EEE3530.02-00 Computer Architecture
Project#3: Pipelined RISC-V Implementation

2021142180 Kim Min Chan

Date: 2023.06.14

[Overview]

[1. Design Procedure]

[2. Result & Analysis]

[3. Discussion]

[1. Design Procedure]

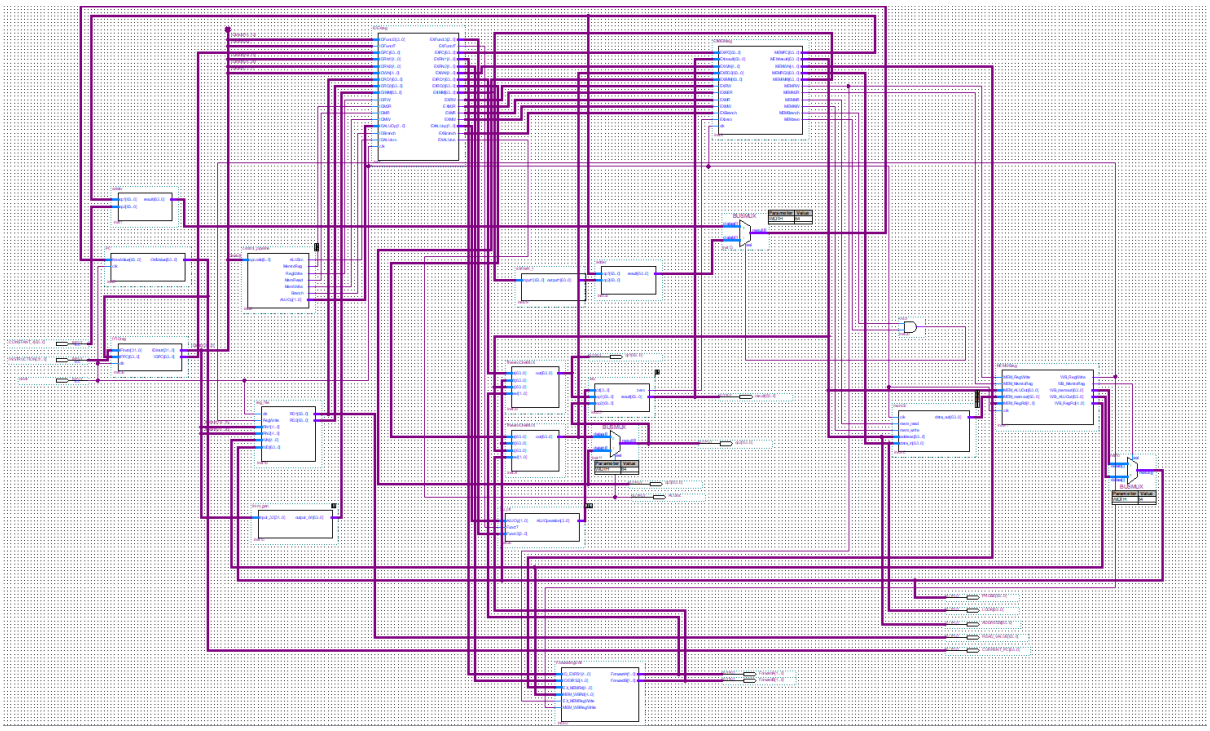

Figure 1.1 Overview of Microprocessor

1.1 IF Stage

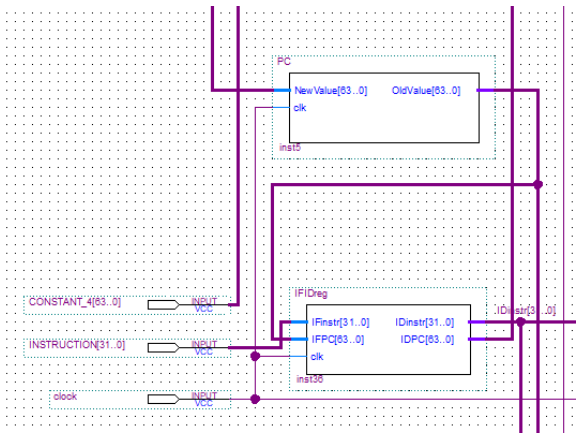

Figure 1.2 IF Stage

The IF (Instruction Fetch) stage is a step that utilizes the PC (Program Counter) value to fetch instructions from the Instruction Cache. The resources involved in this stage are the PC, Instruction Cache, and IF/ID Register. However, in this project, the Instruction Cache has not been implemented. The IF/ID Register is used to store the fetched instruction during the IF stage, and it will be passed on to the next stage (ID stage) for further processing and execution.

```
1  module IFIDreg(IFinstr, IFPC,
2                 IDinstr, IDPC,
3                 clk);
4
5      input [31:0] IFinstr;
6      input [63:0] IFPC;
7
8       output  [31:0] IDinstr;
9       output [63:0] IDPC;
10
11      reg  [31:0] IDinstr;
12      reg [63:0] IDPC;
13
14      input clk;
15
16      initial
17        begin
18              IDinstr <= 0;
19              IDPC <= 0;
20           end
21
22
23  always @(posedge clk)       // IF/ID Pipeline Register
24      begin
25              IDinstr <= IFinstr;
26              IDPC <= IFPC;
27      end
28
29
30  endmodule
```

Figure 1.3 Code For IF/ID Reg.

1.2 ID Stage

The ID (Instruction Decode) stage receives the current PC and instruction from the IF/ID register. In this stage, the instruction is decoded to generate the immediate value, create control signals, and read registers to store data in the ID/EX register. The ID (Instruction Decode) stage consists of several components, including the ID/EX register, control

pipeline, register file, and immediate generator (imm_gen).

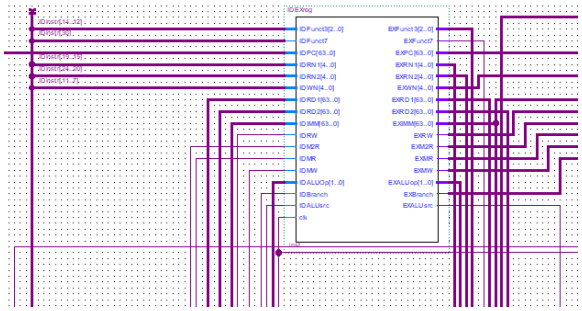

Figure 1.4 ID/EX reg.

