Computer Architecture (EEE-3530)

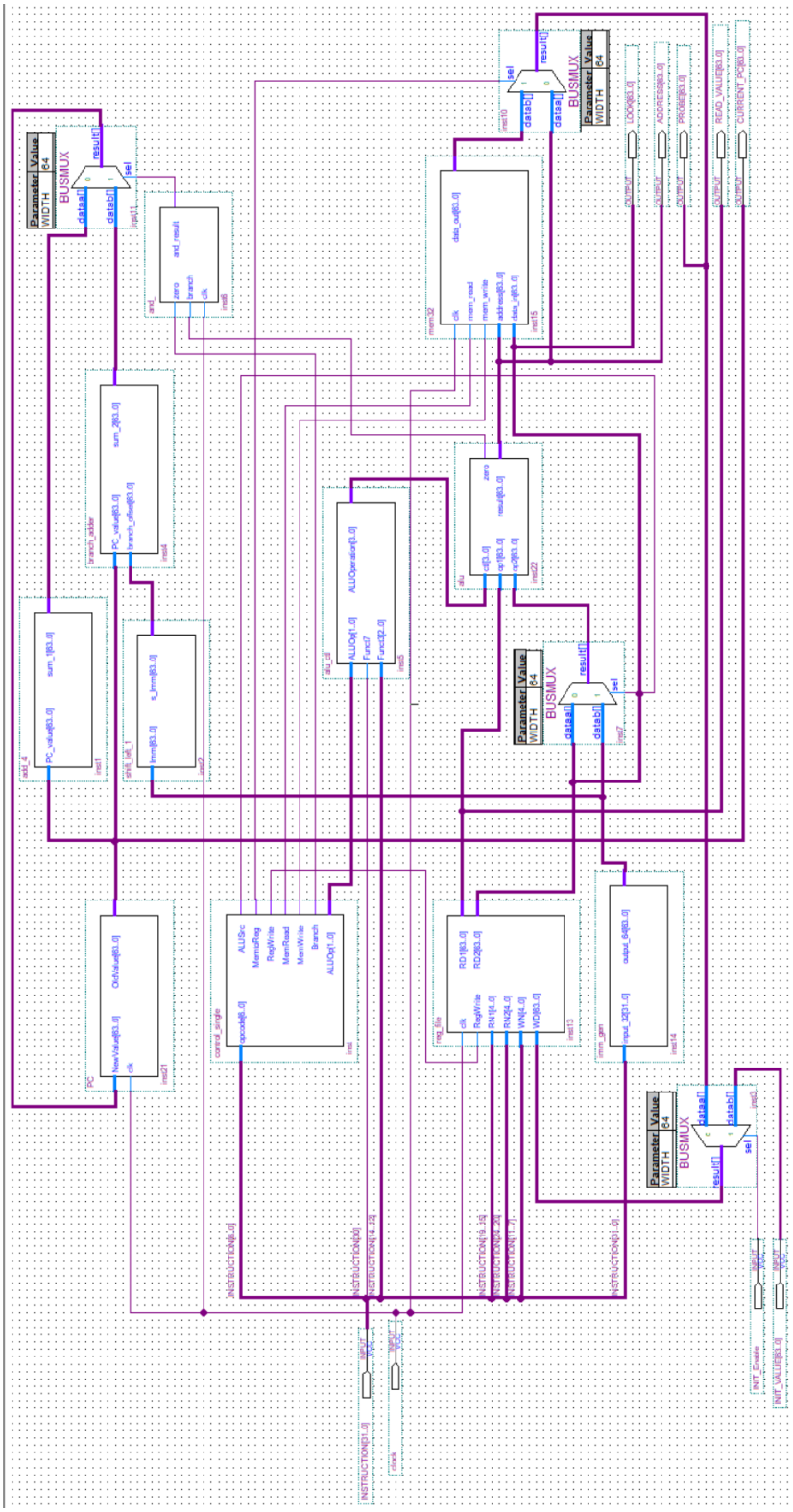
Project #2

2018142059

김서영

Submission date: 2020.06.01

0. Schematic design of top module



*Due to the operational error while executing branch instruction, I added 'clock' signal into those two parts.

