RISC-V 32bit Custom 5-stages Pipeline RTL Design Project

1. 개요 및 요약

RV32I instruction set을 기반으로 5-stages Pipeline Processor를 Verilog를 이용하여 RTL Design을 구현한다.

특정 명령어 순서에 따른 Data Hazard 및 Control Hazard에 대해 Stall 및 Data Forward을 이용하여 처리함으로써 파이프라인내 문제점들을 해결한다.

미리 설정한 Instruction sequence 및 Data Memory를 참고하여 전체 RTL Design을 시뮬레이션하여 Hazard가 해결되었는지 확인해본다.

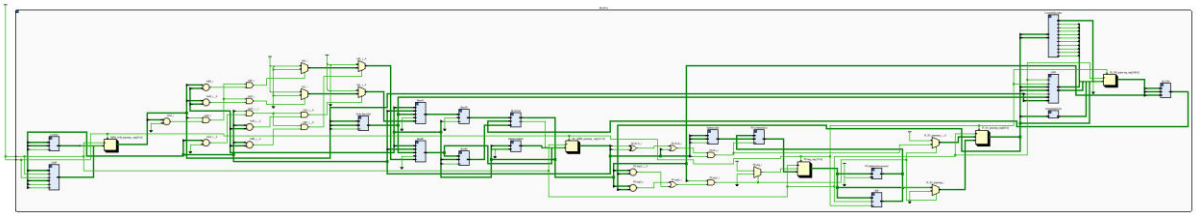


Fig 1. RTL Design Schematic

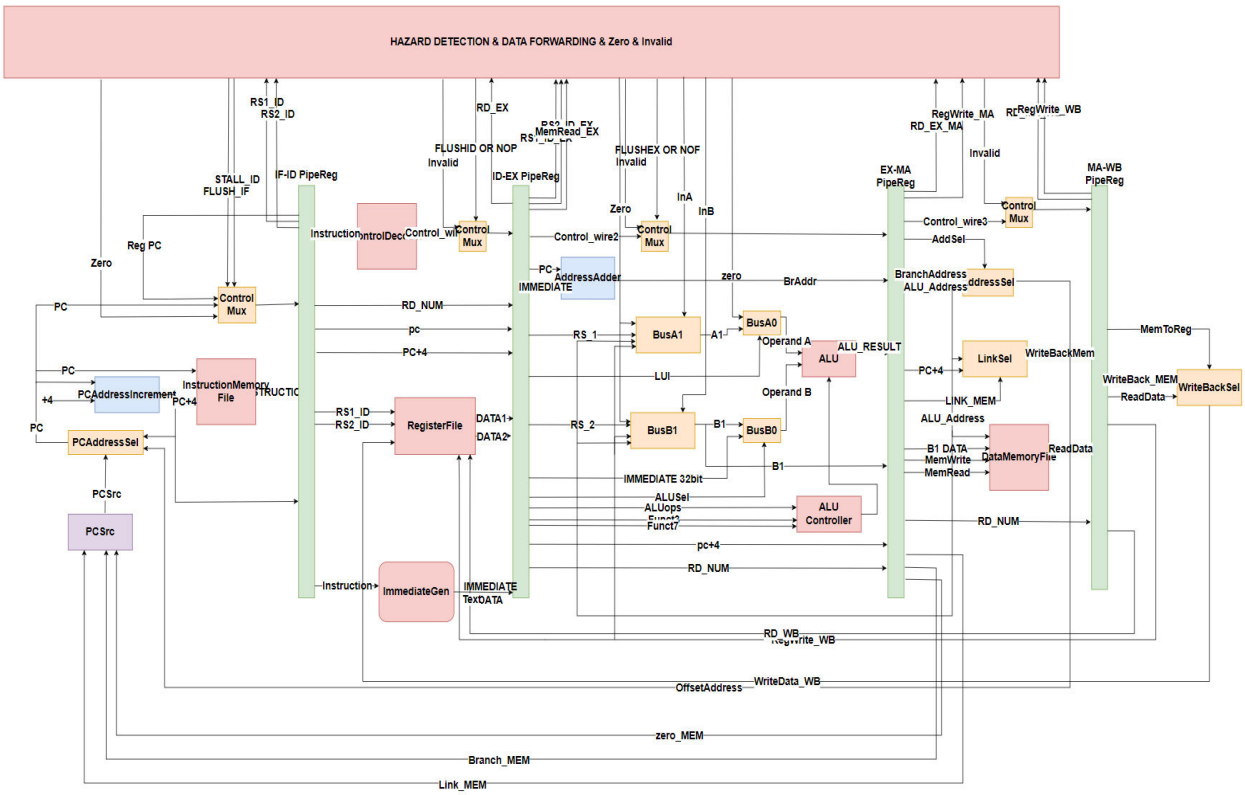


Fig 2. Custom 5-stages Pipeline Processor block diagram

목차

1. RV32I Instruction Set 및 Pipeline 구조

a. RV32I Instruction Set

RV32I 명령어 셋은 RISC-V 계열의 32bit Integer 명령어 셋에 해당한다. 모든 명령어의 길이가 32bit이며 그에 따라 한 word 단위도 32bit, 레지스터의 크기 또한 32bit에 해당한다.

RV32I Instruction Set은 총 32개의 레지스터를 가지고 있으며, 명령어의 종류에 따라 R-type, I-type, S-type, SB-type, UJ-type, U-type으로 나뉘게 된다.

R-type 명령어는 명령어의 참조 레지스터가 3개에 해당하며, 피연산자로 2개의 레지스터를 사용하고 결과 값을 저장하는 용도로 1개의 레지스터를 사용한다.

예를 들면 ADD, SUB, XOR, OR, AND와 같은 논리연산 명령어 및 산술 연산 명령어들이 존재한다.

add

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	00	rs2	rs1	000	rd	01100	11

sub

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
01000	00	rs2	rs1	000	rd	01100	11

xor

