s input. Therefore, all of the given instructions can be executed using this datapath.

Next step is to design and implement each component used in the datapath, instantiate them, and assemble the datapath by wiring the instances together.

Designing Components

MUX[10]

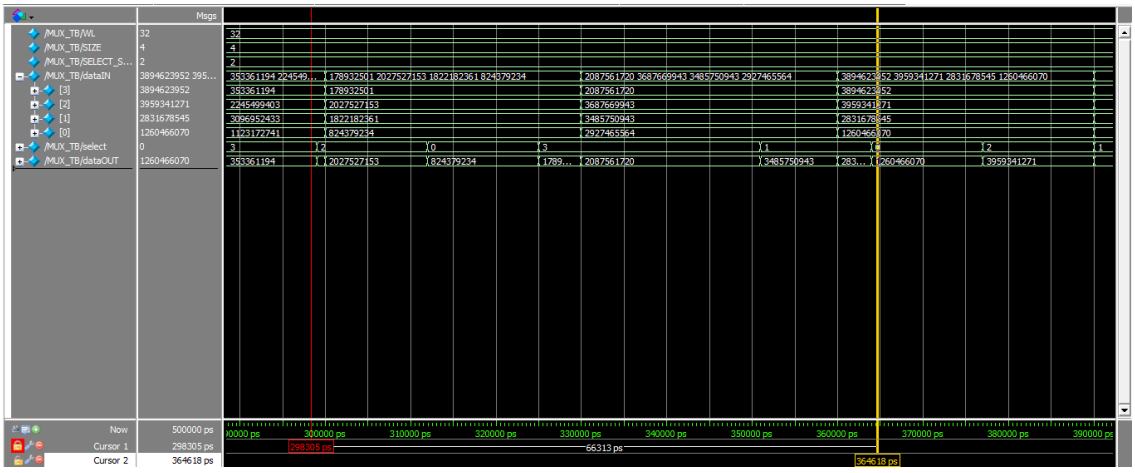
Multiplexer is one of the easiest purely-combinational components we are to implement. The Verilog description of it looks like this:

Figure 2. MUX Verilog Description

```
1   `timescale 1ns/1ps
2
3   module MUX_TB();
4
5       parameter WL = 32, SIZE = 4;
6       parameter SELECT_SIZE = $clog2(SIZE);
7
8       reg [SIZE-1:0][WL-1:0] dataIN;
9       reg [SELECT_SIZE-1:0] select;
10
11      wire [WL-1:0] dataOUT;
12
13      MUX #(SIZE, WL) UUT (dataIN, select, dataOUT);
14
15      always #30 dataIN = {$random, $random, $random, $random, $random};
16
17      always #13 select = $random;
18
19      initial begin
20          #500 $stop;
21      end
22
23  endmodule
```

We also designed a simple testbench to test this MUX with random data, and got these results:

Figure 3. MUX Testbench Results



This waveform (the part between the two cursors) represents different things that could happen to the inputs, and the MUX's response to them, verifying the functionality of the MUX.