1. Verilog HDL code

- **PC.v**

```
module PC (NewValue, OldValue, clk);
	input [63:0] NewValue;
	input clk;
	output [63:0] OldValue;
	reg [63:0] OldValue;

	always@(posedge clk)
	begin
		OldValue <= NewValue;
	end

	initial
	begin
		OldValue <= 64'ha0000000;
	end
endmodule
```

In real RISC-V processor, there is instruction memory to get instruction corresponding to program counter (PC) value. However, in the project, we do not use PC value to fetch instructions but just to check the value changes as the program executed including branch instruction.

'PC.v' part gets its new value and replace old pc value at the rising edge of clock. It is initialized to A0000000.

- **reg_file.v**

```
module reg_file(clk, RegWrite, RN1, RN2, WN, RD1, RD2, WD);
	input clk;
	input RegWrite;
	input [4:0] RN1, RN2, WN;
	input [63:0] WD;
	output [63:0] RD1, RD2;

	reg [63:0] RD1, RD2;
	reg [63:0] ARRAY[0:63];

	// fill in the missing code
```

```
        always @ (posedge clk)
        begin
                RD1<=ARRAY[RN1];
                RD2<=ARRAY[RN2];
        end

        always @ (negedge clk)
        begin
                if(RegWrite==1'b1)
                begin
                        ARRAY[WN]<=WD;
                end
        end
        // fill in the missing code
endmodule
```

The reg_file part takes instruction into 3 part – Read register 1 (RN1), Read register 2 (RN2), and Write Register (WN). It also gets Write data (WD) from the MemtoReg MUX, and RegWrite control signal for register write process.

At the rising edge of the clock, it reads value from register according to RN1 and RN2, then convey them as outputs. Since all the instruction used in RISC-V processor has similar location in instruction field for similar instruction part, the reg_file part can take its inputs just by separating instruction, conveniently.

Some error occurred when the operation executed in the program, since the data change occurs with the rising edge of the clock. Therefore, in the code, when the control signal RegWrite is 1, the part writes data from MemtoReg MUX on the register corresponding to WN, at the falling edge of the clock, inevitably.

- **control_single.v**

```
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'b1100011;
        //parameter ADDi     = 7'b0010011;
```

```
always @(opcode)
begin
        case (opcode)
        // fill in the misisng code

        R_FORMAT:
        begin
                ALUSrc<=1'b0;
                MemtoReg<=1'b0;
                RegWrite<=1'b1;
                MemRead<=1'b0;
                MemWrite<=1'b0;
                Branch<=1'b0;
                ALUOp<=2'b10;
        end

        LD:
        begin
                ALUSrc<=1'b1;
                MemtoReg<=1'b1;
                RegWrite<=1'b1;
                MemRead<=1'b1;
                MemWrite<=1'b0;
                Branch<=1'b0;
                ALUOp<=2'b00;
        end

        SD:
        begin
                ALUSrc<=1'b1;
                MemtoReg<=1'bx;
                RegWrite<=1'b0;
                MemRead<=1'b0;
                MemWrite<=1'b1;
                Branch<=1'b0;
                ALUOp<=2'b00;
        end

        BEQ:
        begin
                ALUSrc<=1'b0;
                MemtoReg<=1'bx;
                RegWrite<=1'b0;
                MemRead<=1'b0;
                MemWrite<=1'b0;
```

```
                                Branch<=1'b1;
                                ALUOp<=2'b01;
                        end
                        // fill in the missing code
                        endcase
        end
endmodule
```

The control_single part gets the opcode part of the instruction (instruction [6:0]) to make 7 signals to control the other parts in the processor. Since the opcode differs for type of instruction, the part compares opcode input with the sample opcode parameter it already has, then passes 7 signals – ALUSrc, MemtoReg, RegWrite, MemRead, Memwrite, Branch, and ALUop – to the appropriate parts.