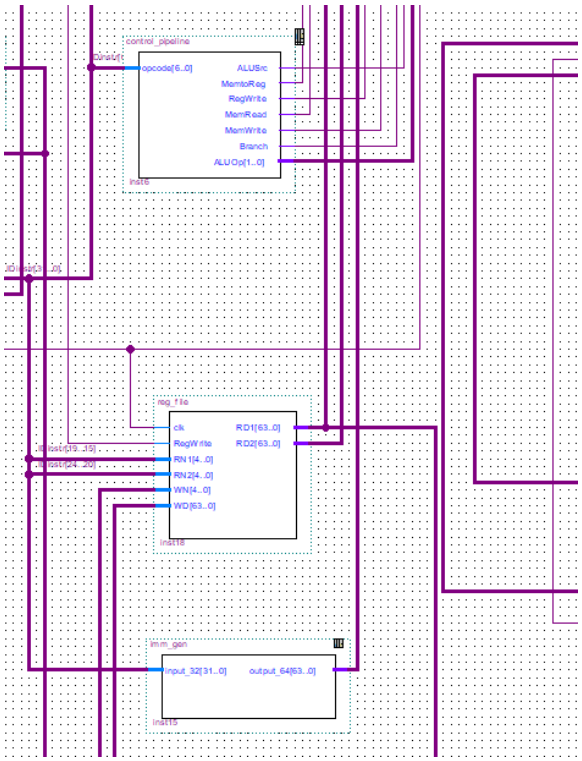

Figure 1.5 Control_Pipeline, Reg_File, Imm_gen

ID/EX register receives Rd, Rs1, Rs2, Instr[14:12], Instr[30] from the IF/ID register. It also receives 64-bit data RD1, RD2 read from the register file and a 64-bit immediate value generated by the immediate generator. These inputs are then passed to the EX stage. Based on this, the following code has been written. The code for the control_pipeline is based on the previous project's content.

```
module IDEXreg(IDFunct3, IDFunct7, IDPC, IDRN1, IDRN2, IDWN,
        IDRD1, IDRD2, IDIMM, IDRW, IDM2R, IDMR, IDMW, IDALUOp, IDBranch, IDALUSrc,
        EXFunct3, EXFunct7, EXPC, EXRN1, EXRN2, EXWN,
        EXRD1, EXRD2, EXIMM, EXRW, EXM2R, EXMR, EXMW, EXALUop, EXBranch, EXALUSrc, clk);

    input IDRW, IDM2R, IDMR, IDMW, IDBranch, IDFunct7, IDALUSrc, clk;
    input [63:0] IDPC, IDRD1, IDRD2, IDIMM;
    input [4:0] IDRN1, IDRN2, IDWN;
    input [2:0] IDFunct3;
    input [1:0] IDALUOp;

    output EXRW, EXM2R, EXMR, EXMW, EXBranch, EXFunct7, EXALUSrc;
    output [63:0] EXPC, EXRD1, EXRD2, EXIMM;
    output [4:0] EXRN1, EXRN2, EXWN;
    output [2:0] EXFunct3;
    output [1:0] EXALUop;

    reg EXRW, EXM2R, EXMR, EXMW, EXBranch, EXFunct7, EXALUSrc;
    reg [63:0] EXPC, EXRD1, EXRD2, EXIMM;
    reg [4:0] EXRN1, EXRN2, EXWN;
    reg [2:0] EXFunct3;
    reg [1:0] EXALUop;

    initial
      begin

        EXRW <= 0;
        EXM2R <= 0;
        EXMR <= 0;
        EXMW <= 0;
        EXBranch <= 0;
        EXALUSrc <= 0;
        EXFunct7 <= 0;
        EXIMM <= 0;
        EXPC <= 0;
        EXRD1 <= 0;
        EXRD2 <= 0;
        EXRN1 <= 0;
