Computer Architecture (EEE-3530)

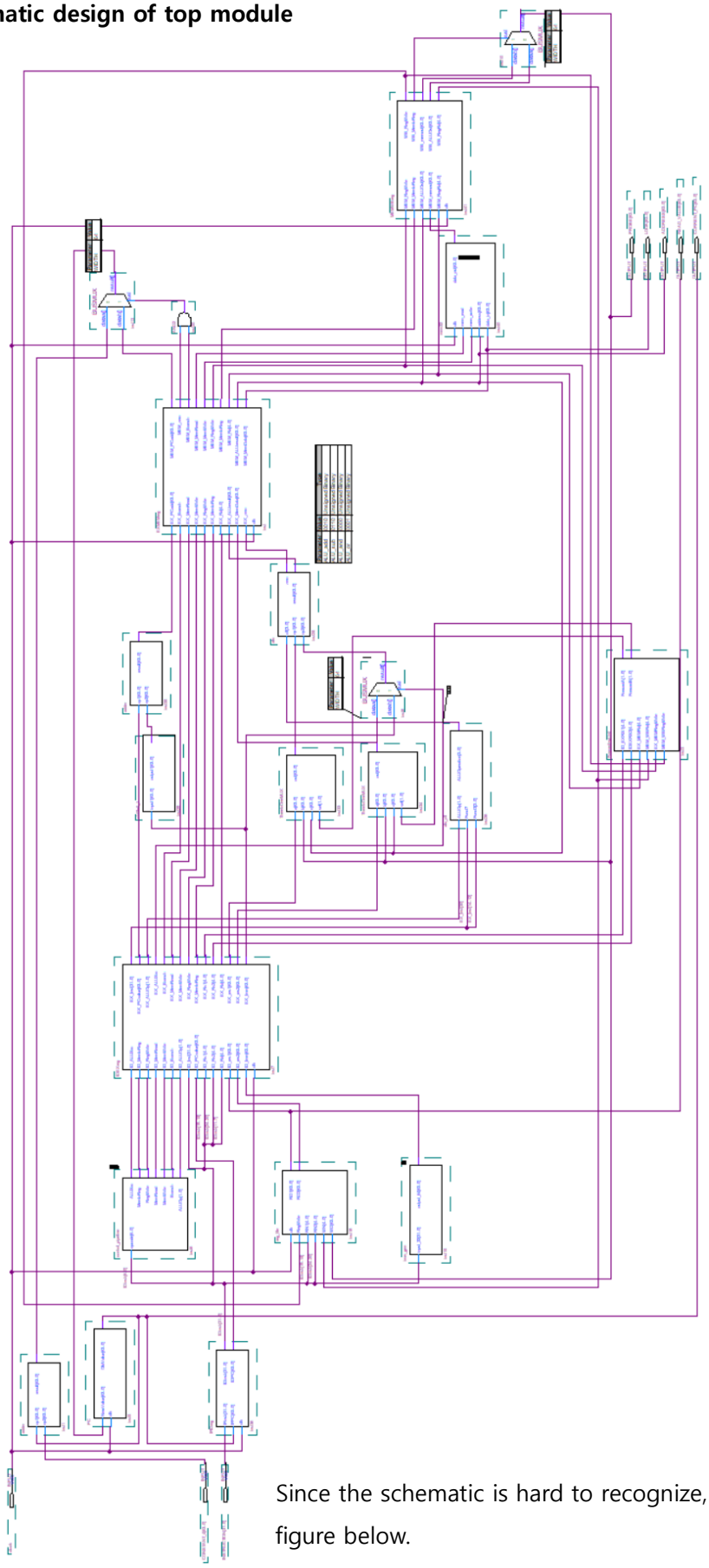
Project #3

2018142059

김서영

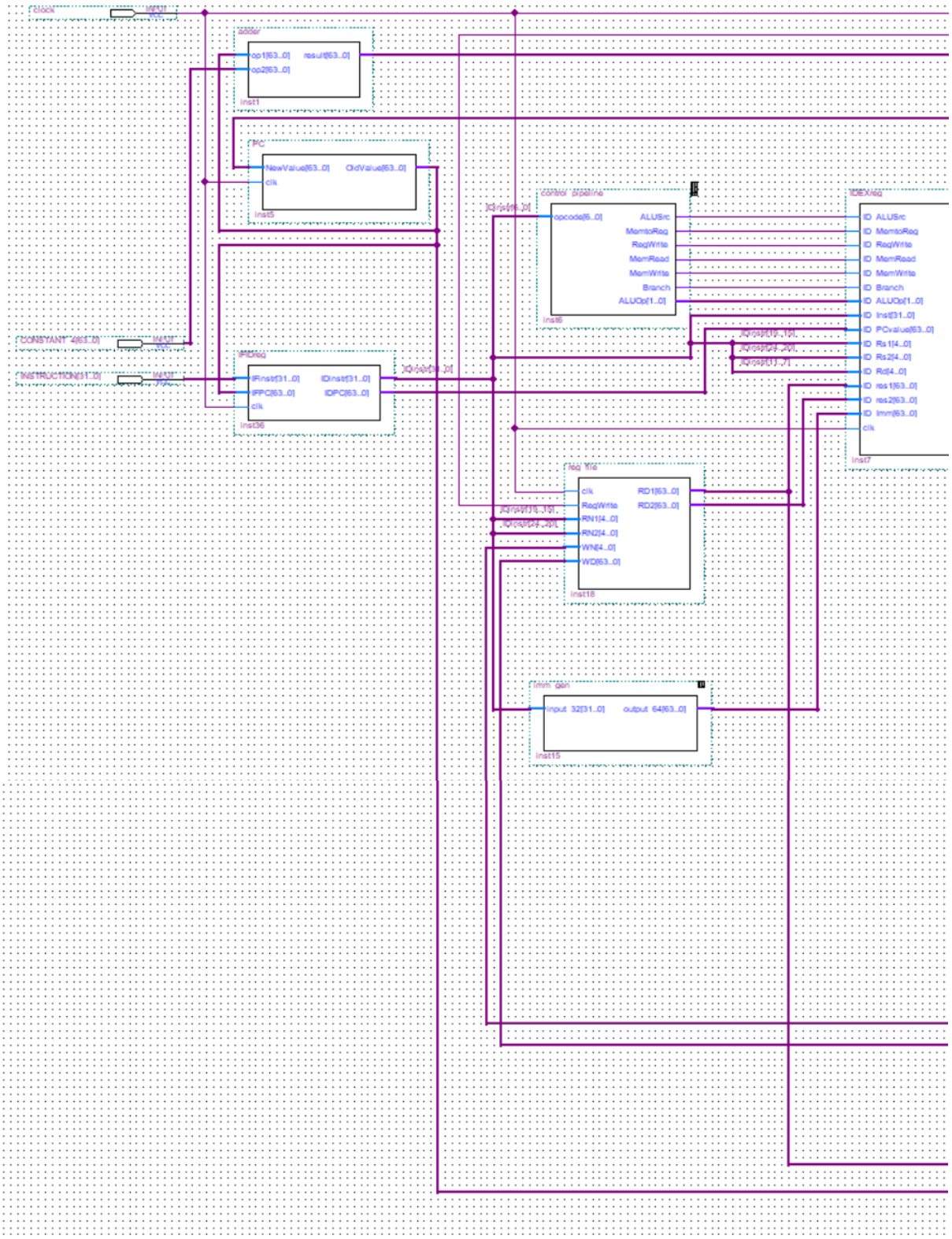
Submission date: 20.06.21.