- **alu_ctl.v**

```
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 F3_add    = 3'b000;
        parameter F3_sub    = 3'b000;
        parameter F3_and    = 3'b111;
        parameter F3_or     = 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;
        always @(ALUOp or Funct7 or Funct3)

        begin
                case (ALUOp)
                // fill in the missing code
```

```
                    2'b10:
                            begin

                            if ((Funct3==F3_add) && (Funct7==F7_add))
                            begin
                            ALUOperation<=ALU_add;
                            end

                            else if ((Funct3==F3_sub) && (Funct7==F7_sub))
                            begin
                            ALUOperation<=ALU_sub;
                            end

                            else if ((Funct3==F3_and) && (Funct7==F7_and))
                            begin
                            ALUOperation<=ALU_and;
                            end

                            else if ((Funct3==F3_or) && (Funct7==F7_or))
                            begin
                            ALUOperation<=ALU_or;
                            end
                            end

                    2'b00:
                    begin
                            ALUOperation<=ALU_add;
                    end

                    2'b01:
                    begin
                            ALUOperation<=ALU_sub;
                    end
                    // fill in the missing code
                    endcase
            end
endmodule
```

The alu_ctl part get input signal to specify which kind of instruction is executed now, then passes appropriate control signal 'ALUOperation', which orders appropriate calculation with 3 bits, to ALU in the processor.

- **alu.v**

```
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;

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

                ALU_add:
                begin
                        result<=(op1+op2);
                end

                ALU_sub:
                begin
                        result<=(op1-op2);
                end

                ALU_and:
                begin
                        result<=(op1&op2);
                end

                ALU_or:
                begin
                        result<=(op1|op2);
                end
```

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

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

The alu part calculates its data inputs op1 and op2 according to the control signal from ALU control part. Also, after the calculation it checks whether the result is zero – whether the two data (op1 and op2) has same value, then send the Boolean result for beq instruction.

- **imm_gen.v**

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

        // fill in the missing code

        wire [31:0] input_32;
        reg [63:0] output_64;

        parameter R_FORMAT = 7'b0110011;
        parameter LD       = 7'b0000011;
        parameter SD       = 7'b0100011;
        parameter BEQ      = 7'b1100011;

        always @(input_32)
        begin

        case (input_32[6:0])
        LD:
        begin
        output_64<={{52{input_32[31]}},input_32[31:20]};
        end
```

```
            SD:
            begin
            output_64<={{52{input_32[31]}},input_32[31:25],input_32[11:7]};
            end

            BEQ:
            begin
            output_64<={{53{input_32[31]}},input_32[7],input_32[30:25],input_32[11:8]};
            end

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

The imm_gen part gets all the instruction sentence, then get immediate value according to the opcode (instruction [6:0]). Since load double, save double, and branch if equal instructions has different way to represent its immediate value, the Immediate Generator part checks opcode of the instruction, and sort them.

- **add_4.v**

```
module add_4(PC_value,sum_1);

            input [63:0] PC_value;
            output [63:0] sum_1;

            reg [63:0] sum_1;

            always @ (PC_value)
            begin
                        sum_1<=(PC_value+64'b100);
            end

endmodule
```

If the branch instruction taken, or more often if the instruction is not branch instruction, new PC value has value of current PC value+4. Therefore, add_4 part add 4 to current PC value when current PC value is updated.

- **shift_left_1.v**

```
module shift_left_1(Imm, s_Imm, clk);

        input [63:0] Imm;
        input clk;
        output [63:0] s_Imm;

        reg [63:0] s_Imm;
        reg [64:0] imm_temp;

        always @(negedge clk)
        begin
                s_Imm<=(Imm<<<1);
        end
endmodule
```

12-bit field of branch instruction becomes halfword address when it multiplied by 2. The calculation could be easily done by shifting one bit to left (put 0 on the right side).
Since there occurs an error when the branch is taken when the PC value changes at rising edge of each clock, in the code, the operation occurs when the falling edge of the clock.

- **branch_adder.v**

```
module branch_adder(PC_value,branch_offset,sum_2);

        input [63:0] PC_value, branch_offset;
        output [63:0] sum_2;

        wire [63:0] branch_offset;
        reg [63:0] temp_sum;
        reg [63:0] sum_2;

        always @(branch_offset)
        begin
                $unsigned(temp_sum);
                $unsigned(sum_2);
                temp_sum<=$unsigned(PC_value)+$signed(branch_offset);
                sum_2=temp_sum;
        end
endmodule
```

Branch addressing is PC-related addressing; Therefore, it should find its target address based on current PC address. The branch adder part adds current PC address and result from shift_left part to get the target address.