        EXRN2 <= 0;
        EXWN <= 0;
        EXFunct3 <= 0;
        EXALUop <= 0;

      end

always @(posedge clk)
   begin

      EXRW <= IDRW;
      EXM2R <= IDM2R;
      EXMR <= IDMR;
      EXMW <= IDMW;
      EXBranch <= IDBranch;
      EXFunct7 <= IDFunct7;
      EXALUSrc <= IDALUSrc;

      EXPC <= IDPC;
      EXRD1 <= IDRD1;
      EXRD2 <= IDRD2;
      EXIMM <= IDIMM;

      EXFunct3 <= IDFunct3;

      EXRN1 <= IDRN1;
      EXRN2 <= IDRN2;
      EXWN <= IDWN;

      EXALUop <= IDALUOp;
   end

endmodule
```

Figure 1.6 Code for ID/EX reg.

```
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 <= 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'b1;
          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

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

Figure 1.7 Code for Control_pipeline

1.3 EX Stage

The EX stage receives the necessary data and control signals from the ID/EX register and performs the operations in the ALU. This stage is composed of EX/MEM reg. and alu_ctl for controlling the ALU's operation, as well as two 3-to-1 MUX and two 2-to-1 MUX. The EX control signals are used in the EX stage and are not stored in the EX/MEM register since they are not needed in the subsequent stages.

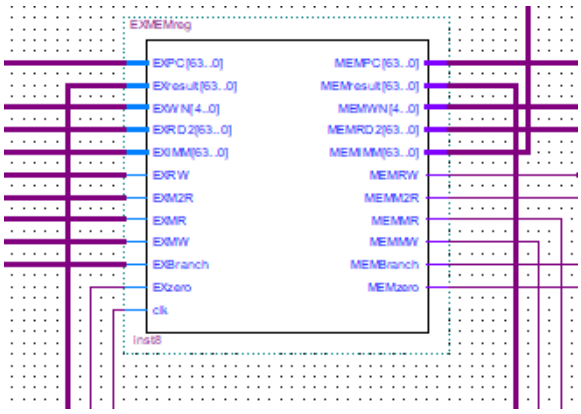

Figure 1.8 EX/MEM reg.

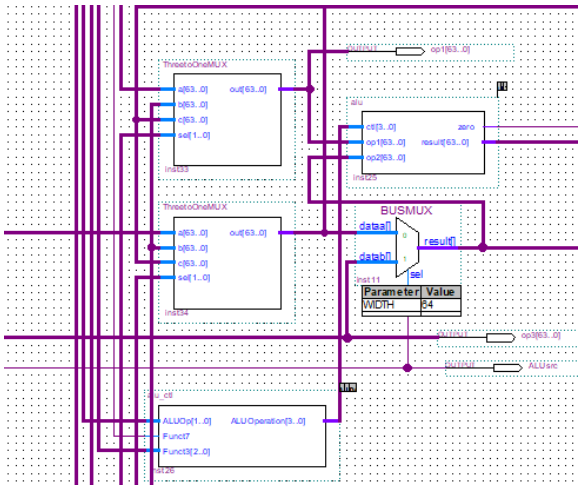

Figure 1.9 EX stage

The EX/MEM register receives 7 control signals, Rd, Rs1, Rs2, RD2, immediate value, and PC from the ID/EX register. Among the 7 control signals, ALUsrc and ALUop are used in the EX-stage and are not passed to the next stage. Based on this, the code can be written as follows.