[10] Multiplexer

Adder

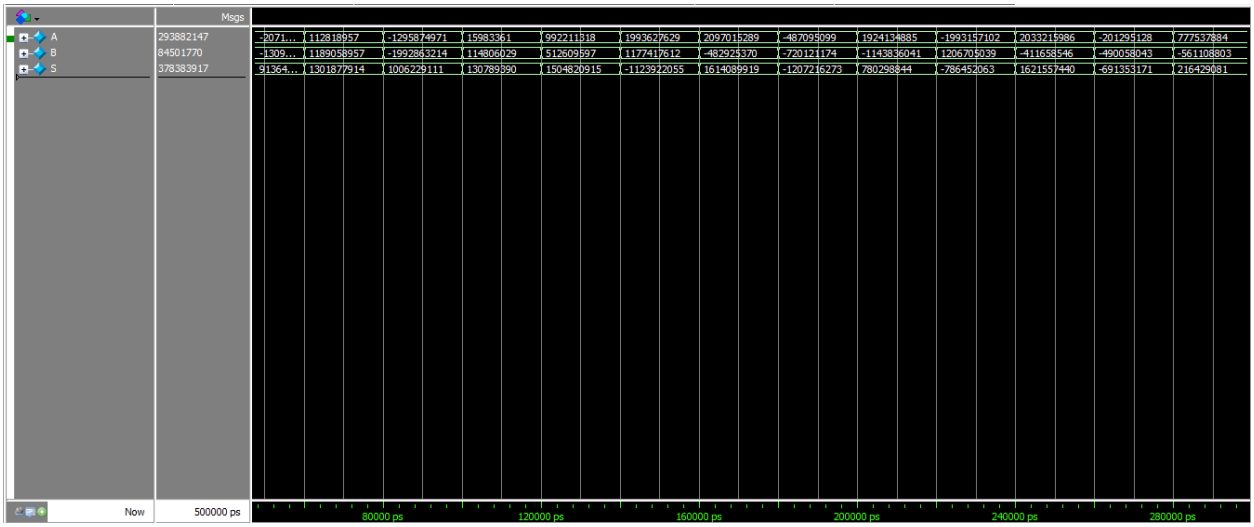
Adder is also simple to implement. Here's the Verilog description:

Figure 4. Adder Verilog Description

```
15  module Adder #(parameter SIZE = 32) (A, B, S);
16
17      input [SIZE-1:0] A;
18      input [SIZE-1:0] B;
19      output [SIZE-1:0] S;
20
21      assign S = A + B;
22  endmodule
```

As for the testing, this waveform confirms its functionality:

Figure 5. Adder Testbench Results



ALU

ALU can be simply implemented using an always block. But, we need to have a clear operation table first. Here's the one we used:

Table 13. ALU Op-Code Table

Op-Code[11]	Operation
000	Add
001	Subtract
010	AND
011	OR
100	SLT
101	XOR

Now we can describe it in Verilog. Here's the description:

Figure 6. ALU Verilog Description

```
26  module ALU #(parameter WL = 32) (OpCode, operand1, operand2, result, ZERO);
27
28      parameter   ADD = 3'b000,
29                  SUB = 3'b001,
30                  AND = 3'b010,
31                  OR =  3'b011,
32                  SLT = 3'b100,
33                  XOR = 3'b101;
34
35      input [2:0] OpCode;
36      input signed [WL-1:0] operand1, operand2;
37      output signed [WL-1:0] result;
38      output ZERO;
39
40      reg signed [WL-1:0] result_temp;
41      assign result = result_temp;
42
43      always @(OpCode, operand1, operand2) begin
44          result_temp = {(WL){1'b0}};
45          case(OpCode)
46              ADD: begin result_temp = operand1 + operand2; end
47              SUB: begin result_temp = operand1 - operand2; end
48              AND: begin result_temp = operand1 & operand2; end
49              OR:  begin result_temp = operand1 | operand2; end
50              SLT: begin result_temp = (operand1 < operand2) ? {{(WL-1){1'b0}}, 1'b1} : {(WL){1'b0}}; end
51              XOR: begin result_temp = operand1 ^ operand2; end
52              default: result_temp = operand1;
53          endcase
54      end
55
56      assign ZERO = &{result};
57  endmodule
```

[11] Operation Code

The inputs and output are defined as signed values, so that “slt” operation becomes possible. As for functionality confirmation, we used this testbench to test its response to different op-codes:

Figure 7. ALU Testbench

```
41  module ALU_TB();
42
43      parameter WL = 32;
44
45      parameter   ADD = 3'b000,
46                  SUB = 3'b001,
47                  AND = 3'b010,
48                  OR = 3'b011,
49                  SLT = 3'b100,
50                  XOR = 3'b101;
51
52      reg [2:0] OpCode;
53      reg signed [WL-1:0] operand1, operand2;
54      wire signed [WL-1:0] result;
55      wire ZERO;
56      ALU #(WL) UUT (OpCode, operand1, operand2, result, ZERO);
57
58      always #20 {operand1, operand2} = {$random, $random};
59
60      integer i;
61      initial begin
62          OpCode = ADD;
63          {operand1, operand2} = {$random, $random};
64          for (i = ADD ; i <= XOR ; i = i + 1) begin
65              OpCode = i;
66              #100;
67          end
68          #1
69          operand1 = 32'd25;
70          operand2 = -32'd25;
71          OpCode = ADD;
72          #20 $stop;
73      end
74
75  endmodule
```

The results are shown in the next page.