0. Schematic design of top module

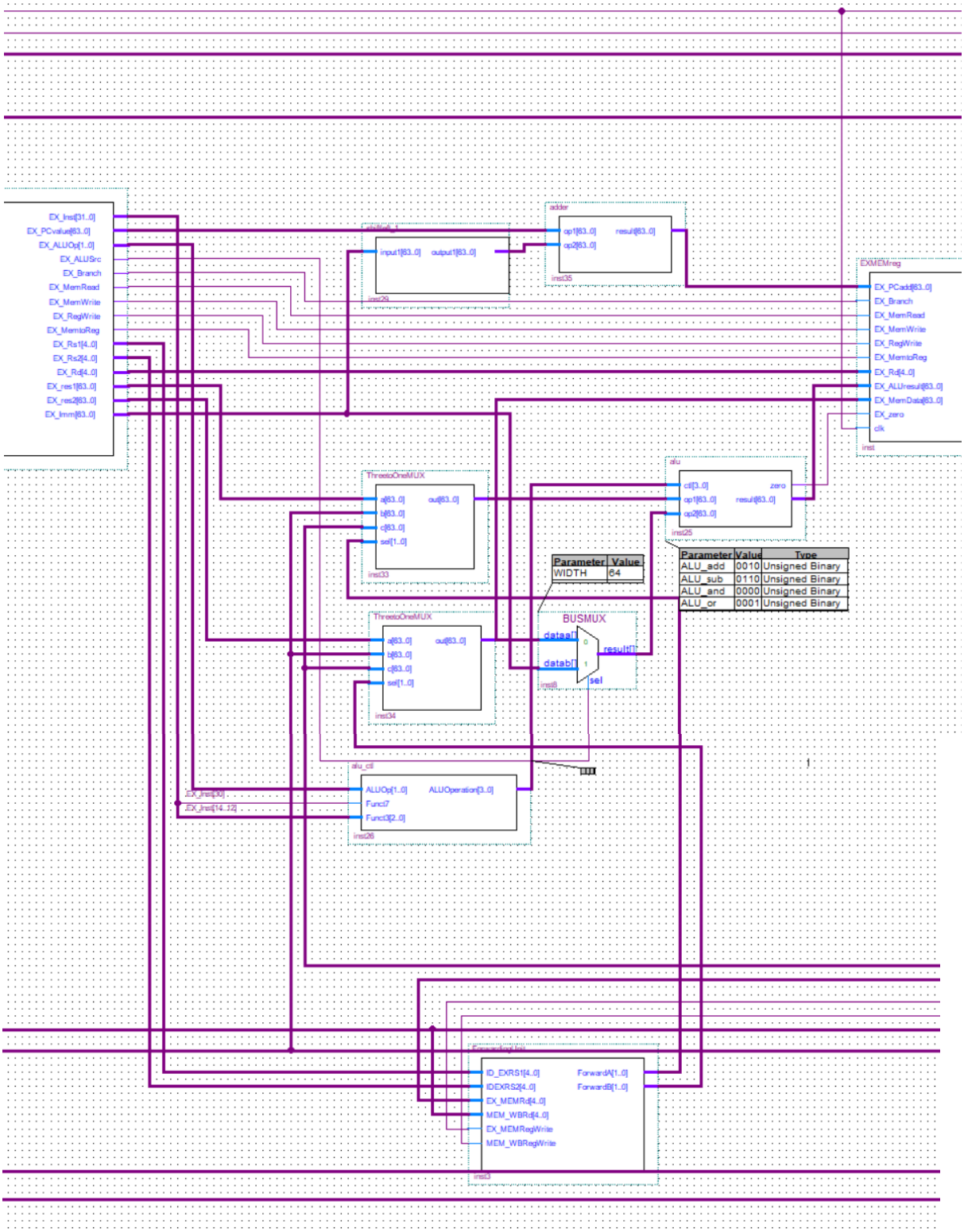


Since the schematic is hard to recognize, I put some partial figure below.