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
00000	00	rs2	rs1	100	rd	01100	11

Fig 3. R-type Instruction

R-type Instruction을 살펴보면 3개의 레지스터를 의미하는 rs1, rs2, rd와 funct3[14:12], funct7[31:25]를 가지는 특징이 있다.

또한 Opcode에 해당하는 [6:0] 7개의 비트를 가지며 이러한 비트 체계들을 분석하는 것이 Processor내 Controller를 만드는데 중요한 역할을 한다.

addi

add immediate

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[11:0]			rs1	000	rd	00100	11

lw

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:0]			rs1	010	rd	00000	11

Fig 4. I-type Instruction (Addi, load word)

다음으로 I-type 명령어이다.

I-type의 명령어는 일반적으로 산술 및 논리연산 명령어 및 레지스터 Load 명령어로 구분할 수 있다. 두 명령어 모두 2개의 레지스터를 사용하고, [31:20]의 12개의 비트 구역을 가지는데 이것을 Immediate 및 offset 비트라고 부른다.

Immediate와 offset의 구분은 주소의 연산에 사용되면 offset이라 나타내고 주소 연산이 아닌 데이터 연산으로 사용되면 immediate라 나타낸다.

산술 논리연산 명령어로 사용될 때는 Immediate값은 피연산자가 되고, Load 명령어에서 Offset 값은 base Register (rs1)에 따라 메모리 내 주소를 나타내는데 사용된다.

sw

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[11:5]		rs2	rs1	010	offset[4:0]	01000	11

Format	sw rs2,offset(rs1)
Description	Store 32-bit, values from the low bits of register rs2 to memory.
Implementation	M[x[rs1] + sext(offset)] = x[rs2][31:0]

Fig 5. S-type Instruction (store word)

다음으로 S-type 명령어이다.

