EEE3530.02-00 (Computer Architecture)

Project #2 Building Data Path

2021142180 Kim Min Chan

Date: 2023.05.24

[Overview]

[1. Design]

1.1 Control Unit

1.2 Immediate Generator

1.3 Register

1.4 ALU

1.5 PC

[2. Results & Analysis]

2.1 Instruction Flow

2.2 Wave Form Simulation

2.3 Simulation Result & Analysis

[3. Discussion]

3.1 Additional Instruction

3.2 About Instruction Cache(register)

[0. Introduction]

This project aims to design a 64-bit single-cycle microprocessor based on RISC-V architecture and simulate its output by inputting the given instruction flow. The key aspect of this project is to implement the behaviors of each resource using Verilog HDL and connect the data paths between these resources to achieve the desired functionality according to the given instructions. In this project, limited instructions were used for the simplicity of the structure.

R-type Instr.: ADD/SUB/AND/OR/XOR

I-type Instr.: LD/ADDI/SUBI

S-type Instr. SD

SB-type Instr.: BEQ

[1. Design]

1.1 Control Unit

The Control Unit is a device that receives a 32-bit Instruction Field as input and outputs the corresponding control signals for the operation. The block responsible for performing the role of the Control Unit is called the 'control_single' block, which takes a 7-bit Opcode as input and outputs 7 control signals. Therefore, the input of 'control_single' is connected to the INSTRUCTION BUS, allowing INSTRUCTION[6..0] to be inputted. Additionally, the 7 control signals are connected to different blocks based on their respective roles.

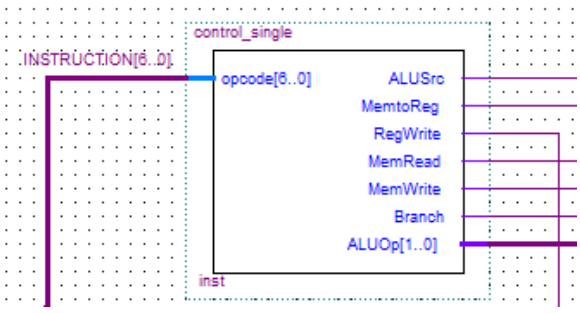

Figure 1.1.1 Control Unit

I have implemented the Control Unit using Verilog HDL to ensure its proper functioning for the intended purpose. Figure 1.1.2 represents the VHDL code, and each control signal performs the following roles:

(1) ALUSrc: Selects the input data for the ALU.

(2) MemtoReg: Selects the Write Back data for the Register.

(3) RegWrite: Determines whether to perform Write Back to the Destination Register.

(4) MemRead: Determines whether to read the value stored in a specific memory address.

(5) MemWrite: Determines whether to write a new value to a specific memory address.

(6) Branch: Determines whether to perform a branch operation.

(7) ALUop: Determines the operation to be performed by the ALU.

Based on this, I have set the values of each control signal for each instruction. For example, in the case of the R-type instruction, RD2 needs to be input to the ALU, so ALUSrc=0. The ALU operation result needs to be written to WD, so RegWrite=1. There is no access to memory, so MemRead and MemWrite are both 0. There is no branching involved, so Branch=0. Lastly, within RISC-V, the ALUOp for R-type is determined to be 10.

I have written the VHDL code for the "control_single" block based on this. You can find the code in the control_single.v file.

1.2 Immediate Generator

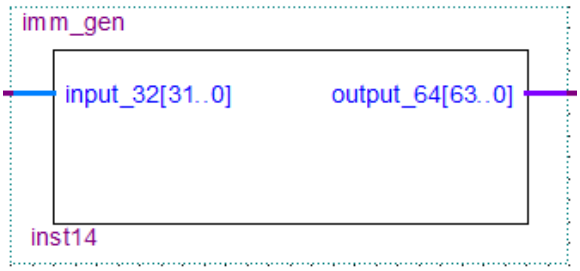

Figure 1.2.1 Block Diagram of Immediate Generator

The provided diagram represents the Block Diagram of the Immediate Generator in Figure 1.2.1. The Immediate Generator is a resource for instructions that require an immediate value, such as LD, SD, and BEQ.

The immediate value occupies 12 bits out of the 32 bits in the instruction. However, the location of the instruction field containing the immediate value varies depending on the instruction type. Therefore, the Immediate Generator accepts all 32-bit instructions as input. Based on the following table, the immediate value is sign-extended to 64 bits.

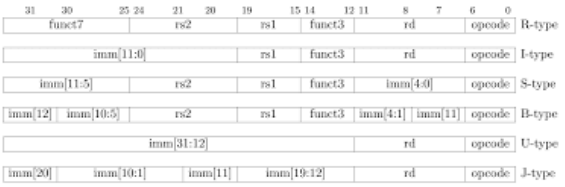

Figure 1.2.2 Instruction Field of Risc-V

Based on this, I have written the VHDL code for imm_gen, and you can find the code in 'imm_gen.v'.