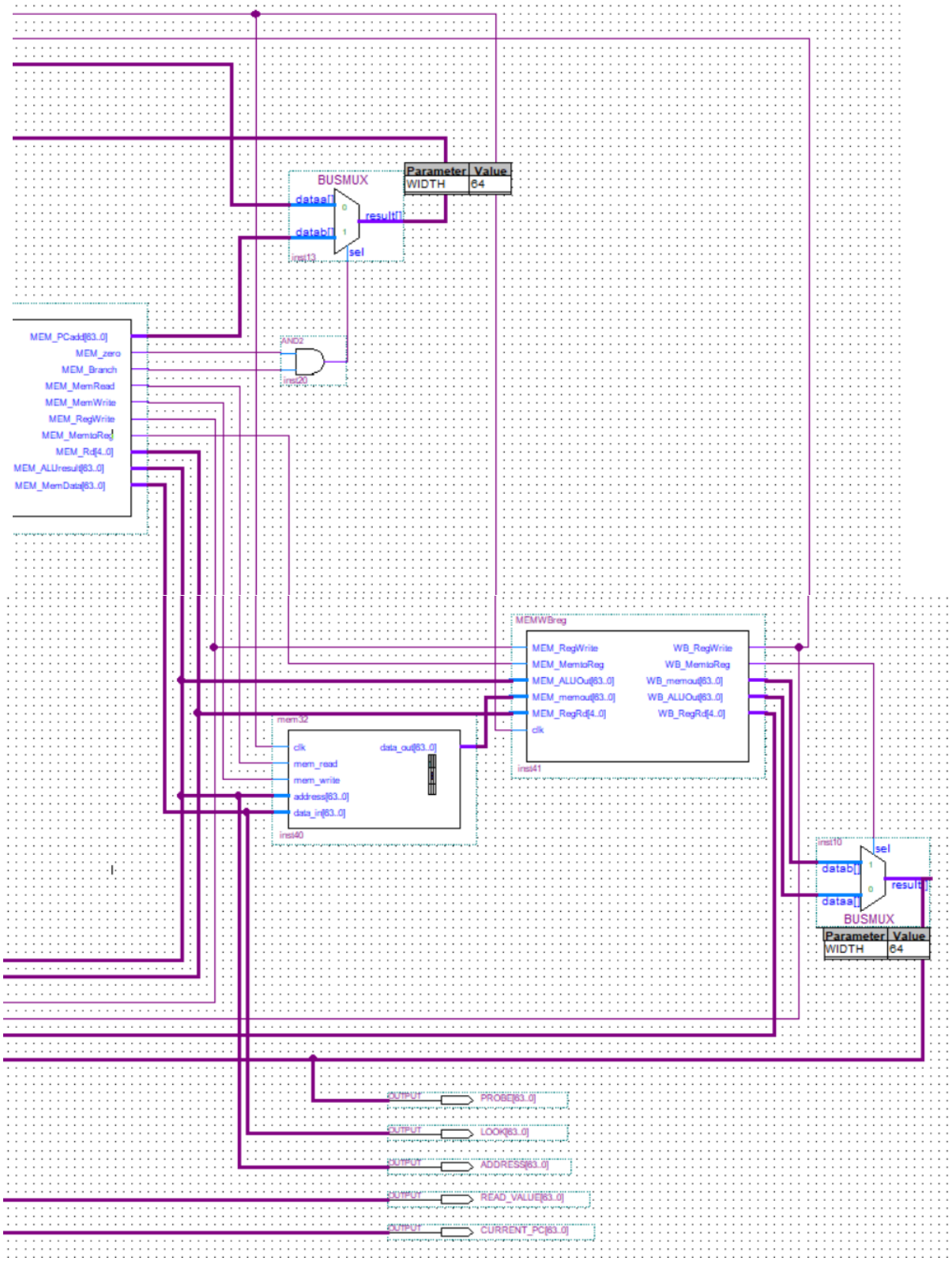
① IF, ID stage


clock
adder
op1[63..0]
op2[63..0]
result[63..0]
inst1
PC
NewValue[63..0]
OldValue[63..0]
clk
inst5
CONSTANT 4[63..0]
INSTRUCTION[31..0]
IFID reg
IFInst[31..0]
IFPC[63..0]
IDInst[31..0]
IDPC[63..0]
clk
inst36
control pipeline
opcode[6..0]
ALUSrc
MemtoReg
RegWrite
MemRead
MemWrite
Branch
ALUOp[1..0]
inst6
ID/EX reg
ID ALUSrc
ID MemtoReg
ID RegWrite
ID MemRead
ID MemWrite
ID Branch
ID ALUOp[1..0]
ID Inst[31..0]
ID PCvalue[63..0]
ID Rs1[4..0]
ID Rs2[4..0]
ID Rd[4..0]
ID rs1[63..0]
ID rs2[63..0]
ID Imm[63..0]
clk
inst7
reg file
clk
RegWrite
RN1[4..0]
RN2[4..0]
WN[4..0]
WD[63..0]
RD1[63..0]
RD2[63..0]
inst18
Imm Gen
input 32[31..0]
output 64[63..0]
inst15

② EX stage


EX_Inst[31..0]
EX_PCvalue[63..0]
EX_ALUOp[1..0]
EX_ALUSrc
EX_Branch
EX_MemRead
EX_MemWrite
EX_RegWrite
EX_MemtoReg
EX_Rs1[4..0]
EX_Rs2[4..0]
EX_Rd[4..0]
EX_res1[63..0]
EX_res2[63..0]
EX_Imm[63..0]
adder
op1[63..0]
op2[63..0]
result[63..0]
inst36
input1[63..0]
output1[63..0]
inst29
EXMEMreg
EX_PCadd[63..0]
EX_Branch
EX_MemRead
EX_MemWrite
EX_RegWrite
EX_MemtoReg
EX_Rd[4..0]
EX_ALUresult[63..0]
EX_MemData[63..0]
EX_zero
clk
inst
ThreetoOneMUX
a[63..0]
b[63..0]
c[63..0]
sel[1..0]
out[63..0]
inst33
ThreetoOneMUX
a[63..0]
b[63..0]
c[63..0]
sel[1..0]
out[63..0]
inst34
alu
ctl[3..0]
op1[63..0]
op2[63..0]
zero
result[63..0]
inst25
Parameter Value
WIDTH 64
BUSMUX
dataa[]
datab[]
sel
result[]
inst8
Parameter Value Type
ALU_add 0010 Unsigned Binary
ALU_sub 0110 Unsigned Binary
ALU_and 0000 Unsigned Binary
ALU_or 0001 Unsigned Binary
alu_ctl
ALUOp[1..0]
Funct7
Funct3[2..0]
ALUOperation[3..0]
inst26
EX_Inst[30]
EX_Inst[14..12]
ID_EXRS1[4..0]
IDEXRS2[4..0]
EX_MEMRd[4..0]
MEM_WBRd[4..0]
EX_MEMRegWrite
MEM_WBRegWrite
ForwardA[1..0]
ForwardB[1..0]
inst3

③ MEM/WB stage



1. Verilog HDL code

- control_pipeline.v

```
module control_pipeline(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;

        initial
        begin
        ALUSrc<=0;MemtoReg<=0;RegWrite<=0;MemRead<=0;MemWrite<=0;Branch<=0;ALUOp<=2'b00;
        end

        always @(opcode)
        begin
                // fill in the misisng code
                case(opcode)
                        R_FORMAT:
                        begin
                                ALUSrc<=0;
                                MemtoReg<=0;
                                RegWrite<=1;
                                MemRead<=0;
                                MemWrite<=0;
                                Branch<=0;
                                ALUOp<=2'b10;
                        end

                        LD:
                        begin
                                ALUSrc<=1;
                                MemtoReg<=1;
                                RegWrite<=1;
                                MemRead<=1;
                                MemWrite<=0;
                                Branch<=0;
                                ALUOp<=2'b00;
                        end
```

```
                SD:
                begin
                        ALUSrc<=1;
                        RegWrite<=0;
                        MemRead<=0;
                        MemWrite<=1;
                        Branch<=0;
                        MemtoReg=1'bx;
                        ALUOp<=2'b00;
                end

                BEQ:
                begin
                        ALUSrc<=0;
                        RegWrite<=0;
                        MemRead<=0;
                        MemWrite<=0;
                        Branch<=1;
                        MemtoReg=1'bx;
                        ALUOp<=2'b01;
                end

                default
                begin
                        ALUSrc<=0;
                        RegWrite<=0;
                        MemRead<=0;
                        MemWrite<=0;
                        Branch<=1'b0;
                        ALUOp<=2'b00;
                end
        endcase
        // fill in the misisng code
    end
endmodule
```

The control part of RISC-V gets 7 bit opcode and compare them with the saved opcodes to generate signals for other parts. To implement pipelined system, those control signals are passed to the pipeline registers.