- **and_.v**

```
module and_(zero,branch,and_result,clk);

        input zero, branch;
        input clk;
        output and_result;

        reg and_result;

        always @(negedge clk)

        begin
                and_result<=(zero && branch);
        end
endmodule
```

To determine new PC value between increased PC value (current PC value+4) and branch target, and_ part get 'zero' control signal from ALU and 'branch' control signal from the control_single part.

In the Verilog HDL code, there was a problem that the signal changing in and_ part and rising clock occurred at the same time, which makes the PC part made inaccurate operation.* Therefore, in the code, I put clock signal to do the operation at the falling edge of the signal.

* Since there was same problem in mem_32 part during save double (SD) instruction execution, I corrected the code 'always@(posedge clk)' in line 30 to 'always @(negedge clk)'.

2. Waveform

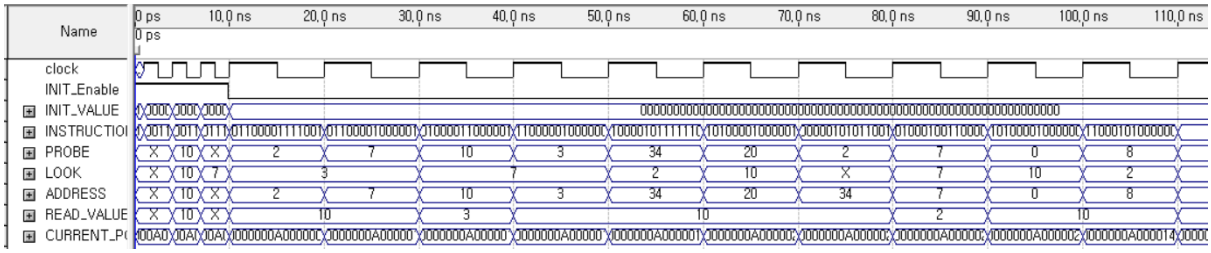
- instruction

	Instruction	Instruction in machine code	Register value
0	(X1 initialize)		X1=10
	(X2 initialize)		X2=7
	(X3 initialize)		X3=3
1	and X4, X1, X3	0000000/00011/00001/111/00100/0110011	X1=10, X3=3, X4=2
2	sub X6, X1, X3	0100000/00011/00001/000/00110/0110011	X1=10, X3=3, X6=7
3	add X5, X3, X2	0000000/00010/00011/000/00101/0110011	X2=7, X3=3, X5=10

4	beq X1, X6 0x440	0001001/00110/00001/000/00000/1100011	(X1=10)≠(X6=7) 0x440=288=100100000_2
5	sd X4, 24(X5)	0000000/00100/00101/111/11000/0100011	X4=2, X5=10
6	add X6, X1, X5	0000000/00101/00001/000/00110/0000011	X1=10, X5=10, X6=20
7	ld X6, 24(X5)	0000000/11000/00101/011/00110/0000011	X5=10, X6=2
8	or X2, X4, X2	0000000/00010/00100/110/00010/0110011	X2=7, X4=2
9	beq X1, X5, 0x440	0001001/00101/00001/000/00000/1100011	(X1=10) = (X5=10) 0x440=288=100100000_2
10	sub X1, X5, X6	0100000/00110/00101/000/00001/0110011	X1=8, X5=10, X6=2

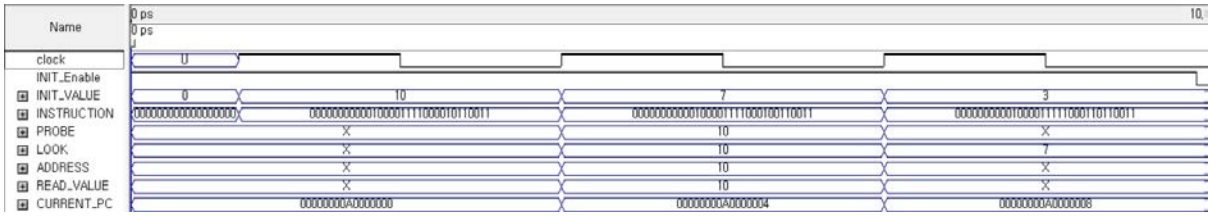
- whole waveform result

*To increase the readability of the waveform, some values are showed in decimal and hexadecimal form, just in the report.