```
module EXMEMreg(EXPC, EXresult, EXWN, EXRD2, EXIMM, EXRW, EXM2R, EXMR, EXMW, EXBranch, EXzero,
      MEMPC, MEMresult, MEMWN, MEMRD2, MEMIMM, MEMRW, MEMM2R, MEMMR, MEMMW, MEMBranch, MEMzero, clk);

   input EXRW, EXM2R, EXMR, EXMW, EXBranch, EXzero, clk;
   input [63:0] EXPC, EXresult, EXRD2, EXIMM;
   input [4:0] EXWN;

   output MEMRW, MEMM2R, MEMMR, MEMMW, MEMBranch, MEMzero;
   output [63:0] MEMPC, MEMresult, MEMRD2, MEMIMM;
   output [4:0] MEMWN;

   reg MEMRW, MEMM2R, MEMMR, MEMMW, MEMBranch, MEMzero;
   reg[63:0] MEMPC, MEMresult, MEMRD2, MEMIMM;
   reg [4:0] MEMWN;

initial
   begin
      MEMRW <= 0;
      MEMM2R <= 0;
      MEMMR <= 0;
      MEMMW <= 0;
      MEMBranch <= 0;
      MEMzero <= 0;
      MEMPC <= 0;
      MEMresult <= 0;
      MEMRD2 <= 0;
      MEMIMM <= 0;
      MEMWN <= 0;
   end

   always @(posedge clk)
      begin

         MEMRW <= EXRW;
         MEMM2R <= EXM2R;
         MEMMR <= EXMR;
         MEMMW <= EXMW;
         MEMBranch <= EXBranch;
         MEMzero <= EXzero;

         MEMPC <= EXPC;
         MEMresult <= EXresult;
         MEMRD2 <= EXRD2;
         MEMIMM <= EXIMM;

         MEMWN <= EXWN;

      end
endmodule
```

Figure 1.10 Code for EX/MEM reg.

1.4 MEM Stage

The MEM stage is a stage where data and control signals required from the EX/MEM register are used to access memory. In this stage, the resolution of Branch Hazard takes place. The MEM stage is composed of MEM/WB reg., Memory and an adder and a multiplexer for the for determining the next PC value.

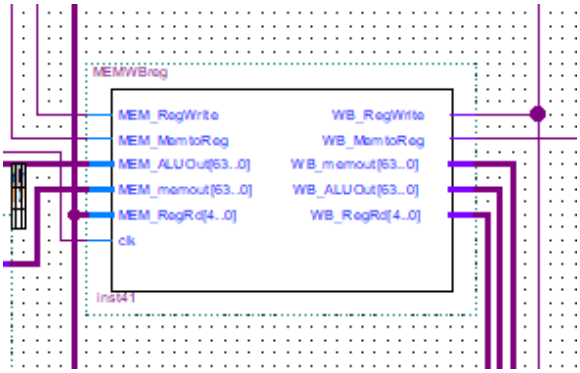

Figure 1.11 MEM/WB reg

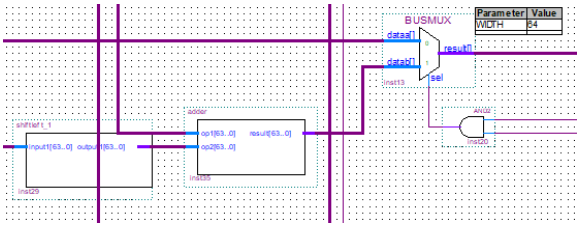

Figure 1.12 Next PC value

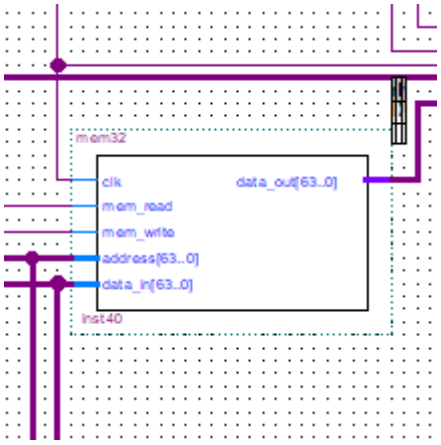

Figure 1.13 Memory

The MEM/WB register stores the WB control signals (RegWrite, MemtoReg), ALUout, Memout, and Rd, and passes them to the next stage, which is the WB stage. Based on this, the code can be written as follows.