- 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];

	initial
	begin
		ARRAY[0] <= 64'd0; ARRAY[1] <= 64'd24; ARRAY[2] <= 64'd20; ARRAY[3] <= 64'd14;
		ARRAY[4] <= 64'd8; ARRAY[5] <= 64'd35; ARRAY[6] <= 64'd2; ARRAY[7] <= 64'd0;
		ARRAY[8] <= 64'd0; ARRAY[9] <= 64'd0; ARRAY[10] <= 64'd0; ARRAY[11] <= 64'd0;
		ARRAY[12] <= 64'd0; ARRAY[13] <= 64'd0; ARRAY[14] <= 64'd0; ARRAY[15] <= 64'd0;
		ARRAY[16] <= 64'd0; ARRAY[17] <= 64'd0; ARRAY[18] <= 64'd0; ARRAY[19] <= 64'd0;
		ARRAY[20] <= 64'd0; ARRAY[21] <= 64'd0; ARRAY[22] <= 64'd0; ARRAY[23] <= 64'd0;
		ARRAY[24] <= 64'd0; ARRAY[25] <= 64'd0; ARRAY[26] <= 64'd0; ARRAY[27] <= 64'd0;
		ARRAY[28] <= 64'd0; ARRAY[29] <= 64'd0; ARRAY[30] <= 64'd0; ARRAY[31] <= 64'd0;
	end

	always@(negedge clk)
	begin
	if(RegWrite)
		begin
			ARRAY[WN] = WD;
		end

		RD1 <= ARRAY[RN1];
		RD2 <= ARRAY[RN2];
	end
endmodule
```

Register in RISC-V gets register number for RD1 and RD2 from IF/ID registers. Also, it gets data to write and the register number to write from WB stage. Every falling edge of clk, it gets and writes data. In pipelined system, the register should write the data first, then get the data. Therefore, I modified the given code so that if the RegWrite signal is 1, the register writes the WD data to WN register first with blocking '=', then gets data with nonblocking '<='.

- IDEXreg

```
module IDEXreg
        (ID_ALUSrc, ID_MemtoReg, ID_RegWrite, ID_MemRead, ID_MemWrite, ID_Branch, ID_ALUOp,
        ID_Inst, ID_PCvalue, ID_Rs1, ID_Rs2, ID_Rd, ID_res1, ID_res2, ID_Imm, clk,
        EX_Inst, EX_PCvalue, EX_ALUOp, EX_ALUSrc, EX_Branch, EX_MemRead, EX_MemWrite, EX_RegWrite,
        EX_MemtoReg, EX_Rs1, EX_Rs2, EX_Rd, EX_res1, EX_res2, EX_Imm);

        input [63:0] ID_PCvalue, ID_res1, ID_res2, ID_Imm;
        input [31:0] ID_Inst;
        input [4:0] ID_Rs1, ID_Rs2, ID_Rd;
        input ID_ALUSrc, ID_MemtoReg, ID_RegWrite,   ID_MemRead, ID_MemWrite, ID_Branch,   clk;
        input [1:0] ID_ALUOp;

        output [63:0] EX_PCvalue, EX_res1, EX_res2, EX_Imm;
        output [31:0] EX_Inst;
        output [4:0] EX_Rs1, EX_Rs2, EX_Rd;
        output [1:0] EX_ALUOp;
        output EX_ALUSrc, EX_Branch, EX_MemRead, EX_MemWrite, EX_RegWrite, EX_MemtoReg;

        reg [63:0] EX_PCvalue, EX_res1, EX_res2, EX_Imm;
        reg [31:0] EX_Inst;
        reg [4:0] EX_Rs1, EX_Rs2, EX_Rd;
        reg [1:0] EX_ALUOp;
        reg EX_ALUSrc, EX_Branch, EX_MemRead, EX_MemWrite, EX_RegWrite, EX_MemtoReg;

        always @(posedge clk)
                begin
                                EX_ALUOp<=ID_ALUOp;
                                EX_ALUSrc<=ID_ALUSrc;
                                EX_Branch<=ID_Branch;
                                EX_MemRead<=ID_MemRead;
                                EX_MemWrite<=ID_MemWrite;
                                EX_RegWrite<=ID_RegWrite;
                                EX_MemtoReg<=ID_MemtoReg;
                                EX_PCvalue<=ID_PCvalue;
                                EX_res1<=ID_res1;
                                EX_res2<=ID_res2;
                                EX_Imm<=ID_Imm;
                                EX_Inst<=ID_Inst;
                                EX_Rs1<=ID_Rs1;
                                EX_Rs2<=ID_Rs2;
                                EX_Rd<=ID_Rd;
                end
```

To maintain signals from every stage, registers are needed between every stage. The ID/EX register gets control signals, resister values, resister numbers, PC value, instruction, and immediate values from the preceding parts. Some of its signals are passed to the next pipeline register, and others get to the parts in EX stage, such as MUX, ALU, shift_left_1, and branch adder.

- EXMEMreg.v

```
module EXMEMreg
        (EX_PCadd, EX_Branch, EX_MemRead, EX_MemWrite, EX_RegWrite,EX_MemtoReg,
        EX_Rd, EX_ALUresult, EX_MemData, EX_zero, clk,
        MEM_PCadd, MEM_zero, MEM_Branch, MEM_MemRead, MEM_MemWrite, MEM_RegWrite,
        MEM_MemtoReg, MEM_Rd, MEM_ALUresult, MEM_MemData);

        input [63:0] EX_PCadd, EX_ALUresult, EX_MemData;
        input [4:0] EX_Rd;
        input EX_Branch, EX_MemRead, EX_MemWrite, EX_RegWrite, EX_MemtoReg, EX_zero, clk;

        output [63:0] MEM_PCadd, MEM_ALUresult, MEM_MemData;
        output [4:0] MEM_Rd;
        output MEM_Branch, MEM_MemRead, MEM_MemWrite, MEM_RegWrite, MEM_MemtoReg, MEM_zero;

        reg [63:0] MEM_PCadd, MEM_ALUresult, MEM_MemData;
        reg [4:0] MEM_Rd;
        reg MEM_Branch, MEM_MemRead, MEM_MemWrite, MEM_RegWrite, MEM_MemtoReg, MEM_zero;

        always @(posedge clk)
                begin
                        MEM_ALUresult<=EX_ALUresult;
                        MEM_MemData<=EX_MemData;
                        MEM_Rd<=EX_Rd;
                        MEM_Branch<=EX_Branch;
                        MEM_MemRead<=EX_MemRead;
                        MEM_MemWrite<=EX_MemWrite;
                        MEM_RegWrite<=EX_RegWrite;
                        MEM_MemtoReg<=EX_MemtoReg;
                        MEM_zero<=EX_zero;
                        MEM_PCadd<=EX_PCadd;
                end

        initial
                begin
                        MEM_Branch<=0;
                        MEM_MemRead<=0;
                        MEM_MemWrite<=0;
                        MEM_RegWrite<=0;
                        MEM_MemtoReg<=0;
                        MEM_zero<=0;
```