1.3 Register

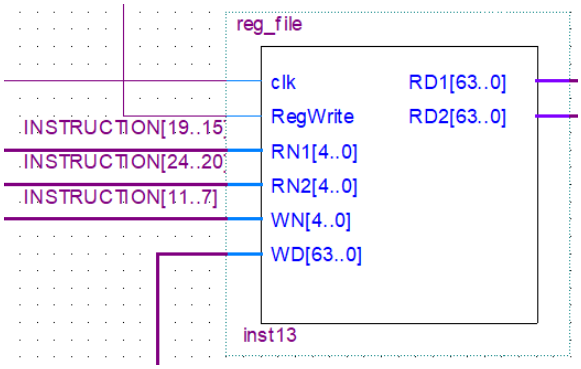

Figure 1.2.1 Block Diagram of Register

Registers serve the role of storing data. On the rising edge, data is written to the register at the Rd address based on the RegWrite signal, and on the falling edge, the data stored in the registers at the Rs1 and Rs2 addresses is read. Figure 1.2.1 represents the block diagram of the register. The register receives a total of 5 inputs: clk, RN1, RN2, WN, and WD. It takes RN1, RN2, and WN as inputs from the instruction. The instruction fields for RN1 and WN are consistent for all instructions and correspond to INSTRUCTION[19..15] and INSTRUCTION[11..7], respectively. For instructions requiring RN2, its position matches INSTRUCTION[24..20], allowing the construction of a datapath as shown in Figure 1.2.1.

As a sequential logic circuit performing read/write operations based on the clock, it takes clk as an input. The result of the ALU operation or data read from memory is input to WD. Additionally, the data read from the register is output as RD1 and RD2.

I have written VHDL code considering that writing is performed on the rising edge and reading is performed on the falling edge. It operates based on the clock, so I have used the syntax 'always @(negedge clk/posedge clk)'. You can find the corresponding code in the 'reg_file.v' file.

1.4 ALU

The ALU (Arithmetic Logic Unit) is a unit within the CPU that performs actual computations. In this project, we consider arithmetic operations such as addition and subtraction, as well as logical operations such as AND, OR, and XOR. To enable the ALU to perform the appropriate operation based on the instruction, a 4-bit control signal called ALUoperation is required.

To fulfill this requirement, an ALU control block has been designed. Figure 1.3.1 represents the block diagram of the ALU control, which determines the ALU operation based on the given instruction.

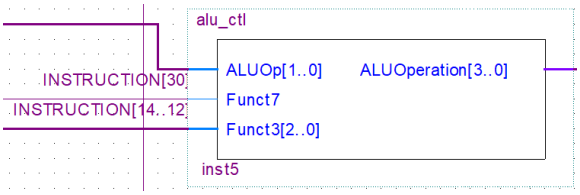

Figure 1.3.1 Block Diagram of ALU Control

As shown in Figure 1.3.1, the ALUOp, Funct7, and Funct3 are three inputs received by the ALU control. Among these, Funct7 and Funct3 are part of the instruction field, with their positions fixed at INSTRUCTION[31:25] and INSTRUCTION[14:12], respectively. However, for this project, only a limited set of operations is implemented, so only the 6th bit of the 7-bit Funct7 is required. Therefore, we can see that INSTRUCTION[30] is input to Funct7.

To perform the appropriate operation based on the instruction, I referred to the instruction format and opcode table from the textbook and determined the ALUoperation signal based on the input values. For example, for the ADD instruction, where ALUop=10, Funct7=0, and Funct3=000, if the three inputs match these values, the control signal ALUoperation will

output ALU_add=4'b0010, which corresponds to the addition operation. Another example is the BEQ instruction, which does not require Funct3 and Funct7, and the ALU_operation is solely determined by ALUOp. Since BEQ requires a subtraction operation, the output will be ALUoperation=ALU_sub=4'b0110. Based on this, I have written VHDL code, which can be found in the 'alu_ctl.v' file.

Based on that, you have designed the actual ALU responsible for performing the arithmetic and logical operations. Figure 1.3.2 represents the block diagram of the ALU, which illustrates its components and their interconnections for carrying out the operations.