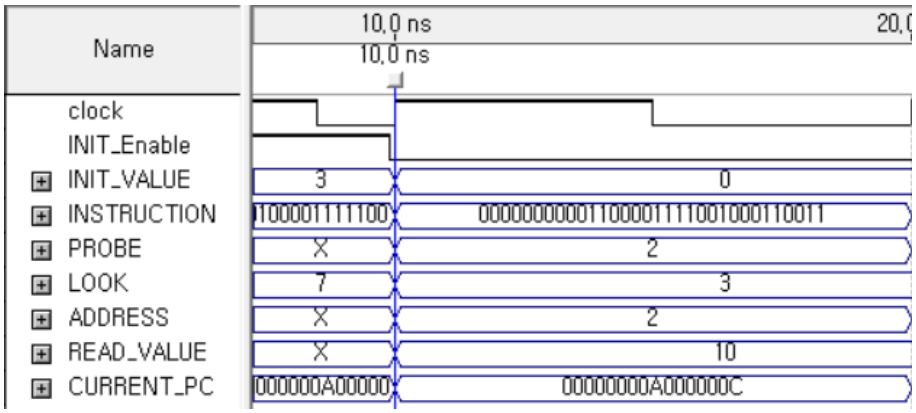
.

◎ initialization



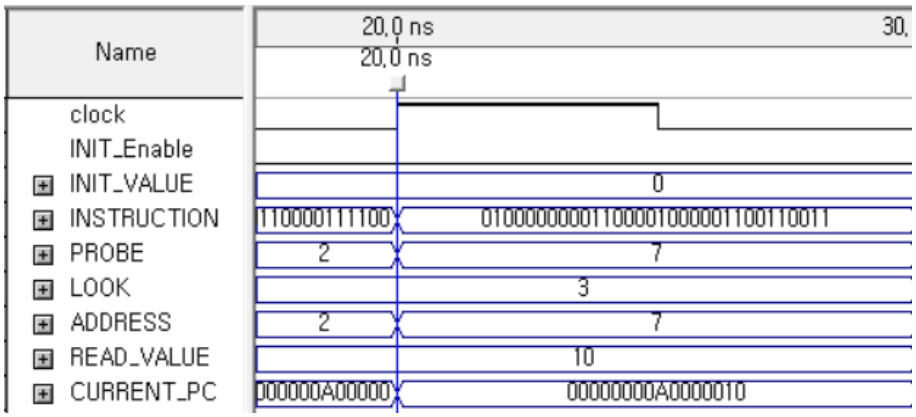
From 0ns to 10ns, the waveform initializes the register X1, X2, and X3. Then the given instruction starts from 10ns, each instruction takes 10 ns. While initialization, the PC becomes A0000008.

① and X4, X1, X3



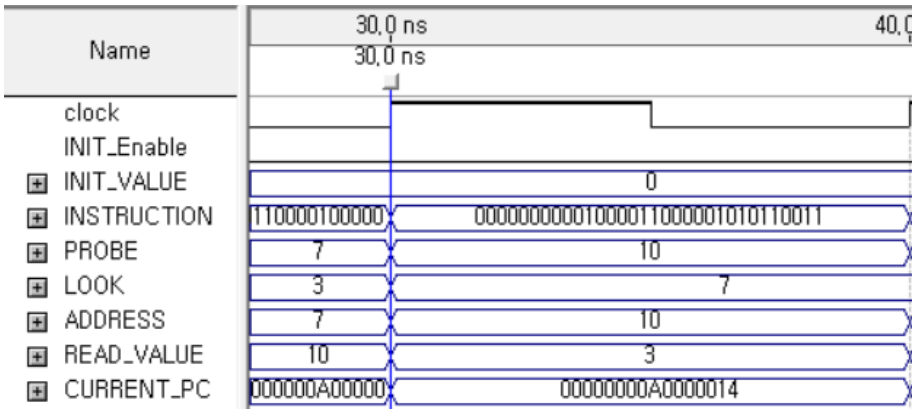
First instruction given was and instruction. Since X1=10=1010_2 and X3=3=11_2, the calculation result which should be the value of X4 is 10_2=2. Therefore, Probe and Address output value are 2, Read value which is the value of read register 1 is X1=10, and LOOK got the value of X3=3. The PC gets A0000008+4=A000000C.