S-type 명령어는 일반적으로 레지스터에 저장된 데이터를 메모리내 저장하는 명령을 수행한다. Fig 5 내 SW명령어는 word단위 저장으로, 4byte의 데이터를 메모리내 4 byte 공간에 저장하는 명령어이다. Word 단위 외에도 half-word, byte 단위로 저장 명령어가 존재한다.

beq

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[12|10:5]		rs2	rs1	000	offset[4:1|11]	11000	11

Format	beq rs1,rs2,offset
Description	Take the branch if registers rs1 and rs2 are equal.
Implementation	if (x[rs1] == x[rs2]) pc += sext(offset)

Fig 6. SB-type Instruction (branch equal)

SB-type 명령어는 상태에 따른 분기 명령어를 의미한다. 즉, 상태에 따라 명령어의 실행 위치를 바꾸는 것을 의미한다. 여기서 상태(Condition)란, 주어진 레지스터 rs1, rs2 간의 대소 비교 로 이루어지며, 분기가 발생하면 현재 PC 값 기준으로 OFFSET 값을 통해 실행할 명령어 주소를 구하게 된다.

jal

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
offset[20|10:1|11			19:12]		rd	11011	11

Format	jal rd,offset
Description	Jump to address and place return address in rd.
Implementation	x[rd] = pc+4; pc += sext(offset)

Fig 7. UJ-type Instruction (Jump and link)

UJ-type 명령어는 점프와 관련된 명령어들이다. 점프 명령어란 현재 Program Counter 레지스터가 가리키는 명령어의 주소가 아닌 다른 주소로의 이동을 의미한다. 즉, 순차적 명령어의 실행이 아닌 원하는 위치로의 명령어 실행이 가능하다.

Jal 명령어는 현재 PC(Program Counter)값을 저장할 레지스터인 rd와 현재 PC 값 기준으로 이동할 명령어의 주소 위치를 구할 때 사용되는 offset이 존재한다.

Jal 명령어는 PC+Offset의 명령어 주소를 얻어 다음 PC값으로 사용하며, 저장 레지스터 rd에는 현재 PC +4의 값을 저장하게 된다.

lui

load upper immediate.

31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
imm[31:12]					rd	01101	11

Fig 8. U-type Instruction (load upper immediate)

```
lui  x10, 0x87654       # x10 = 0x87654000
addi x10, x10, 0x321    # x10 = 0x87654321
```

Fig 9. Lui 및 Addi 사용 예시

U-type 명령어는 upper immediate를 위해 주로 사용된다. 앞서 설명한 Addi 명령어를 살펴보면 immediate에서 받아오는 비트 수가 12비트에 해당하게 된다. 하지만 레지스터의 비트 수는 32비트로 나머지 20개의 비트는 Sign-extension으로 채워지게 되는데, 이때 더 많은 데이터 값을 쉽게 할당할 수 있도록 하기위해 U-type 명령어가 존재한다.

b. Pipeline Structure

하나의 명령어를 하나의 사이클에서 실행하는 Single Cycle과 다르게, 하나의 명령어를 세분화하여 여러 사이클 내 세분화된 작업을 실행하여 하나의 명령어를 실행하는 구조가 파이프라인 구조이다.

Single Cycle 구조를 사용하지 않는 이유는 분명한데, 이는 add 명령어와 lw와 같은 메모리 접근하는 명령어가 서로 같은 Cycle를 소모하지 않는다는 것이다. 즉, 실제 실행되는 명령어당 걸리는 Cycle수가 서로 다르고 사용되는 하드웨어 또한 다르기에 이러한 Single Cycle 구조를 실제 제품이 접목시킨다는 것은 비효율적이라 할 수 있다.

그리하여 각 명령어의 구조를 분석하여 명령어 실행에 사용되는 과정을 세분화 및 사용되는 하드웨어를 구분하여 만든 구조를 파이프라인 구조라 할 수 있다.

이러한 방식을 통해 명령어들의 실행을 서로 중첩시킬 수 있고 같은 시간내 전체 명령어의 실행 수를 높일 수 있다.

(하나의 명령어에 대해 걸리는 Cycle수는 Single Cycle과 동일하지만 Cycle당 명령어 실행 수는 Pipeline에서 이득을 볼 수 있다.)

(명령어 실행에 대한 Latency는 줄일 수 없지만 전체 시스템의 Throughput을 늘릴 수 있다.)