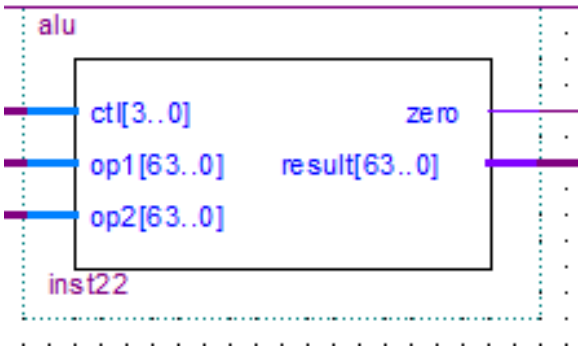

Figure 1.3.2 Block Diagram of ALU

As shown in Figure 1.3.2, the ALU receives three inputs: a control signal, op1, and op2. Based on the previously designed ALU Control, the ALU is designed to perform specific operations when it receives the corresponding control signal. For example, when an ADD instruction is executed and the ALU Control outputs ALUoperation=ALU_add=4'b0010, the ALU performs the operation op1+op2 when it receives 4'b0010 as the control signal. Another example is when a BEQ instruction is executed and the ALU Control outputs ALUoperation=ALU_sub=4'b0110, the ALU performs the operation op1-op2.

Based on this, the VHDL code for the ALU has been written, and it can be found in the 'alu.v' file.

1.5 PC

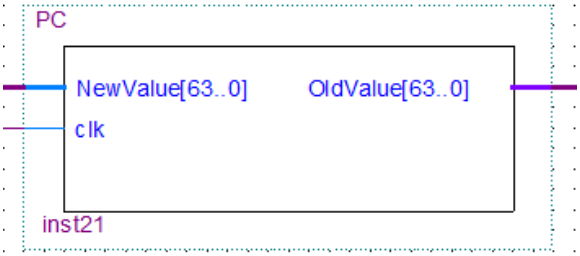

Figure 1.5.1 Block Diagram of PC

Figure 1.5.1 represents the block diagram of the PC (Program Counter). The PC is a sequential logic circuit where a new value is output based on the clock signal. On the rising edge of the clock, the NewValue is assigned to the OutputValue.

Next, a design has been devised to update the PC (Program Counter) based on the execution of instructions. In RISC-V, instructions are 32 bits long, and if byte-access registers are used, the address of the next instruction would be PC + 4. Typically, after the current instruction is completed, the next instruction located at Current PC + 4 is executed. However, in the case of a branch instruction, the instruction corresponding to Current PC + offset needs to be executed. To implement this functionality, the design shown in Figure 1.4.1 has been employed.

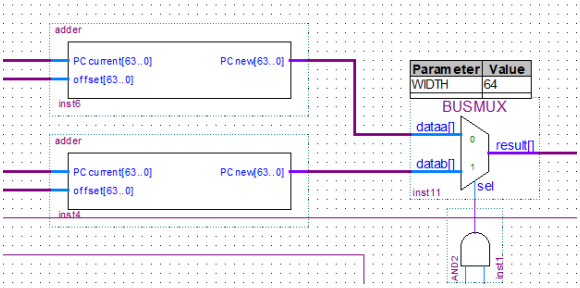

Figure 1.5.2 Block Diagram for PC increment

First, in the case of a BEQ instruction, when Branch=1 is assigned, it means that a branch is taking place. For BEQ, if the subtraction result is zero (i.e., zero=1), a branch should occur. Therefore, both the Branch and Zero control signals are input to an AND gate, and the output of the AND gate is used as the control signal for a MUX. If the MUX's control signal is 0, it means that no branch occurs, and PC+4 is selected. If the control signal is 1, indicating a branch, PC+offset is selected. Here, the offset is the output of the immediate generator.

In this design, an adder takes PCcurrent and the offset as inputs and outputs PCnew. PCcurrent is connected to the OldValue of the PC. The result of the MUX is connected to the NewValue of the PC.

[2. Results & Analysis]

2.1 Instruction Flow

Sequence	Instruction
1	add R4, R1, R2
2	sub R5, R3, R2
3	beq R1, R5, 0x440
4	add R4, R1, R5
5	sd R4, 20(R5)
6	add R6, R4, R1
7	ld R5, 6(R6)
8	or R2, R6, R5
9	xor R1, R4, R3
10	and R5, R1, R2
11	addi R1, R1, -7
12	ori R1, R1, 7

Figure 2.1.1 Instruction Flow

Sequence	Register No.						Mem	
	1	2	3	4	5	6	20(R5)=6(R6)	
0	7	2	9	x	x	x	x	x
1	7	2	9	9	x	x	x	x
2	7	2	9	9	7	x	x	x
3	7	2	9	9	7	x	x	x
4	7	2	9	14	7	x	x	x
5	7	2	9	14	7	x	14	14
6	7	2	9	14	7	21	14	14
7	7	2	9	14	14	21	14	14
8	7	31	9	14	14	21	14	14
9	7	31	9	14	14	21	14	14
10	7	31	9	14	7	21	14	14
11	0	31	9	14	7	21	14	14
12	7	31	9	14	7	21	14	14

Figure 2.1.2 Value stored in register and memory w.r.t sequence.