```
                    MEM_ALUresult<=0;
                    MEM_MemData<=0;
                    MEM_Rd<=0;
                    MEM_PCadd<=0;
                end
endmodule
```

The EX/MEM register gets control signals for MEM and WB stage, immediate-value-added-PC value, ALU result value, Data to be written on memory, and register number for WB stage. The control signals and data for WB stage just get to MEM/WB registers, and the others are passed to the data memory parts in MEM stage.

- ForwardingUnit.v

```
module ForwardingUnit
        (ID_EXRS1, IDEXRS2, EX_MEMRd, MEM_WBRd,
        EX_MEMRegWrite, MEM_WBRegWrite, ForwardA, ForwardB);

        input [4:0] ID_EXRS1, IDEXRS2, EX_MEMRd, MEM_WBRd;
        input EX_MEMRegWrite, MEM_WBRegWrite;

        output [1:0] ForwardA, ForwardB;
                //decide whether the register declarations are needed
        reg [1:0] ForwardA, ForwardB;

        // fill in the missing code

        always @(ID_EXRS1 or IDEXRS2) //or EX_MEMRd or MEM_WBRd or EX_MEMRegWrite or MEM_WBRegWrite)
        begin
        if((EX_MEMRegWrite==1) && (EX_MEMRd !=0) && (EX_MEMRd == ID_EXRS1))
                begin
                ForwardA<=2'b10;
                end

        else if ((MEM_WBRegWrite==1) && (MEM_WBRd!=0) && ~ ((EX_MEMRegWrite==1) && (EX_MEMRd !=0)
                && (EX_MEMRd == ID_EXRS1))  && (MEM_WBRd==ID_EXRS1))
                begin
                ForwardA<=2'b01;
                end

        else
                begin
                ForwardA<=2'b00;
                end
```