```
module MEMWBreg(MEM_RegWrite, MEM_MemtoReg, MEM_ALUOut, MEM_memout, MEM_RegRd,
            WB_RegWrite, WB_MemtoReg, WB_memout, WB_ALUOut, WB_RegRd,
            clk);

    input MEM_RegWrite, MEM_MemtoReg;
    input [63:0] MEM_ALUOut, MEM_memout;
    input [4:0] MEM_RegRd;

    output WB_RegWrite, WB_MemtoReg;  // WB Control Signals
     output  [63:0] WB_memout, WB_ALUOut;
     output  [4:0] WB_RegRd;

    reg WB_RegWrite, WB_MemtoReg;  // WB Control Signals
     reg  [63:0] WB_memout, WB_ALUOut;
     reg  [4:0] WB_RegRd;

     wire [63:0] WB_wd;

     input clk;

     initial
       begin
            WB_RegWrite <= 0;
            WB_MemtoReg <= 0;
            WB_ALUOut   <= 0;
            WB_memout   <= 0;
            WB_RegRd    <= 0;
        end

always @(posedge clk)      // MEM/WB Pipeline Register
    begin
            WB_RegWrite <= MEM_RegWrite;
            WB_MemtoReg <= MEM_MemtoReg;
            WB_ALUOut   <= MEM_ALUOut;
            WB_memout   <= MEM_memout;
            WB_RegRd    <= MEM_RegRd;
    end

endmodule
```

Figure 1.14 Code for MEM/WB stage

1.5 WB Stage

The WB stage is the stage where necessary data and control signals are received from the MEM/WB stage, and data is written to registers. This stage consists of a multiplexer that selects which data to write. In terms of instruction execution, the WB stage is the final stage and does not have a pipeline register. Instead, the instruction register serves as a substitute. As mentioned earlier, the instruction register was not implemented in this project.

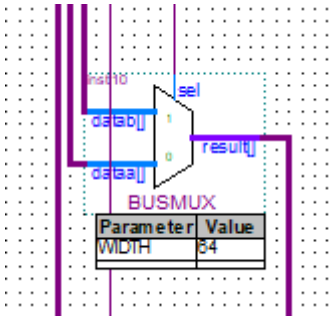

Figure 1.14 MUX for data selection

1.6 Forwarding Unit

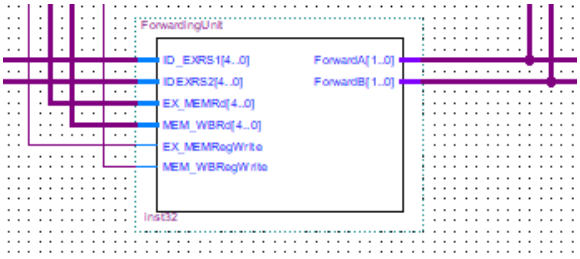

Figure 1.15 Forwarding Unit

The Forwarding Unit is a resource used to resolve Data Hazards. Data Hazards can occur due to EX Hazard and MEM Hazard, where the data in op1 and op2 of the ALU enter the pipeline without being updated, causing hazards. To resolve this, the Forwarding Unit generates control signals for a 3-to-1 multiplexer. In the case of an EX Hazard, it generates "10" to select the ALU result stored in EX/MEM. In the case of a MEM Hazard, it generates "01" to select the Mem out stored in MEM/WB. If no hazard occurs, it generates "00" to select the original data, Rd1 (or Rd2). Based on this, the code can be written as follows.