Figure 2.1.1 represents the Instruction Flow, while Figure 2.1.2 represents the values stored in registers and memory based on the flow.

Each instruction needs to be inputted as a 32-bit binary code, so I have converted each instruction into 32-bit machine language. Figure 2.1.3 represents the machine language for each sequence, organized by fields. When converting each instruction into a 32-bit code, I utilized the information provided in the textbook materials.

Sequence	Instr. Field					
	FUNCT7/ imm(7)	rs2/ imm(5)	rs1	funct3	rd/ imm(5)	opcode
1	0000000	00010	00001	000	00100	0110011
2	0100000	00010	00011	000	00101	0110011
3	0100010	00101	00001	000	00000	1100111
4	0000000	00101	00001	000	00100	0110011
5	0000000	00100	00101	111	10100	0100011
6	0000000	00001	00100	000	00110	0110011
7	0000000	00110	00110	011	00101	0000011
8	0000000	00101	00110	110	00010	0110011
9	0000000	00011	00100	100	00001	0110011
10	0000000	00010	00001	111	00101	0110011
11	1111111	11001	00001	000	00001	0010011
12	0000000	00111	00001	110	00001	0010011

Figure 2.1.3 Instruction 32-Bit Machine Language

2.2 Wave Form Simulation

Initial Values

Register No.	Initial Value
R1	7
R2	2
R3	9
PC	A0000000

Figure 2.2.1 Initial Value

Before proceeding with the actual simulation, it is necessary to initialize the values of the registers. The initialization process is carried out using INIT_Enable and INIT_Value. When INIT_Enable is set to 1, the WD (Write Data) is set to INIT_Value instead of the result from MUX. Therefore, instructions such as ADD with RegWrite=1 can be executed to perform the initialization. In this case, the destination register, Rd, should be set to the register you want to initialize.

Figure 2.2.2 represents the setting of input values for the Waveform Simulation. Since three registers need to be initialized, we can observe that INIT_Enable is active (set to 1) for three cycles. In each cycle, the desired initialization values are set as INIT_Value. The Clock Cycle starts from 1, and the value changes every 20ns, indicating a Clock Period of 40ns.