② sub X6, X1, X3



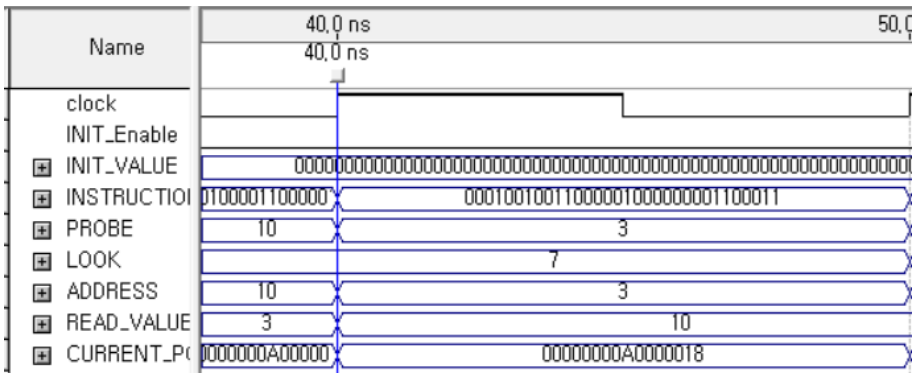
Second instruction was sub instruction. Since X1 and X3 values did not change while the previous instruction execution, Probe and Read_Value has same output. X1-X3=10-3=7 should be the value of X6, which is shown by 'Probe' and 'Address' output. The PC value become A0000010, which is A000000C+4.

③ add X5, X3, X2



The third instruction is 'add' instruction. X2 and X3 still have their initial value, 7 and 3. Those values are shown in 'Look' and 'Read_Value' outputs. Then the rd register X5 should get the value 10, as shown in Probe and Address outputs. The PC value increases about 4, becoming A0000014.

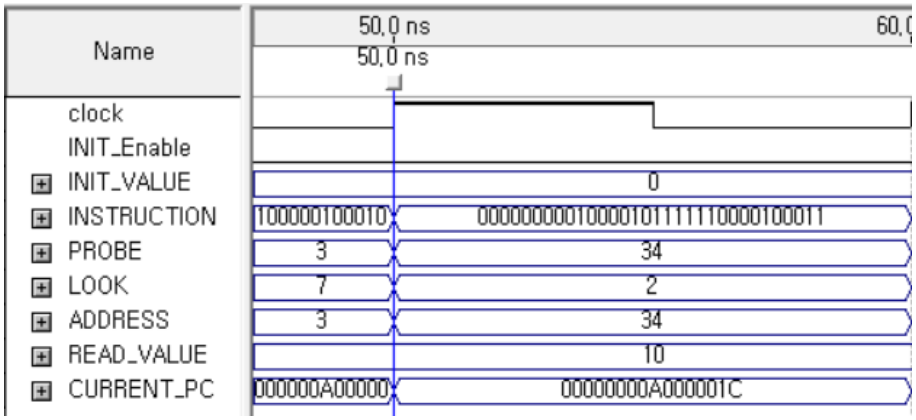
④ beq X1, X6 0x440



The branch instruction is given, therefore the ALU would compare the value of registers X1 and X6. X1 has value of 10 and X6 has value of 7, each shown in Read_Value and Look output. ALU compare the values by subtracting them and check whether the value is 0. Since X1-X6=10-7=3, which is not 0, the branch did 'not taken'. The subtraction result is shown in 'Probe' and 'Address'. The PC value was just added 4.

Since the 'MemtoReg' control signal has 'don't care' state, it does not change its value '0' from its previous instruction. Therefore, the value of 'Probe' output is just same as that of 'Address' output.