```
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)
    begin
        if(EX_MEMRegWrite == 1 && EX_MEMRd != 0 && ID_EXRS1 == EX_MEMRd)
            ForwardA <= 2'b10;
        else if ((MEM_WBRegWrite == 1 && MEM_WBRd != 0 && ID_EXRS1 == MEM_WBRd)
                && ~(EX_MEMRegWrite == 1 && EX_MEMRd != 0 && ID_EXRS1 == EX_MEMRd))
            ForwardA <= 2'b01;
        else
            ForwardA <= 0;

        if (EX_MEMRegWrite == 1 && EX_MEMRd != 0 && IDEXRS2 == EX_MEMRd)
            ForwardB <= 2'b10;
        else if ((MEM_WBRegWrite == 1 && MEM_WBRd != 0 && IDEXRS2 == MEM_WBRd)
                && ~(EX_MEMRegWrite == 1 && EX_MEMRd != 0 && IDEXRS2 == EX_MEMRd))
            ForwardB <= 2'b01;
        else
            ForwardB <= 0;
    end

endmodule
```

Figure 1.16 Code for Forwarding Unit

[2. Simulation & Analysis]

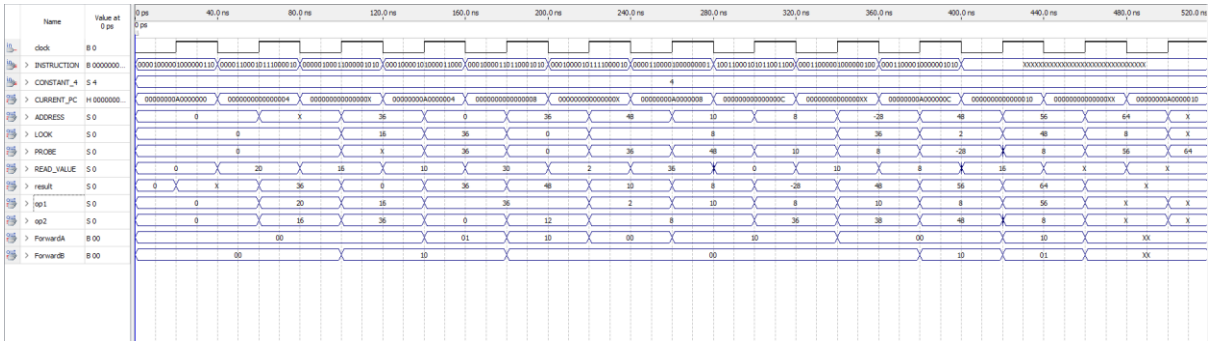

Figure 2.1 Simulation Result

2.1 Simulation Result

Before proceeding with the simulation, I have organized the values stored in each register and memory, as well as the Instruction Sequence and 32-bit Instruction Field, into a table.

Sequence	Instruction	Calculation
1	add x3, x1, x2	x3=x1+x2=36
2	and x1, x2, x3	x1=x2&x3=0
3	sub x5, x3, x1	x5=x3-x1=36
4	sd x4, 12(x5)	memory(48) <- x4=8
5	or x5, x6, x4	x5=x6||x4=10
6	and x1, x5, x4	x1=x5&x4=8
7	beq x1, x3, 0x440	x1!=x3
8	ld x6, 38(x5)	x6 <- memory(48)=8
9	add x2, x1, x6	x2=x1+x6=16
10	add x5, x2, x6	x5=x2+x6=24

Table 2.1 Instruction Sequence

Sequence	Instr. Field					
	FUNCT7/ imm(7)	rs2/ imm(5)	rs1	funct3	rd/ imm(5)	opcode
1	0000000	00010	00001	000	00011	0110011
2	0000000	00011	00010	111	00001	0110011
3	0100000	00001	00011	000	00101	0110011
4	0000000	00100	00101	000	01100	0100011
5	0000000	00100	00110	110	00101	0110011
6	0000000	00100	00101	111	00001	0110011
7	0100010	00011	00001	000	00000	1100011
8	0000001	00110	00101	011	00110	0000011
9	0000000	00110	00001	000	00010	0110011
10	0000000	00110	00010	000	00101	0110011

Table 2.2 Instruction Field

Sequence	Register No.						Memory
	1	2	3	4	5	6	
1	20	16	10	8	30	2	x
2	0	16	36	8	30	2	x
3	0	16	36	8	36	2	x
4	0	16	36	8	36	2	8
5	0	16	36	8	10	2	8
6	8	16	36	8	10	2	8
7	8	16	36	8	10	2	8
8	8	16	36	8	10	8	8
9	8	16	36	8	10	8	8
10	8	16	36	8	24	8	8

Table 2.3 Data Stored in Register/Memory

2.2 Analysis

ADDRESS is connected to the MEM_result of the EX/MEM reg., representing the result of the ALU operation. LOOK is connected to the MEMRD2 of the EX/MEM reg., indicating the result of the 3-to-1 MUX. PROBE is connected to the 2-to-1 MUX, representing the Write Data (WD). READ_VALUE is connected to RD1 of the reg_file, representing the value stored in Rs1, RD1.

Sequence	ADDRESS
1	x1+x2=36
2	x2&x3=0
3	x3-x1=36
4	x5+12=48
5	x6||x4=10
6	x5&x4=8
7	x-x3=-28
8	x5+38=48
9	x1+x6=16
10	x2+x6=24

Table 2.4 Theoretical Value of ADDRESS

First, for ADDRESS, we can observe meaningful results from the 4th cycle onwards. This is because ADDRESS is connected to MEM_result, and starting from the 4th cycle, the ALU result of the initially fetched instruction is displayed. Additionally, we can see that the instructions up to the 8th one match Table 2.4. The discrepancy between the 9th and 10th instructions with Table 2.1 is related to Data Hazards, Instruction Stall, and will be explained in more detail in the Discussion.