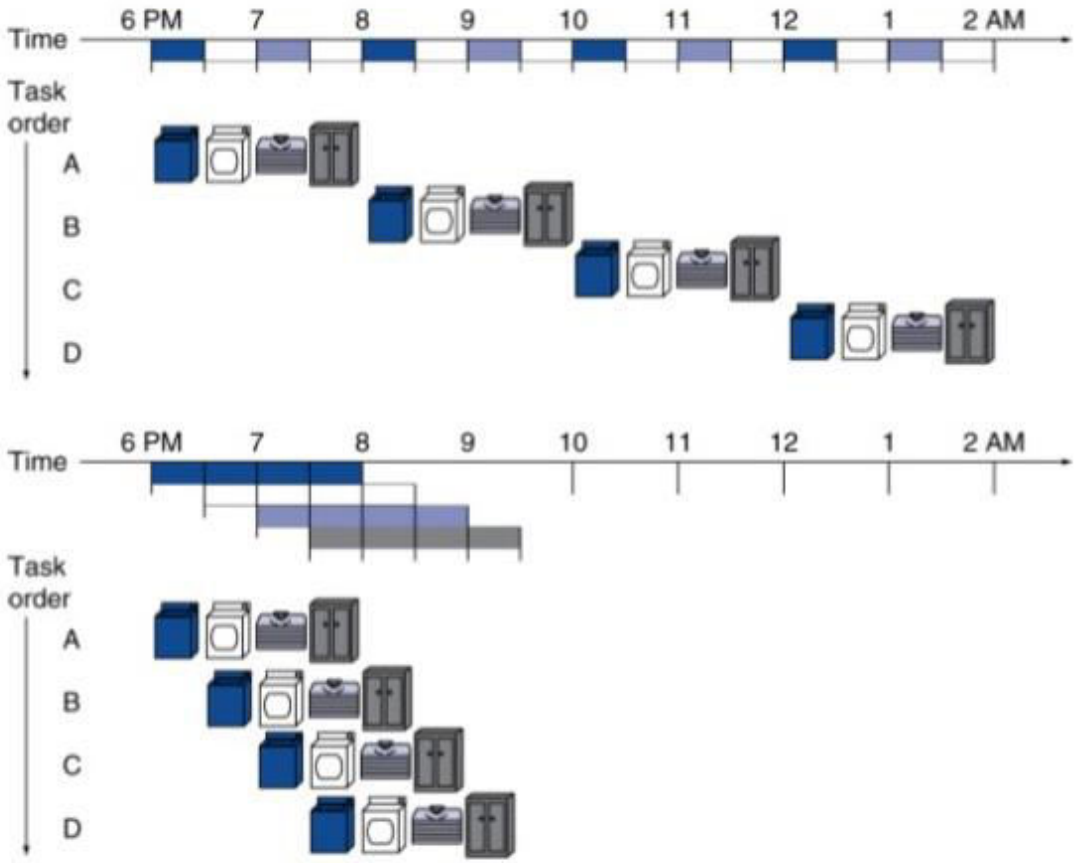


Fig 10. Single Cycle VS Pipeline structure

본 프로젝트에서 진행한 파이프라인은 5-stages Pipeline structure에 해당하며, 하나의 명령어 실행을 5가지 작업으로 나눠 명령어를 실행하는 구조이다.

작업은 IF, ID, EX, MA, WB으로 나누었다. IF는 Instruction Fetch를 의미하며 PC값에 따라 명령어를 가져오는 단계라 할 수 있다.

ID는 Instruction Decode를 의미하며, IF단계에서 가져온 명령어에 따라 Control signal, Immediate Sign-extension 및 Register File에 접근하는 단계이다.

EX 단계는 Execute 단계를 의미하며 해당 단계에서는 ALU를 통한 연산 과정을 진행한다.

MA단계는 Memory Access 단계를 의미하며, LW 및 SW 명령어 등 메모리에 접근하여 레지스터 내 값을 저장하거나 레지스터에 값을 할당하는 과정이다.

마지막 단계인 WB단계는 Writeback 단계에 해당하며 Register File 내 존재하는 레지스터에 값을 갱신해주는 단계를 의미한다.