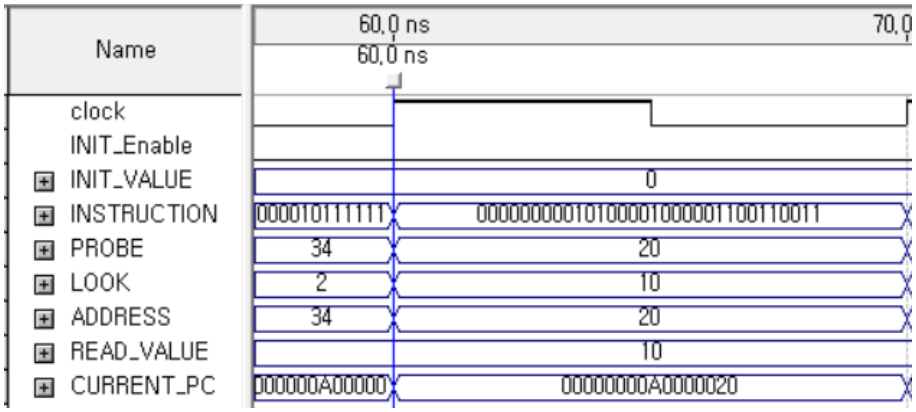
⑤ sd X4, 24(X5)



Save Doubleword (SD) instruction gives two register number to get value, RN1 value to save in memory and RN2 value to get memory address. In the waveform, RN1, X5 value is Read_Value, 10, and RN2, X4 value is 2 in the Look output. The address for the memory should be add with the immediate value in ALU part, and it appears to be 34 at the 'Address' output.

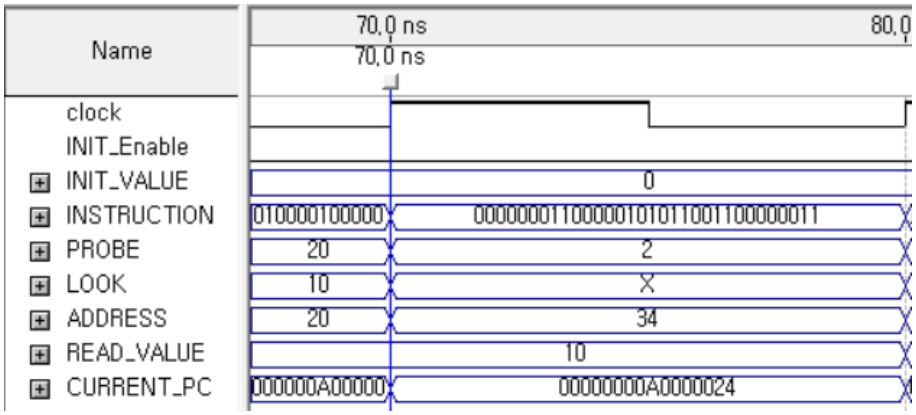
Still the 'MemtoReg' control signal has 'don't care' state, 'Probe' output has the same value as that of 'Address' output. Since the branch did 'not taken' at the previous 'beq' instruction, the PC value increased about 4, becoming A000001C.

⑥ add X6, X1, X5



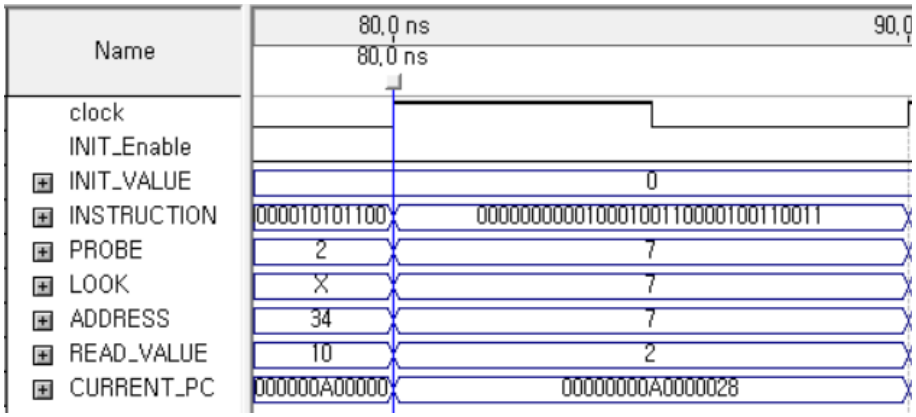
6th instruction is 'add' instruction, and both value of RN1 (X1) and RN2 (X5) is 10, and they appeard in 'Read_Value' and 'Look' value. The calculation result from ALU is 20, which appeared in 'Address' output, and the value is sent to Register, which appeared in 'Probe' output. The PC value is A0000020, which increased about 4 from previous value.