Figure 8. ALU Testbench Results

Immediate Extension

This part is nothing but wiring and multiplexing, and is done due to this table:

Table 14. Immediate Formatting

immsrc	Type	Size	Format
000	I-Type	12	{{20{Instr[31]}}, Instr[31:20]}
001	S-Type	12	{{20{Instr[31]}}, Instr[31:25], Instr[11:7]}
010	B-Type	13	{{20{Instr[31]}}, Instr[7], Instr[30:25], Instr[11:8], 1'b0}
011	J-Type	21	{{12{Instr[31]}}, Instr[19:12], Instr[20], Instr[30:21], 1'b0}
100	U-Type	21	{Instr[31:12], 12'b0}

Where "Instr" stands for instruction. The rest is just a matter of writing this in Verilog.

Figure 9. Immediate Extension Verilog

```
module ImmediateExtension (ImmSrc, Inst, Imm);

    parameter[2:0]  I_TYPE =    2'b000,
                    S_TYPE =    2'b001,
                    B_TYPE =    2'b010,
                    J_TYPE =    2'b011,
                    U_Type =    2'b100;

    input   [2:0]   ImmSrc;
    input   [31:0]  Inst;
    output  [31:0]  Imm;

    reg [31:0] Imm_temp;
    assign Imm = Imm_temp;

    always @(ImmSrc, Inst) begin
        Imm_temp = 32'b0;
        case(ImmSrc)
            I_TYPE: begin Imm_temp = {{20{Inst[31]}}, Inst[31:20]}; end
            S_TYPE: begin Imm_temp = {{20{Inst[31]}}, Inst[31:25], Inst[11:7]}; end
            B_TYPE: begin Imm_temp = {{20{Inst[31]}}, Inst[7], Inst[30:25], Inst[11:8], 1'b0}; end
            J_TYPE: begin Imm_temp = {{12{Inst[31]}}, Inst[19:12], Inst[20], Inst[30:21], 1'b0}; end
            U_Type: begin Imm_temp = {Inst[31:12], 12'b0}; end
            default: Imm_temp = 32'b0;
        endcase
    end

endmodule
```

With this, all combinational components are ready. The rest of the components are sequential. But due to their simplicity[12], we did not use "Huffman Coding Style" to implement them.

[12] All of them are simple storage units, making them easy to describe in a single always block.

PC (Program counter)

PC is just a simple register. Here's how it's described in Verilog:

Figure 10. Program Counter Verilog Description

```
85
86  module ProgramCounter #(parameter SIZE = 32) (parallelIN, clk,rst, parallelOUT);
87
88      input [SIZE-1:0] parallelIN;
89      input clk,rst;
90      output [SIZE-1:0] parallelOUT;
91
92      reg [SIZE-1:0] Storage;
93      assign parallelOUT = Storage;
94
95      always @(posedge rst, posedge clk) begin
96          if(rst) Storage <= {(SIZE){1'b0}};
97          else    Storage <= parallelIN;
98      end
99
100 endmodule
```

The rest of the components are memory-type ones. For better test-automation, we decided to use some extra files to store some pre-determined data in them.

Instruction Memory

We decided to save the instructions inside a file and read the file upon simulation. Here's how it's done:

Figure 11. Instruction Memory Verilog Description

```
102
103  module InstructionMemory (pc, instruction);
104
105
106      input [31:0] pc;
107      output [31:0] instruction;
108
109      reg [7:0] instMem [$pow(2, 16)-1:0];
110
111      wire [31:0] adr;
112      assign adr = {pc[31:2], 2'b00};
113
114      initial $readmemh("Ins_Fin.mem", instMem);
115
116      assign inst = {instMem[adr + 3], instMem[adr + 2], instMem[adr + 1], instMem[adr]};
117
118  endmodule
```

Thankfully, Verilog has the "$readmemh" function and we do not need to read the file line-by-line manually. The file must have ".mem" postfix, and each line of it must contain 2 hexadecimal digits.