본 프로젝트의 파이프라인은 이렇게 5가지 역할을 가지는 단계로 나누어져 있다.

2. Pipeline내 hazard 처리 방안 제시

Hazard란 하나의 명령어를 여러 개의 작업으로 나눔으로써 발생하는 문제점들을 의미한다.

일반적으로 명령어를 여러 개의 작업으로 나눌 때, 서로 독립적으로 작업이 진행되며 서로 간의 간섭이 존재하지 않아야 이상적이다.

하지만 이런 이상적 환경을 만드는 것을 불가능에 가깝기에 문제를 최소화한 상태에서 문제 해결에 집중하는 것이 중요하다.

a. Structural Hazard

Structural Hazard는 명령어내 작업들이 서로 같은 하드웨어 자원을 사용하려는 경우 발생하는 문제라 할 수 있다.

예를 들어, IF 단계에서 Data Memory File에 접근하는 작업이 존재하고, MA 단계에서도 Data Memory File에 접근하는 과정이 존재하는 경우, 이를 Structural Hazard가 존재한다고 할 수 있다.

하지만 본 프로젝트에 사용된 RISC-V 계열 Instruction에서의 5-stages pipeline 내에서는 Structural Hazard가 존재하지 않기에, 이를 해결하지 않아도 된다.

b. Data Hazard

Data Hazard는 명령어내 작업들이 서로 같은 Data에 접근하려고 하거나 수정되지 않은 데이터에 접근하려는 상태의 문제를 의미한다.

예를 들면, WB작업이 진행되지 않은 상태에서 동일한 레지스터에 접근하려는 작업이 존재한다면, 실제로는 갱신된 데이터 값을 사용해야 하는데, 해당 상황에서는 갱신되지 않은 데이터를 이용하여 명령어를 처리할 것이다. 이러한 문제를 Data Hazard라 할 수 있다.

본 프로젝트에서는 Data Hazard가 발생할 수 있는 상황을 2가지로 나누었다.

첫번째 상황은 Load 명령어가 존재하는 상황에서 해당 명령어의 Rd 레지스터에 동일한 접근을 하는 다음(next) 명령어가 존재하는 경우이다.

해당 경우에서는 Stall (Bubble)을 사용하여 처리할 수 있다. 하지만 명령어 실행 순서가 load -> add 인 순서라면, Stall을 3번을 사용해야 Data Hazard를 없앨 수 있다. 이는 전체 명령어 실행 시간에 영향을 크게 주기에, Data Forward(Bypassing) 과정을 넣어주면 Stall을 1번만 사용해도 된다. 이러한 해결책을 이용할 예정이다.

- **Stall: Hazard를 없애기 위해 현재 명령어의 실행이 진행될 동안 다음 명령어내 작업이 진행되지 않도록 하는 기술이다.**
- **Data Forward(Bypassing)은 명령어의 종료 이후가 아닌, 작업 도중에 다른 명령어의 작업으로 Data를 보내는 것을 의미하며, 이를 통해 이전 명령어의 WB작업을 기다리지 않고도 Data를 원하는 과정으로 보낼 수 있다.**

c. Control Hazard

Control Hazard는 명령어의 분기에 의해 나타나는 문제이다. 일반적으로 Pipeline 과정에서는 명령어의 순차 과정을 의식하여 Instruction Fetch 과정을 진행한다. 하지만 분기가 발생할 경우 해당 과정으로 Fetch된 명령어들은 실행하지 않아야 하는 명령어들에 해당된다. 분기가 실행되는지 알 수 있는 과정은 EX 이후에 알 수 있다.

만약 분기가 발생하게 된다면 EX 작업 이전에 수행되는 명령어들의 작업들은 Control Hazard에 해당되며 해당 작업들은 제거되어야 한다.

제거하는 방식으로 Flush 방식을 채택하였다. 이전 작업들의 Control Signal을 전부 Invalid로 설정하여 아무것도 실행되지 않게 만드는 방식이다.

Flush를 통해 Control Hazard를 해결할 수 있다.