Regarding PROBE, we can see that it matches Table 2.5 up to the 8th instruction. However, we notice a difference in the 8th instruction compared to ADDRESS. This is because ADDRESS represents the ALU operation result, while PROBE indicates the data to be written back to the destination register.

Sequence	PROBE
1	x1+x2=36
2	x2&x3=0
3	x3-x1=36
4	x5+12=48
5	x6||x4=10
6	x5&x4=8
7	x-x3=-28
8	MEMORY(48)=8
9	x1+x6=16
10	x2+x6=24

Table 2.5 Theoretical Value of PROBE

Sequence	LOOK
1	x2=16
2	x3=36(Forwarded)
3	x1=0(Forwarded)
4	x4=8
5	x4=8
6	x4=8
7	x3=36
8	x6=2
9	x6=8
10	x6=8

Table 2.6 Theoretical Value of LOOK

LOOK matches Table 2.6 up to the 8th instruction. Since LOOK is connected to the result of the 3-to-1 MUX, we can observe the updated values influenced by the Forwarding Unit.

Sequence	READ_VALUE
1	x1=20
2	x2=16
3	x3=10
4	x5=30
5	x6=2
6	x5=36
7	x1=0
8	x5=10
9	x1=8
10	x2=16

Table 2.7 Theoretical Value of READ_VALUE

For READ_VALUE, we can see that all instructions match Table 2.6. By comparing READ_VALUE and LOOK, we can observe the impact of the Forwarding Unit. While LOOK reflects the updated result due to forwarding, READ_VALUE displays the unchanged values.

The 7th instruction is a BEQ instruction, and since the values stored in x1 and x3 are not equal, we can determine that the branch is not taken. However, if the branch were taken and the next instruction were executed, it would result in the flushing of two instructions. This is because, in the design implemented in this project, the branch operation is resolved in the MEM stage. This is related to Branch Hazard, and it will be explained in more detail in the Discussion section.

[3. Discussion]

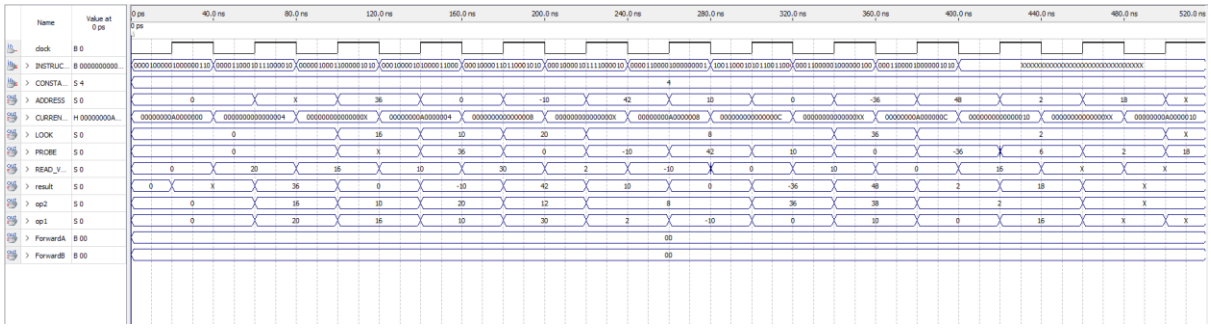

Figure 3.1 Simulation Result without Forwarding Unit

3.1 About Forwarding Unit

As mentioned earlier, the Forwarding Unit contributes to resolving Data Hazards through Data Forwarding. To examine the functionality of the Forwarding Unit, we conducted a simulation with the Forwarding Unit "disabled." The results obtained were consistent with Figure 3.1.

We can observe that ADDRESS, PROBE, LOOK, and READ_VALUE all differ from the theoretically calculated values. This discrepancy is due to Data Hazards, where the results of the previous instructions

are not updated, leading to issues when using values that have not been updated in the calculations.

Taking the third instruction "sub x5, x3, x1" as an example, when the Forwarding Unit is present, the calculated result of "x3-x1=36-0=36" matches the theoretical calculation. However, when the Forwarding Unit is not present, the values of x1 and x3 are not updated, resulting in x1=20 and x3=10, and the ALU operation yields "x3-x1=10-20=-10".

This indicates that without the Forwarding Unit, the Micro Processor does not operate correctly due to Data Hazards.

3.2 About Hazard of this Design

There is a critical flaw in the design implemented in this project, as it lacks a Hazard Detection Unit. As a result, it is vulnerable to certain hazards that can occur when implementing a pipeline.

3.2.1 About Data Hazard