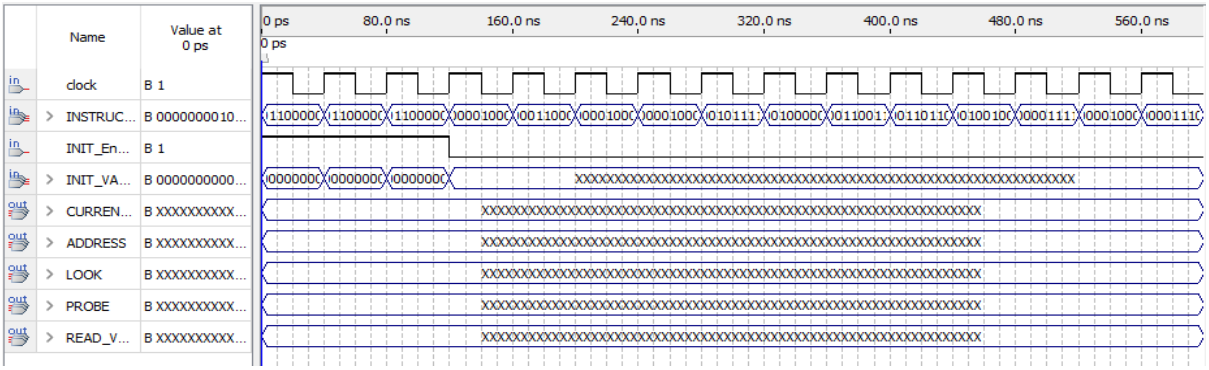

Figure 2.2.2 Wave Form Simulation Setting

2.3 Simulation Results & Analysis

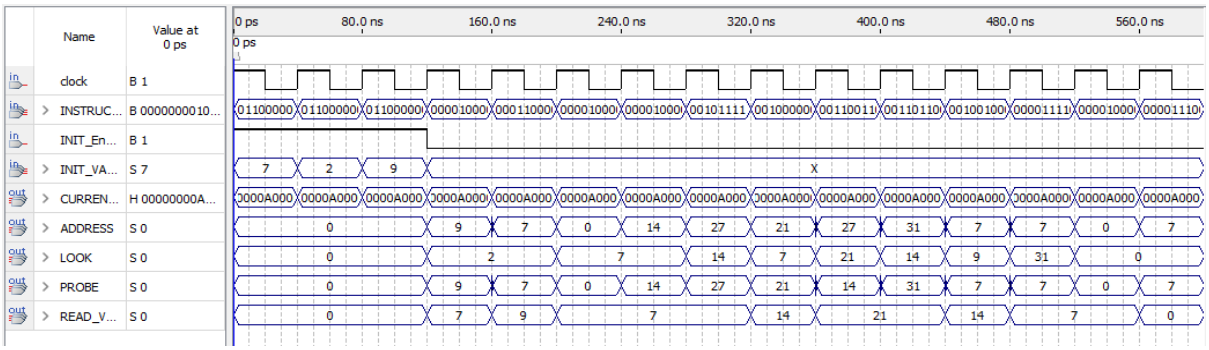

Figure 2.3.1 Wave Form Simulation Results

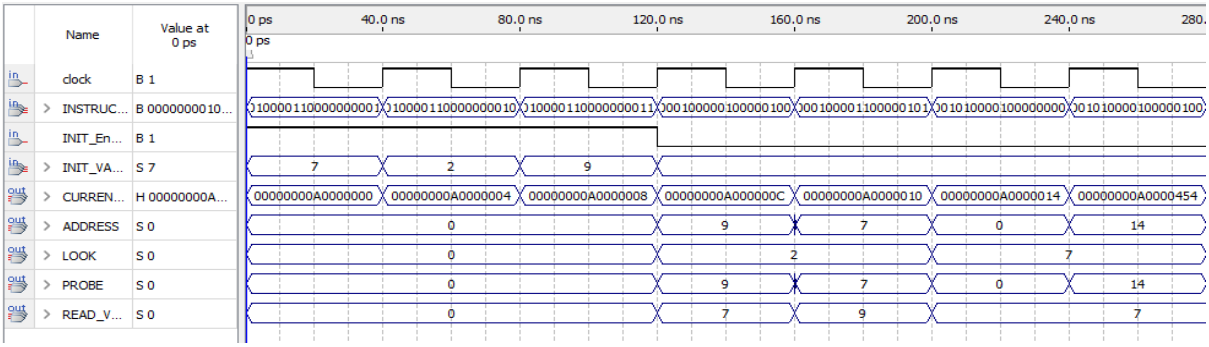

Figure 2.3.2 Wave Form Simulation Results: 0ns ~ 280ns

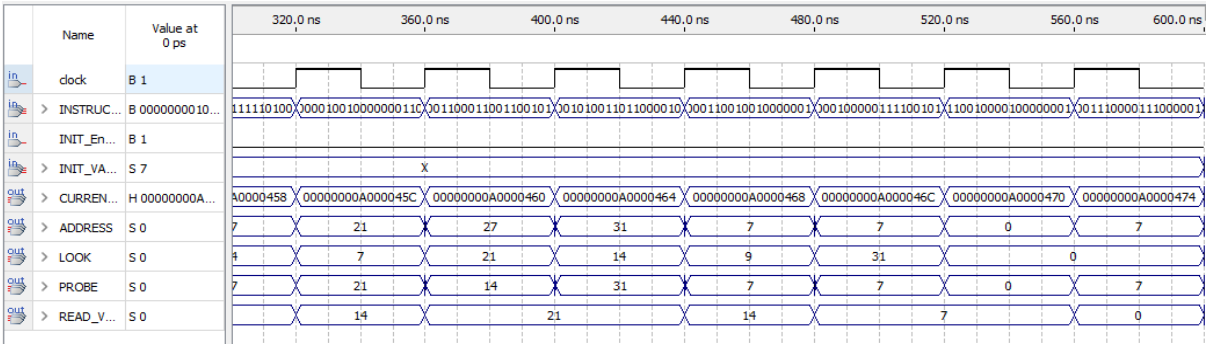

Figure 2.3.3 Wave Form Simulation Results: 280ns ~ 320ns

Figure 2.3.1 to Figure 2.3.3 represent the results of the waveform simulation. Through the outputs, we can verify if the instructions were executed correctly.

Firstly, CURRENT PC represents the current value of the Program Counter. ADDRESS represents the result of the ALU operation, LOOK represents the Data_in input to the memory, PROBE represents the data input to WD when INIT_Enable=0. Lastly, READ_VALUE represents the value of RD2.

By analyzing these output signals, we can determine if the instructions were executed as intended.

Through this, we can confirm that the microprocessor designed in this project is operating according to the input Instruction Flow. After the initial three cycles of initialization, we can observe meaningful results from the ADDRESS to READ_VALUE signals.

The first executed instruction has a PC value of 0x0C, which is the result of the initialization instructions executed in the first three cycles.