- Flush: Pipeline내 존재하는 Control Hazard를 처리하기 위해 이전 실행되는 명령어들의 실행을 무효화 시키는 방식

결론적으로 Hazard는 Structural Hazard, Data Hazard, Control Hazard로 3가지의 Hazard가 존재하지만 실제 RISC-V 내 존재하는 Hazard는 2가지로 Data Hazard, Control Hazard이다. 이러한 Hazard의 해결은 STALL, DATA FORWARD, FLUSH 방식을 통해 가능하다.

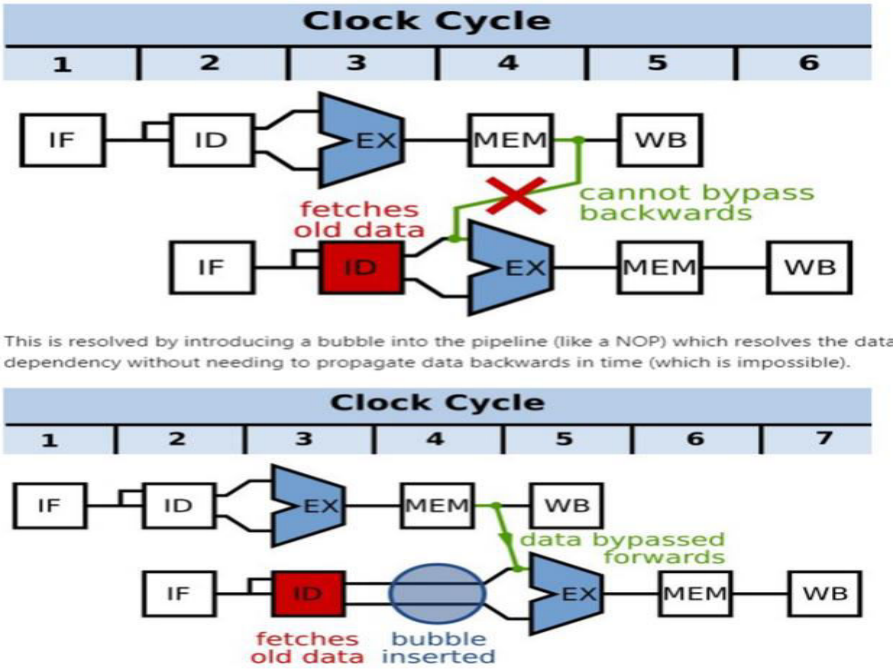


Fig 11. ID-MA Hazard (STALL -> DATA FORWARD)