```
        if ((EX_MEMRegWrite==1) && (EX_MEMRd !=0) && (EX_MEMRd == IDEXRS2))
                begin
                ForwardB<=2'b10;
                end
        else if ((MEM_WBRegWrite==1) && (MEM_WBRd!=0) && ~ ((EX_MEMRegWrite==1) && (EX_MEMRd !=0)
                 && (EX_MEMRd == IDEXRS2))  && (MEM_WBRd==IDEXRS2))
                begin
                ForwardB<=2'b01;
                End
        else
                begin
                ForwardB<=2'b00;
                end
        end
        initial
                begin
                ForwardA=2'b00;
                ForwardB=2'b00;
                end
endmodule
```

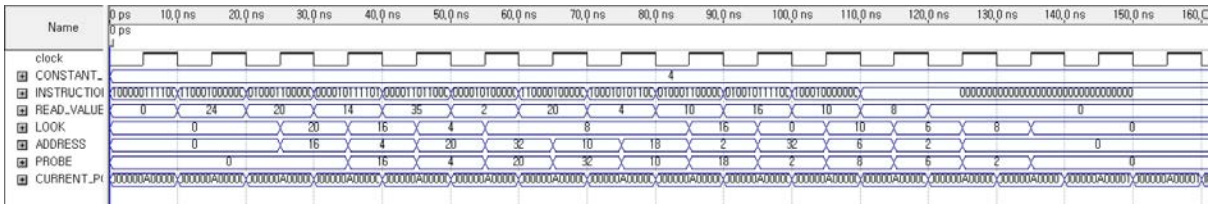
Since data hazard due to instruction dependence, pipelined RISC-V needs a part to detect whether there is any data hazard to occur and do forwarding operation to get rid of those hazards. Forwarding Unit part do the work. The part gets register number to read from ID/EX register, register numbers to write and RegWrite control signals from EX/MEM register and MEM/WB registers. Then it compares the registers numbers, check the RegWrite control signals are 1 or 0 so that it decides appropriate Forward signals for 3-1 MUXs to decide the value to calculate.

2. Waveform

-instruction

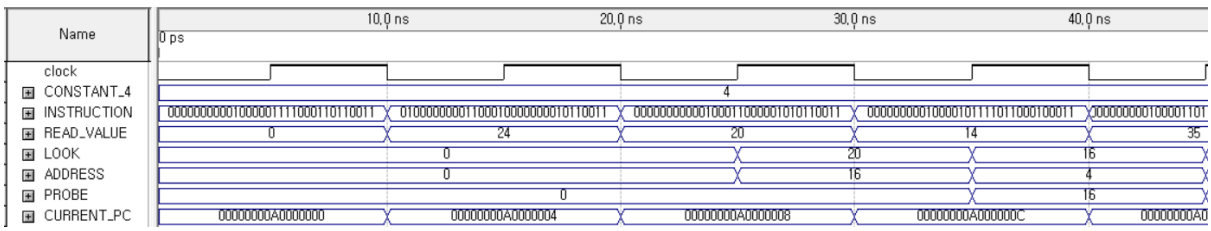
#	Instruction	Instruction in Machine Code	Expected Value				
			RV	Look	Address	Prove	Current PC
1	And x3, x1, x2	0000000/00010/00001/111/00011/0110011	24	20	16	16	A0000000
2	Sub x1, x2, x3	0100000/00011/00010/000/00001/0110011	20	16	4	4	A0000004
3	Add x5, x3, x1	0000000/00001/00011/000/00101/0110011	14	4	20	20	A0000008
4	Sd x4, 12(x5)	0000000/00100/00101/111/01100/0100011	35	8	32	?	A000000C
5	Or x5, x6, x4	0000000/00100/00110/110/00101/0110011	2	8	10	10	A0000010
6	Add x1, x5, x4	0000000/00100/00101/000/00001/0110011	20	8	18	18	A0000014
7	Beq x1, x3, 0x440	0001001/00011/00001/000/00000/1100111	4	16	2	?	A0000018
8	Ld x6, 22(x5)	0000000/10110/00101/011/00110/0000011	10	?	32	8	A000001C
9	Sub x1,x3,x5	0100000/00101/00011/000/00001/0110011	16	10	6	6	A0000020
10	And x3,x5, x1	0000000/00001/00101/111/00011/0110011	10	6	2	2	A0000024
11	Beq x4, x6, 0x440	0001001/00110/00100/000/00000/1100111	8		0	?	A0000028
							A0000032
							A0000036
							A0000148

-whole waveform result



In the schematic figure, 'current PC' output is in IF stage, 'read value' output is in ID stage, 'Look' and 'Address' stage in MEM stage, and the 'Probe' output is in WB stage. Therefore, we should see the signal for longer range of time to check whether the instruction worked well.

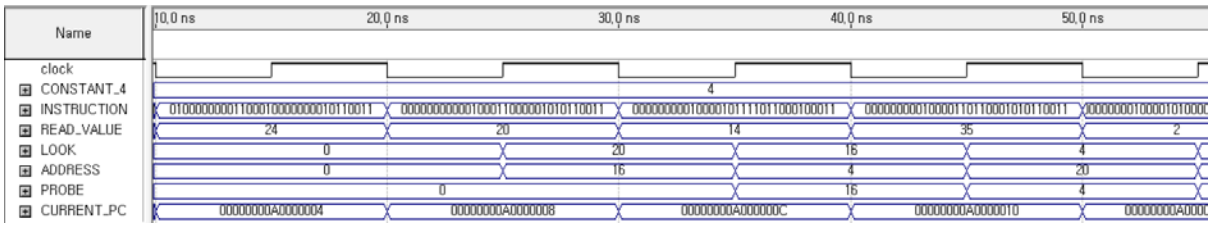
① And x3, x1, x2



The first instruction is 'and' instruction. At the IF stage(0ns~10ns), the instruction fetched. Then at

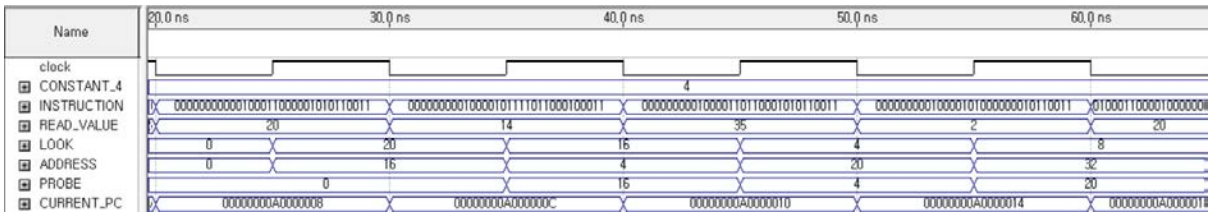
the following ID stage(10ns~20ns), processor got the register values of x1 and x2, and the value of x1, 24, is on the 'Read Value' output. Then, at the next positive value at 25ns, the 'LOOK' output shows the x2 value, 20, and the result of 'and' operation, which is 16, appears on 'ADDRESS' output. At the WB stage, the 'PROBE' output shows the value to written on x3 register, which is 16, same value as the 'LOOK' value. Since the instruction is not instruction, the PC value increase about 4.

② Sub x1, x2, x3



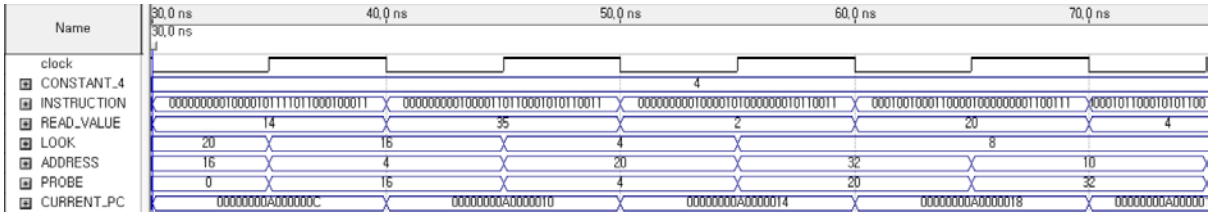
The second instruction is 'sub' instruction. At the IF stage(10ns~20ns), the instruction fetched. Then at the following ID stage(20ns~30ns), processor got the register values RD, which are x2 and x3, and the value of x2, 20, is on the 'Read Value' output. Then, at the next positive value at 35ns, the 'LOOK' output shows the x3 value, 16, and the result of 'sub' operation, which is 4, appears on 'ADDRESS' output. At the WB stage, the 'PROBE' output shows the value to written on x1 register, which is 4, same value as the 'LOOK' value. PC value still increases about 4.

③ Add x5, x3, x1



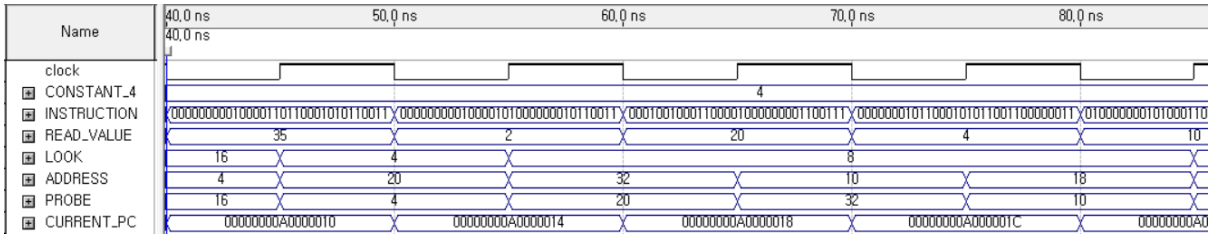
Next instruction is 'add' instruction. At 20ns~30ns, the instruction fetched, and for the next cycle, the processor got register value for x3 and x1. Since read value is the data for RD1(x3) before forwarding unit, the is sill 14. After the forwarding operation, ALU get new x3 and x1 data from EX/MEM register and MEM/WB register and do 'add' operation. The result of the operation is 16+4=20, which is appeared on the 'ADDRESS' output, and the forwarded value of X3, 4, is shown on the 'LOOK' output. Then the register got data to write, which is the result of calculation, 20, and it is shown in the 'PROBE' output. The current PC value increased about 4.