⑦ ld X6, 24(X5)



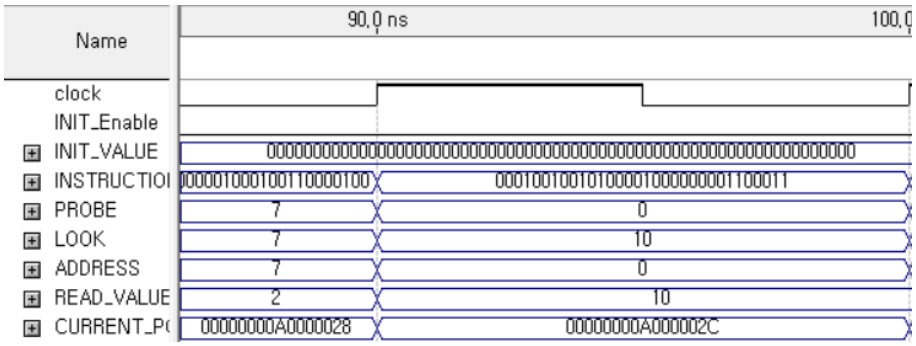
Load Doubleword (LD) instruction gets value from memory and save it at the register. Sum of the offset and value of X5 is address in memory. The value of X5 is shown in 'Read Value' output, and the sum is appeared in 'Address' value. Since the instruction does not directs is RN2 value, the 'Look' output is not determined. Since the value of designated memory is saved as 2 at 5th instruction (SD), the value is loaded and shown in the 'Probe' output. Then the value saved at register X6, WN in the register part. The PC increased about 4, becoming A0000024.

⑧ or X2, X4, X2



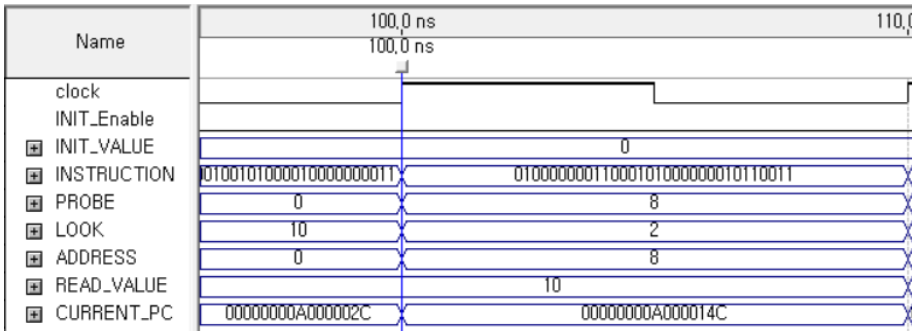
'or' instruction gets two register value and do logic operation on them. RN1, X4 has value of $2=10_2$, and RN2, X2 has value of $7=111_2$, as shown in 'Read Value' and 'Probe' output. The operation result is 111_2 and we can check it on 'Address' output. Then the value sent to the Register part, which is appeared on 'Probe' output. The PC value increased about 4, becoming A0000028.

⑨ beq X1, X5, 0x440



Another 'branch if equal' instruction is given. RN1 gets value 10 from register X1 and RN2 gets value from register X5, which is also 10. Each value is shown on 'Read_Value' and 'Look' outputs. The ALU compares those value by subtracting them and found out the result is 'zero', as appeared in 'Address' output. Since MemtoReg signal is 'don't care', it did not change from the previous clock cycle. As a result, the 'Probe' output has the same value 0 as 'Address'. The PC value get increased about 4, which is A000002C.

⑩ sub X1, X5, X6



Last instruction is 'sub' instruction. The value of RN1, X5 is 10, and value of RN2, X6 is 2. Those values can be checked in 'Read_Value' and 'Look' outputs. The calculation result is 10-2=8, and the value is shown in 'Address'. The result value sent to the register part, and it is shown in 'Probe' part. Since the branch instruction has taken on the previous operation, the 'Current PC' Value is A00014C.

3. Discussion

In the project, all the simulation outcome came out as expected. However, some errored occurred when the data changes at the rising edge of the clock. To prevent the error, I changed the problematic operations to be executed at the falling edge inevitably.

While doing the project, I could learn some basic Verilog code for the first time. Also, doing simulation helped me to understand the operation of RISC-V processor.

4. Reference

Computer Organization and Design RISC-V Edition, David A. Patterson