Data Memory

This memory needs to be loaded as well, but it also needs initialization to get the data we aim to sort. Therefore, we decided to do the initialization upon the negative edge of the restart signal. Here's how the Verilog description looks like:

Figure 12. Data Memory Verilog

```
121  module DataMemory (addressIN, dataIN, writeEN, clk,rst, dataOUT);
122
123      input [31:0] addressIN;
124      input [31:0] dataIN;
125      input clk,rst, writeEN;
126      output [31:0] dataOUT;
127
128
129      reg [7:0] dataMem [$pow(2, 16)-1:0];
130
131      wire [31:0] adr;
132      assign adr = {addressIN[31:2], 2'b00};
133
134      always @(negedge rst) begin
135          $readmemh("Data.mem", dataMem);
136      end
137
138      initial $readmemh("Data.mem", dataMem);
139
140      integer i;
141      always @(posedge clk, posedge rst) begin
142          if(rst) begin
143              for(i = 0 ; i < $pow(2, 16); i = i + 1) dataMem[i] = 0;
144          end
145          else begin
146              if (writeEN)
147                  {dataMem[adr + 3], dataMem[adr + 2], dataMem[adr + 1], dataMem[adr]} <= dataIN;
148          end
149      end
150
151
152      assign dataOUT = {dataMem[adr + 3], dataMem[adr + 2], dataMem[adr + 1], dataMem[adr]};
153
154  endmodule
```

Register File

This part is easy, since it is just a simple double-output single-input register array. Here's the Verilog code:

Figure 13. Register File

```
105  module RegisterFile (rs1, rs2, rd, writeData, regwrite, clk,rst, data1, data2);
106
107      input [4:0] rs1, rs2, rd;
108      input [31:0] writeData;
109      input regwrite;
110      input clk,rst;
111      output [31:0] data1, data2;
112
113      reg [31:0][31:0] RegFile;
114
115      always @(posedge clk, posedge rst) begin
116          if(rst) RegFile = 0;
117          else begin
118              if(regwrite & (rd != 5'b00000)) RegFile[rd] <= writeData;
119          end
120      end
121
122      assign data1 = RegFile[rs1];
123      assign data2 = RegFile[rs2];
124
125  endmodule
```

What is important here, is line 118. Writing in "ZERO" register is not allowed. It must always remain 0. The rest is just a matter of multiplexing.

With this, all the components are ready, allowing us to proceed to the next phase.

Assembling Datapath

With all the components ready at hand, we can assemble the datapath by instantiating the components and wiring them together. Here's how the Verilog code looks like:

Figure 14. Datapath Module Verilog Description

```
module Datapath(pcsrc, ImmSrc, regwrite, ALUsrc, OpCode, memwrite, resultsrc, clk,rst, ZERO, instruction);

    input regwrite, ALUsrc, memwrite;
    input [1:0] pcsrc, ImmSrc, resultsrc;
    input [2:0] OpCode;
    input clk,rst;
    output ZERO;
    output [31:0] instruction;

    wire [31:0] PCIN, PCOUT;
    ProgramCounter #(32) PC (.parallelIN(PCIN), .clk(clk),.rst(rst), .parallelOUT(PCOUT));

    InstructionMemory IMEM (.pc(PCOUT), .instruction(instruction));

    wire [31:0] data1, data2, writeData;
    RegisterFile regfile (.rs1(instruction[19:15]), .rs2(instruction[24:20]), .rd(instruction[11:7]),
                          .writeData(writeData), .regwrite(regwrite), .clk(clk),.rst(rst), .data1(data1), .data2(data2));

    wire [31:0] immediate, ALU_data2;
    MUX #(2, 32) ALU_input_MUX (.dataIN({data2, immediate}), .select(ALUsrc), .dataOUT(ALU_data2));

    wire [31:0] ALU_out;
    ALU #(32) (.OpCode(OpCode), .operand1(data1), .operand2(ALU_data2), .result(ALU_out), .ZERO(ZERO));

    wire [31:0] DataMemOut;
    DataMemory DMEM (.addressIN(ALU_out), .dataIN(data2), .writeEN(memwrite), .clk(clk),.rst(rst), .dataOUT(DataMemOut));

    ImmediateExtension imm_ext (.ImmSrc(ImmSrc), .Inst(instruction), .Imm(immediate));

    wire [31:0] jump_adr, NEXT_PC;
    Adder #(32) (.A(immediate), .B(PCOUT), .S(jump_adr));
    Adder #(32) (.A(PCOUT), .B(32'd4), .S(NEXT_PC));

    MUX #(3, 32) pc_src_mux (.dataIN({ALU_out, jump_adr, NEXT_PC}), .select(pcsrc), .dataOUT(PCIN));
    MUX #(4, 32) resultsrc_mux (.dataIN({ALU_out, DataMemOut, NEXT_PC, immediate}), .select(resultsrc), .dataOUT(writeData));

endmodule
```

Now we need to design the controller. Since we are using single-cycle approach, there is no need for a state machine. All we need is a combinational logic.

Controller

The controller is just a simple lookup table. The rest is just a matter of using “always” blocks and “assign” statements to model the lookup tables. The code was way too long to be included here (170 lines in total). Therefore, we only included snapshots of one of the always blocks, handling “resultsrc” and “regwrite”.

Figure 15. Controller: Result Source Handling Always Block

```
125     always @(opc, f3, f7, ZERO) begin: Result_Source
126         resultsrc_temp = 2'b00;
127         regwrite_temp = 1'b0;
128         if ((opc == ADD_OPC) & (f3 == ADD_F3) & (f7 == ADD_F7)) begin       resultsrc_temp = 2'b00; regwrite_temp = 1'b1; end
129         else if ((opc == SUB_OPC) & (f3 == SUB_F3) & (f7 == SUB_F7)) begin  resultsrc_temp = 2'b00; regwrite_temp = 1'b1; end
130         else if ((opc == AND_OPC) & (f3 == AND_F3) & (f7 == AND_F7)) begin  resultsrc_temp = 2'b00; regwrite_temp = 1'b1; end
131         else if ((opc == OR_OPC) & (f3 == OR_F3) & (f7 == OR_F7)) begin     resultsrc_temp = 2'b00; regwrite_temp = 1'b1; end
132         else if ((opc == SLT_OPC) & (f3 == SLT_F3) & (f7 == SLT_F7)) begin  resultsrc_temp = 2'b00; regwrite_temp = 1'b1; end
133         else if ((opc == LW_OPC) & (f3 == LW_F3)) begin                     resultsrc_temp = 2'b01; regwrite_temp = 1'b1; end
134         else if ((opc == ADDI_OPC) & (f3 == ADDI_F3)) begin                 resultsrc_temp = 2'b00; regwrite_temp = 1'b1; end
135         else if ((opc == XORI_OPC) & (f3 == XORI_F3)) begin                 resultsrc_temp = 2'b00; regwrite_temp = 1'b1; end
136         else if ((opc == ORI_OPC) & (f3 == ORI_F3)) begin                   resultsrc_temp = 2'b00; regwrite_temp = 1'b1; end
137         else if ((opc == SLTI_OPC) & (f3 == SLTI_F3)) begin                 resultsrc_temp = 2'b00; regwrite_temp = 1'b1; end
138         else if ((opc == JALR_OPC) & (f3 == JALR_F3)) begin                 resultsrc_temp = 2'b10; regwrite_temp = 1'b1; end
139         // else if ((opc == SW_OPC) & (f3 == SW_F3)) begin                  resultsrc_temp = 2'b;   regwrite_temp = 1'b1; end
140         else if ((opc == JAL_OPC)) begin                                    resultsrc_temp = 2'b10; regwrite_temp = 1'b1; end
141         // else if ((opc == BEQ_OPC) & (f3 == BEQ_F3)) begin                resultsrc_temp = 2'b;   regwrite_temp = 1'b1; end
142         // else if ((opc == BNE_OPC) & (f3 == BNE_F3)) begin                resultsrc_temp = 2'b;   regwrite_temp = 1'b1; end
143         else if ((opc == LUI_OPC)) begin                                    resultsrc_temp = 2'b11; regwrite_temp = 1'b1; end
144     end
```

Now we can generate the top module and test it.