④ Sd x4, 12(x5)



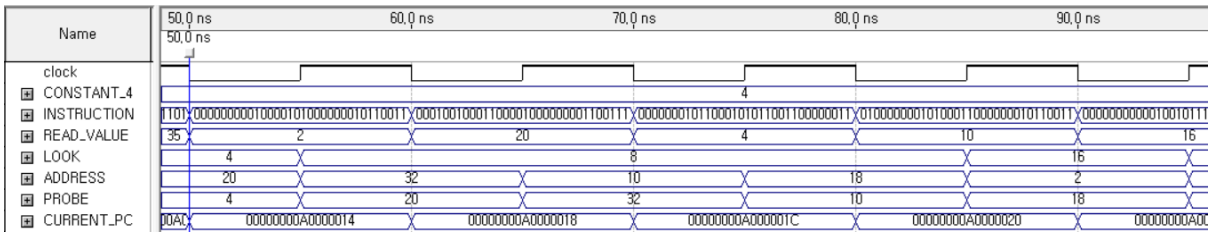
SD instruction comes next. The instruction fetched at IF stage, 30ns~40ns. Then the processor got value from x5 register, 35, which is not updated yet from the previous instruction, and value of x4, which is 8. In EX stage, ALU got the forwarded x5 value 20 and immediate value 12, then add them. The result 32 is on the 'ADDRESS' output on 55ns to 65ns. Then 8, which is the value of x4, is saved on the memory address. The value is on 'LOOK' output also on 55ns to 65ns. Since the MemtoReg signal is 'don't care', it did not change from the previous one, 0. Therefore, the value of 'PROBE' output is 32 on 65ns to 75ns. The PC value increases 4

⑤Or x5, x6, x4



For the operation, instruction fetched in IF stage, 40ns~50ns. Then the processor got value of x6 and x4, which is 2 and 8 each. The value of X6 is appeared on 'Read Value' output on 50ns~60ns, and the value of x4 is appeared on 'LOOK' output on 65ns to 75ns. The result of the or operation is 1010(2)=10, which is shown on the 'ADDRESS' output on 65ns to 75ns. Then the value is passed to the register to be written on X5 register, therefore the value 10 is on 'PROBE' output on 75ns to 85ns. The address still increases for 4.

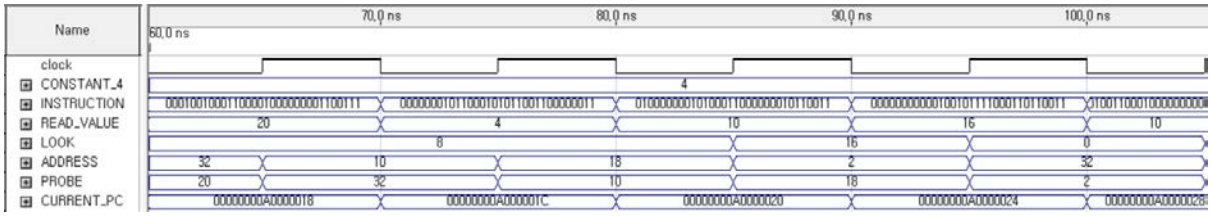
⑥ Add x1, x5, x4



The 6th instruction is add instruction. It fetched on 50ns to the 60ns. Since the value of x5 did not updated yet from the last instruction, the value is still 20 on 'Read Value' output. The value of x4 is 8, appeared on the 'LOOK' output on 75ns to 85ns. ALU of the processor gets value after forwarding, therefore the result of ALU calculation is 10+8=18, and its on the 'ADDRESS' output on 75ns to 85ns, and the value gets to the register to be written on x1 in next cycle, 85ns to 95ns. The address

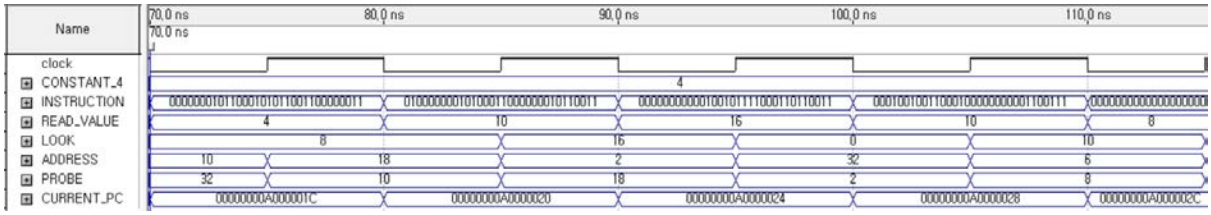
is still increase about 4.

⑦ Beq x1, x3, 0x440



The 7th instruction is beq instruction. Since the x1 value did not updated from the previous instruction, at the 'Read Value' output on 70ns to 80ns, the x1 value before forwarding operation, 4 appeared. After forwarding, x1 value is 18 and x3 value is 16. The x3 value is on 'LOOK' output, on 85ns to 95ns. In BEQ instruction, ALU check whether the values are same by subtract the given register values. The result of ALU subtraction is 18-16=2, therefore branch is not taken. The result value 2 is on 'ADDRESS' output. Since the MemtoReg signal is 'don't care' in BEQ instruction, the value does not change from 0 from previous instruction operation, therefore the 'PROBE' output gets the calculation result value 2, on 95ns to 105ns.

⑧ Ld x6, 22(x5)



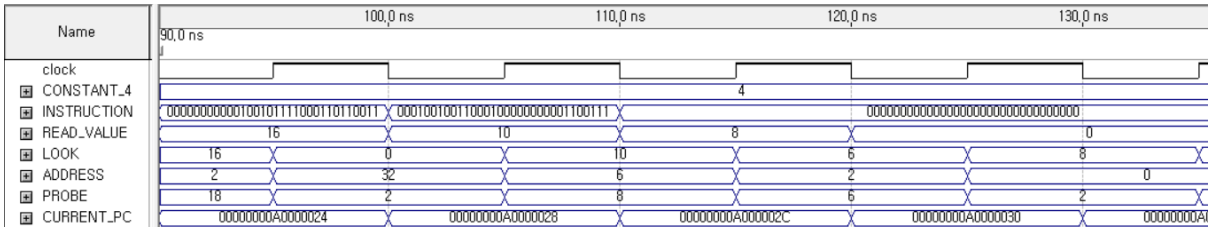
ld instruction fetched on 70ns to 80ns. THen the processor gets the register value of x5. The x5 value is 10 and its on 'Read Value' output on 80ns to 90ns. ALU calculates the memory address, adding immediate value 22 and x5 value 10. The result 22+10=32, and the value is on 'ADDRESS' output on 95ns to 105ns. Value for 'LOOK' output does not matter since the value is part of immediate value. Since 8 was saved on memory address 32 at the 4th instruction, the memory passes value 8 to the register, and the value saved at x6. The memory value 8 is on 'PROBE' output, 105ns to 110ns. The address become +4.

⑨ Sub x1, x3, x5

Next instruction is 'sub' instruction. At the IF stage(80ns~90ns), the instruction fetched. Then at the

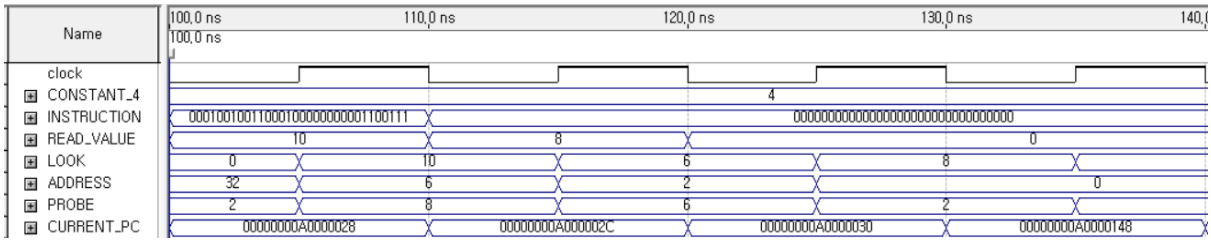
following ID stage(100ns~110ns), processor got the register values RD, which are x3 and x5, and the value of x3, 16, is on the 'Read Value' output. Then, at the next positive value at 105ns, the 'LOOK' output shows the x5 value, 10, and the result of 'sub' operation, which is 6, appears on 'ADDRESS' output. At the WB stage, the 'PROBE' output shows the value to be written on x1 register, which is 6, same value as the 'LOOK' value. PC value still increases about 4.

⑩ And x3, x5, x1