When the third instruction, 'beq R1, R5, 0X440,' is executed, we can see that ADDRESS is 0. This indicates that the beq instruction subtracted op1 from op2, resulting in 0. Therefore, the branch will be taken, and we can verify that the next PC value is not 4 but increased by 440. This confirms that the branch operation is functioning correctly.

When the fifth instruction, 'sd R4, 20(R5),' is executed, the value 14 stored in R4 is stored in memory at address 20 + 7 = 27. Then, in the seventh instruction, 'ld R5, 6(R6),' the value 14 stored in memory at address 6 + 21 = 27 is written into R5. By observing that LOOK is 14 in the fifth sequence and PROBE is 14 in the seventh sequence, we can confirm that these instructions are functioning correctly.

Overall, these waveform simulation results provide evidence that the microprocessor designed in this project is operating correctly according to the input Instruction Flow.

[3. Discussion]

3.1 Additional Instruction

In this project, the ADDi and Ori instructions were implemented as additional functionalities. Both instructions belong to the I-type category and require an immediate value. They involve specific operations in the ALU, namely addition and OR operations. To implement these instructions, the following code was added to the 'control_single.v' and 'alu_ctl' files:

To handle the ADDi instruction, the immediate value is assigned to the op2 operand, and the ALU performs the addition operation. For the Ori instruction, the immediate value is assigned to the op2 operand, and the ALU performs the OR operation:

By incorporating these modifications, the ADDi and Ori instructions were successfully implemented as additional functionalities in the project.

```
parameter ADDi      = 7'b0010011;
parameter ORi       = 7'b0010011;

7'b0010011: begin
   ALUSrc <= 1'b1;
   MemtoReg <= 1'b0;
   RegWrite <= 1'b1;
   MemRead <= 1'b0;
   MemWrite <= 1'b0;
   Branch <= 1'b0;
   ALUOp <= 2'b11;
   end
```

Figure 3.1.1 additional VHDL code of 'control_single.v'

```
parameter F3_addi = 3'b000;
parameter F3_ori =  3'b110;

parameter ALU_addi = 4'b0010;
parameter ALU_ori  = 4'b0001;

2'b11: begin
   if(Funct3==F3_addi) ALUOperation<=ALU_addi;
      else if(Funct3==F3_ori) ALUOperation<=ALU
   end
```

Figure 3.1.2 additional VHDL code for 'alu_ctl'

3.2 About Instruction Cache (register)

In the microprocessor implemented in this project, there is no Instruction Cache (Register) to store instructions. Therefore, the execution of instructions is not based on the PC value but on the INSTRUCTION input values set in the waveform simulation. This means that there is no dependency between the PC value and the execution of instructions.

As a result, even if a branch operation is executed, only the PC value changes, and the next instruction is executed. Thus, the microprocessor designed in this project lacks flexibility in terms of the instruction flow being fixed.

One possible solution to address this limitation is to introduce an Instruction Register or integrate the instruction cache into the design. For example, in this project, 64 64-bit registers are used. In the RISC-V architecture, 32 registers are used as data registers, so utilizing the remaining 32 registers as an instruction cache could potentially resolve this issue. By incorporating an instruction cache, the microprocessor would be able to fetch instructions based on the PC value and allow for a more dynamic instruction flow.

It's important to note that implementing an instruction cache or modifying the register usage would require careful consideration and potentially impact the overall design and functionality of the microprocessor. Any modifications should align with the specific requirements and goals of the project.

[4. Reference]

[1] David A. Patterson, John Hennessy. "*Computer Organization And Design*". Morgan Kaufmann.

APPENDIX

```
module control_single(opcode, ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch, ALUOp);
   input [6:0] opcode;
   output ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch;
   output [1:0] ALUOp;
   reg ALUSrc, MemtoReg, RegWrite, MemRead, MemWrite, Branch;
   reg [1:0] ALUOp;

   parameter R_FORMAT = 7'b0110011;
   parameter LD       = 7'b0000011;
   parameter SD       = 7'b0100011;
   parameter BEQ      = 7'b1100111;
   parameter ADDi     = 7'b0010011;
   parameter ORi      = 7'b0010011;

   always @(opcode) //when opcode is changed, following block is executed
   begin
      case (opcode)
      // fill in the misisng code
      7'b0110011: begin
         ALUSrc <= 1'b0;
         MemtoReg <= 1'b0;
         RegWrite <= 1'b1;
         MemRead <= 1'b0;
         MemWrite <= 1'b0;
         Branch <= 1'b0;
         ALUOp <= 2'b10;
         end
      7'b0000011: begin
         ALUSrc <= 1'b1;
         MemtoReg <= 1'b1;
         RegWrite <= 1'b1;
         MemRead <= 1'b1;
         MemWrite <= 1'b0;
         Branch <= 1'b0;
         ALUOp <= 2'b00;
         end
      7'b0100011: begin
         ALUSrc <= 1'b1;
         MemtoReg <= 1'bx;
         RegWrite <= 1'b0;
         MemRead <= 1'b1;
         MemWrite <= 1'b1;
         Branch <= 1'b0;
         ALUOp <= 2'b00;
         end
       7'b1100111: begin
          ALUSrc <= 1'b0;
          MemtoReg <= 1'bx;
          RegWrite <= 1'b0;
          MemRead <= 1'b0;
          MemWrite <= 1'b0;
          Branch <= 1'b1;
          ALUOp <= 2'b01;
          end
       7'b0010011: begin
          ALUSrc <= 1'b1;
          MemtoReg <= 1'b0;
          RegWrite <= 1'b1;
          MemRead <= 1'b0;
          MemWrite <= 1'b0;
          Branch <= 1'b0;
          ALUOp <= 2'b11;
          end
       // fill in the missing code

       endcase
    end
endmodule
```