There are two types of Data Hazards: EX Hazard and MEM Hazard. In the case of EX Hazard, there is no need for bubbles or instruction stalls as the result from the EX stage can be forwarded directly to the ALU's operand. However, in the case of MEM Hazard, where the value read from memory in the MEM stage needs to be passed to the ALU's operand, even with Data Forwarding, a one-cycle bubble is inevitable. This requires an instruction stall. Unfortunately, in the design of the project, there is no Hazard Detection Unit to detect and stall instructions, resulting in incomplete resolution of data hazards. This can be observed from the result of the 9th instruction in Figure 2.1.

If a Hazard Detection Unit were present, it would generate a one-cycle bubble through instruction stalls. This would allow the Forwarding Unit to receive the values MEM/WB RegWrite=1, MEM/WBRd=6, and IDEXRS2=6, generating ForwardB=01. As a result, the 3-to-1 MUX would select Memory(48)=8. However, without the one-cycle bubble, the Forwarding Unit receives EX/MEM RegWrite=1, EX/MEMRd=6, and IDEXRS2=6, generating ForwardB=10. This leads to op2=48 being inputted, resulting in an ALU operation of 48+8=56.

From this, we can observe that without a Hazard Detection Unit, the MEM Hazard, which is one of the two types of Data Hazards, cannot be fully resolved.

3.2.2 About Branch Hazard

In the design of this project, there is no Instruction Register, making Branch Instructions less significant. However, in actual microprocessor design, where instructions are executed based on the PC value, Branch Hazard becomes a significant issue.

The reason why Branch Instructions pose a problem is that the next PC value is determined in the MEM stage. Without using methods like Branch Prediction, a two-cycle bubble is unavoidable.

In the design of this project, without a Hazard Detection Unit, instruction stalls to create a two-cycle bubble will not occur. As a result, regardless of the presence of a branch instruction, the next instruction will continue to execute until the branch is encountered two cycles later. This can lead to issues if there are data dependencies between the executed instructions and the branched instruction. If data dependencies exist, incorrect values will be updated, causing the microprocessor to behave incorrectly. In this case, the two instructions that were executed after the branch should be flushed.

Therefore, in order to implement a pipelined microprocessor that operates correctly for all instruction flows, the Hazard Detection Unit is absolutely necessary.

If we use a Hazard Detection Unit, it leads to a 2-cycle bubble, which increases the total CPI and has a negative impact on performance.

To resolve this issue, there are two possible approaches. The first is to place the comparator and MUX in the IF stage, which reduces the bubble caused by the branch instruction to a single cycle. The second approach is Branch Prediction, where the prediction is made on whether the branch will be taken or untaken, allowing the next instruction to be fetched accordingly. There are two main types of Branch Prediction: Static Prediction and Dynamic Prediction. If the prediction is incorrect, two cycles of instructions need to be flushed, and the next instruction should be fetched accordingly.