The 10th instruction is 'and' instruction. The instruction fetched in IF stage, 90ns to 100ns. Then at the following ID stage(100ns~110ns), processor got the register values of x5 and x1, and the value of x5, 10, is on the 'Read Value' output. Then, at the next positive value at 115ns, the 'LOOK' output shows the x2 value, 6, and the result of 'and' operation, which is 2, appears on 'ADDRESS' output. At the WB stage, the 'PROBE' output shows the value to written on x3 register, which is 2, same value as the 'LOOK' value. Since the instruction is not instruction, the PC value increase about 4.

⑪ Beq x4, x6, 0x440

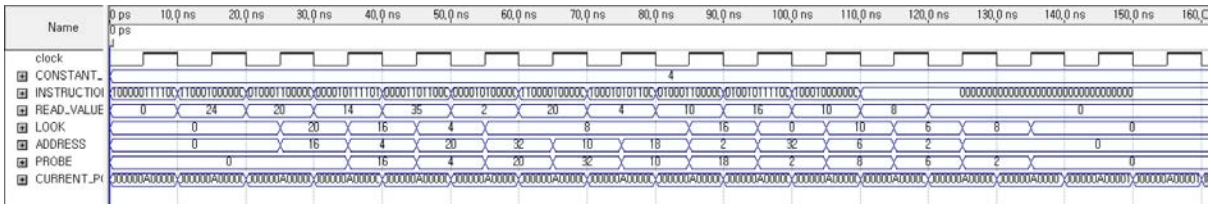


To check whether the PC value changes when the BEQ instruction taken, without considering flushing, I put beq instruction which is 'taken'. The instruction fetched in IF stage on 100ns to 110ns. In ID stage, the processor gets the values from given register. Value of X4 is on 'Read Value' output on 110ns to 120ns, which is 8. The value of x6, which is also 8, appeared on 'LOOK' output on 125ns to 135ns. Since both values are same, the subtraction result from ALU is 0, and it is shown on 'ADDRESS' output. 'PROBE' output got value of the result of ALU calculation, since the MemtoReg is 'don't care' therefore it did not change from the previous value 0. Since the branch is 'taken', the current PC value is added about 0x440=120(16) while the instruction is on MEM stage, few cycles after fetching.

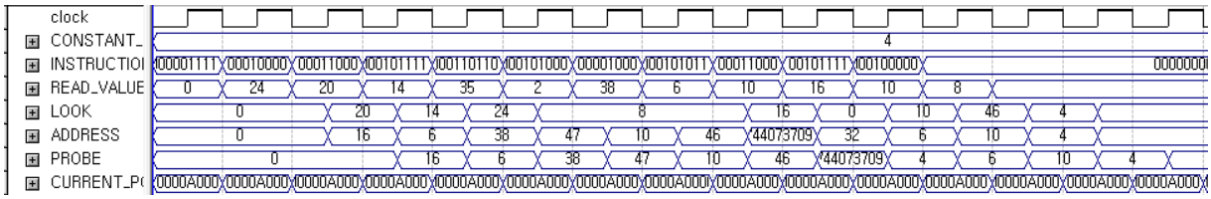
3. Problems

① compare the implementation without the forwarding unit and the one with the forwarding unit

If there is a forwarding unit, the result is just like the waveform below.



However, if there is no forwarding unit, the simulation result is apparently different.



If there is no forwarding unit, processor parts ins EX, MEM, and WB stage cannot get value intended in instruction. Instead, they get not-updated-register values. For example, in the second instruction, the processor get not-updated-x3 value, therefore it gets 14, instead of the right value 16, which is the result of previous instruction. These data hazards happen whenever there is any dependence on adjacent instruction. Therefore, to get intended result without forwarding unit, the processor should stall the following dependent instruction while the preceding dependent instruction processing is over, which is waste of clock cycle.

② Is the branch instruction (instruction 7, beq) taken or not? If not taken, how many instructions will be nop? Also, explain the appropriate method to resolve this problem.

Processor decides whether the conditional branch instruction is taken or not in MEM stage. Therefore, instead of stalling 3 instruction, lots of processors assumes conditional branch instruction will not be taken to reduce the average cost of the taken branch. With the assumption, there is no need to stall three instructions when the branch is taken.

In the given instruction, instruction 7 is not taken since the two register values are differ 18 and 16. However, if the branch instruction is taken, since the decision occurs in MEM, instructions in IF, ID and EX stages should be flushed by changing into 'nop' instruction.

To reduce the waste of processor performance, firstly we can implement some parts to make branch decision earlier. If the decision is made on ID or EX stage, we can reduce number of instructions to be flushed, though it requires more hardware parts. Also, we can improve branch prediction. If we predict whether the branch will be taken or not based on whether the conditional branch equation

was taken or not last time, the prediction would be more accurate, on average.

4. Discussion

In the project, the simulation output came out as the given instruction intended. Also, by using '=' and '<=' in Verilog HDL appropriately, errors which were just like structural hazard in register could be solved. However, since the design in the project does not have hazard detection unit, it cannot 'flush' the instruction with nop operation when conditional branch instruction taken, therefore I described the operational difference between when there is hazard detection unit, or when there is not.

5. References

Computer Organization and Design RISC-V Edition, David A. Patterson, John L. Hennessy, Morgan Kaufmann