Appendix A. VHDL code for control_single

```
module alu_ctl(ALUOp, Funct7, Funct3, ALUOperation);
   input [1:0] ALUOp;
   input Funct7;
   input [2:0] Funct3;
   output [3:0] ALUOperation;
   reg [3:0] ALUOperation;

   // symbolic constants for instruction function code
   parameter F7_add  = 1'b0;
   parameter F7_sub  = 1'b1;
   parameter F7_and  = 1'b0;
   parameter F7_or   = 1'b0;
   parameter F7_xor  = 1'b0;

   parameter F3_add  = 3'b000;
   parameter F3_sub  = 3'b000;
   parameter F3_and  = 3'b111;
   parameter F3_or   = 3'b110;
   parameter F3_xor  = 3'b100;

   parameter F3_addi = 3'b000;
   parameter F3_ori =  3'b110;

   // symbolic constants for ALU Operations
   parameter ALU_add  = 4'b0010;
   parameter ALU_sub  = 4'b0110;
   parameter ALU_and  = 4'b0000;
   parameter ALU_or   = 4'b0001;
   parameter ALU_xor  = 4'b0101;

   parameter ALU_addi = 4'b0010;
   parameter ALU_ori  = 4'b0001;

   always @(ALUOp or Funct7 or Funct3)
   begin
      case (ALUOp)
      // fill in the missing code
```

```
   always @(ALUOp or Funct7 or Funct3)
   begin
      case (ALUOp)
      // fill in the missing code

      2'b10: begin
         if(Funct7==F7_sub && Funct3==F3_sub) ALUOperation<=ALU_sub;
            else if(Funct7==F7_add && Funct3==F3_add) ALUOperation<=ALU_add;
            else if(Funct7==F7_and && Funct3==F3_and) ALUOperation<=ALU_and;
            else if(Funct7==F7_or && Funct3==F3_or) ALUOperation<=ALU_or;
            else if(Funct7==F7_xor && Funct3==F3_xor) ALUOperation<=ALU_xor;
         end

      2'b00: ALUOperation<=ALU_add;

      2'b01: ALUOperation<=ALU_sub;

      2'b11: begin
         if(Funct3==F3_addi) ALUOperation<=ALU_addi;
            else if(Funct3==F3_ori) ALUOperation<=ALU_ori;
         end


      // fill in the missing code

      endcase
   end
endmodule
```

Appendix B. VHDL code for alu_ctl

```
module alu(ctl, op1, op2, zero, result);
   input [3:0] ctl;
   input [63:0] op1, op2;
   output [63:0] result;
   output zero;

   reg [63:0] result;
   reg zero;

   // symbolic constants for ALU operations
   parameter ALU_add = 4'b0010;
   parameter ALU_sub = 4'b0110;
   parameter ALU_and = 4'b0000;
   parameter ALU_or  = 4'b0001;
   parameter ALU_xor = 4'b0101;

   parameter ALU_addi = 4'b0010;
   parameter ALU_ori = 4'b0001;

   always @(op1 or op2 or ctl)
   begin
      case (ctl)
      // fill in the missing code

      4'b0010: result <= op1 + op2;
      4'b0110: result <= op1 - op2;
      4'b0000: result <= op1 & op2;
      4'b0001: result <= op1 | op2;
      4'b0101: result <= op1 ^ op2;

      // fill in the missing code
      default : result <= result;
      endcase

      if (result == 63'd0)
      begin
         zero <= 1;
      end
      else
      begin
         zero <= 0;
      end
   end
endmodule
```