Top Module

Here’s how the top module looks like:

Figure 16. Top Module Verilog Description

```
module RISC_V (clk, rst);

    input clk,rst;

    wire regwrite, ALUsrc, memwrite;
    wire [1:0] pcsrc, resultsrc;
    wire [2:0] OpCode, ImmSrc;
    wire clk,rst;
    wire ZERO;
    wire [31:0] instruction;

    Datapath DPTH   (pcsrc, ImmSrc, regwrite, ALUsrc, OpCode, memwrite, resultsrc, clk,rst, ZERO, instruction);
    Controller CNT  (instruction, ZERO, pcsrc, ImmSrc, regwrite, ALUsrc, OpCode, memwrite, resultsrc);

endmodule
```

The only inputs of the top module are clock and reset signals. Since control unit is implemented in a purely combinational manner, it does not need clock or reset signals.

With the top module ready at hand, we could proceed to the testing phase.

Testing Phase

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 its done in C programming language:

Figure 17. 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 18. 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 19. 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 data and instruction memories.

Generating RISC-V Machine Code

Figure 20. 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 21).

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 hex instruction 0x5ba40903 is written in the memory 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

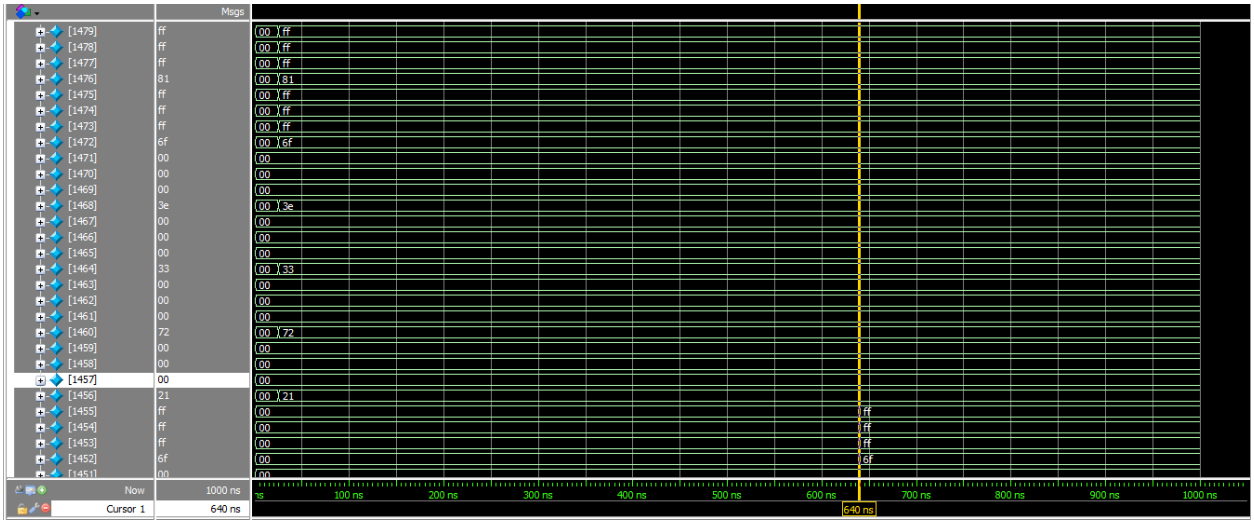
We used this testbench to test the code:

Figure 22. 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 #1000 $stop; end
124
125  endmodule
```

And the memory was updated like this:

Figure 23. Testbench Results



This confirms the minimum finding functionality.