Appendix C. VHDL code for alu

```
module imm_gen (input_32, output_64);
   input [31:0] input_32;
   output [63:0] output_64;

   // fill in the missing code

   reg[63:0] output_64;

   always @(input_32)
   begin
      case (input_32[6:0])

         7'b0000011:
            begin
               output_64[11:0] <= input_32[31:20];
               output_64[63:12] <= {52{input_32[31]}};
            end

         7'b0100011:
            begin
               output_64[4:0] <= input_32[11:7];
               output_64[11:5] <= input_32[31:25];
               output_64[63:12] <= {52{input_32[31]}};
            end

         7'b0010011:
            begin
               output_64[11:0] <= input_32[31:20];
               output_64[63:12] <= {52{input_32[31]}};
            end

         7'b1100111:
            begin
               output_64[0] <= 1'b0;
               output_64[4:1] <= input_32[11:8];
               output_64[10:5] <= input_32[30:25];
               output_64[11] <= input_32[7];
               output_64[12] <= input_32[31];
               output_64[63:13] <= {52{input_32[31]}};
            end

         endcase
      end
   // fill in the missing code
endmodule
```

Appendix D. VHDL code for imm_gen

```
1
2       module PC (NewValue, OldValue, clk);
3       input [63:0] NewValue;
4       input clk;
5       output [63:0] OldValue;
6       reg [63:0] OldValue;
7
8       always@(posedge clk)
9       begin
10         OldValue <= NewValue;
11      end
12
13      initial
14      begin
15         OldValue <= 64'ha0000000;
16      end
17    endmodule
18
```

Appendix D. VHDL code for PC

```
1
2   module mem32(clk, mem_read, mem_write, address, data_in, data_out);
3      input clk, mem_read, mem_write;
4      input [63:0] address, data_in;
5      output [63:0] data_out;
6      reg [63:0] data_out;
7
8      parameter BASE_ADDRESS = 25'd0; // address that applies to this memory - change if desired
9
10     reg [63:0] mem_array [0:63];
11     wire [9:0] mem_offset;
12     wire address_select;
13
14     assign mem_offset = address[12:3];  // drop 3 LSBs to get doubleword offset
15     assign address_select = (address[63:13] == BASE_ADDRESS);  // address decoding
16
17     always @(mem_read or address_select or mem_offset or mem_array[mem_offset])
18     begin
19        if (mem_read == 1'b1 && address_select == 1'b1)
20        begin
21           if ((address % 8) != 0)
22              $display($time, " rom32 error: unaligned address %d", address);
23           data_out = mem_array[mem_offset];
24           $display($time, " reading data: Mem[%h] => %h", address, data_out);
25        end
26        else data_out = 64'hxxxxxxxxxxxxxxxx;
27     end
28
29     // for WRITE operations
30     always @(posedge clk)
31     begin
32        if (mem_write == 1'b1 && address_select == 1'b1)
33        begin
34        $display($time, " writing data: Mem[%h] <= %h", address,data_in);
35        mem_array[mem_offset] <= data_in;
36        end
37     end
38
39     // initialize with some arbitrary values
40     integer i;
41     initial
42     begin
43        for (i=0; i<7; i=i+1) mem_array[i] = i;
44     end
45  endmodule
```

Appendix E. VHDL code for mem32

```
1
2  module reg_file(clk, RegWrite, RN1, RN2, WN, RD1, RD2, WD);
3     input clk;
4     input RegWrite;
5     input [4:0] RN1, RN2, WN;
6     input [63:0] WD;
7     output [63:0] RD1, RD2;
8
9     reg [63:0] RD1, RD2;
10    reg [63:0] ARRAY[0:63]; //64 64-bit wide register?
11
12    // fill in the missing code
13    always @(negedge clk)
14    begin
15       if (RegWrite==1)
16       begin
17          ARRAY[WN]<=WD;
18       end
19    end
20
21    always @(posedge clk)
22    begin
23       RD1 <= ARRAY[RN1];
24       RD2 <= ARRAY[RN2];
25    end
26    // fill in the missing code
27 endmodule
```

Appendix F. VHDL code for reg_file

```
1  module adder(PCcurrent, offset, PCnew);
2
3     input [63:0] PCcurrent;
4     input [63:0] offset;
5     output [63:0] PCnew;
6
7     reg [63:0] PCnew;
8
9     always @(PCcurrent)
10    begin
11       PCnew <= PCcurrent + offset;
12    end
13 endmodule
14
```

Appendix G. VHDL code for adder

```
1   module binary_constant(constant);
2
3       output [63:0] constant;
4
5       reg [63:0] constant;
6
7       parameter const = 3'b100;
8
9       always
10
11      begin
12          constant[2:0] <= const;
13          constant[63:3] <= {61'b0};
14      end
15
16  endmodule
```

Appendix H. VHDL code for constant