3. Pipeline 구조에 대한 RTL Design 방안 및 분석

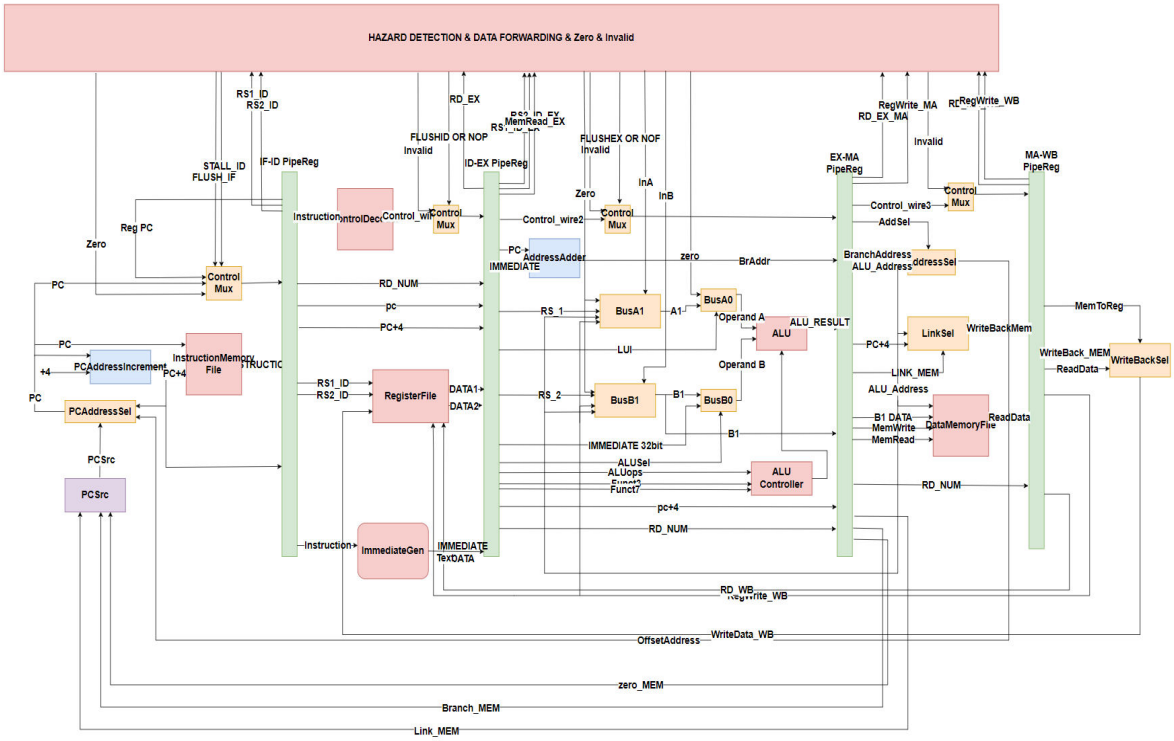


Fig 12. 프로젝트내 RTL Design의 block diagram

Fig 12을 참고하면 RTL Design을 통해 구현된 하드웨어 및 Signal, Dataflow가 담겨져 있다.

해당 구조에서는 앞서 말한 단계, IF-ID-EX-MA-WB 작업들 단계 사이에 pipeReg를 설정하여 각 단계들의 실행 이후 데이터를 pipeReg에 넣어 Hazard 처리 및 Clock Cycle에 따른 작업 실행을 고려해주었다.

전체 하드웨어에서 PC값 설정 및 IF-ID, ID-EX, EX-MA, MA-WB pipeReg를 제외하곤 모두 Clock과 비 동기적으로 진행된다.

Clock에 동기화되어 진행되는 곳은 PC값을 설정하는 곳과 위 4 개의 단계별 pipeReg에 해당하고 해당 작업마다 1개의 클럭을 소모하게 된다. 그렇게 5개의 클럭을 소모해야 하나의 Instruction 실행이 완료된다.

```
module Add(Result, A, B);

    input       [31:0] A,B;
    output  reg [31:0] Result;

always @(A,B)
begin
    Result=A+B;
end
endmodule
```

```
module Mux2(Out, I0, I1, Sel); // 2 X 1 ,32 Bit wide

    input  [31:0] I0,I1;
    input  Sel;
    output [31:0] Out;

    assign Out=(~Sel)?I0:I1;

endmodule

module Mux4(Out, I0, I1, I2, I3, Sel); // 4 X 1 ,32 Bit wide

     input  [31:0] I0,I1,I2,I3;
     input  [1:0] Sel;
     output reg [31:0] Out;

    always@(I0,I1,I2,I3,Sel)
    begin
        case(Sel)
            2'b00: Out=I0;
            2'b01: Out=I1;
            2'b10: Out=I2;
            2'b11: Out=I3;
            default: Out=I0;
        endcase // Sel
    end
endmodule
```

Fig 13. 32-bit adder Module 및 32-bit 2-1 MUX, 32-bit 4-1 MUX

해당 모듈은 Fig 12 block diagram 내 PCAdressIncrement 와 AddressAdder로 사용된다.

PCAddressIncrement는 PC +4를 통해 순차적 명령어 실행 순서에서 다음 명령어 주소를 구하는 모듈이다.

+4를 해주는 이유는 명령어가 32bit로 되어 있지만 Memory 구조는 byte Address를 채택하기에 32가 아닌 4 byte를 더해 다음 명령어 주소를 구하게 된다.

AddressAdder는 현재 명령어 주소인 PC 값과 OFFSET 값을 서로 더해주는 연산에 해당한다. 해당 연산을 통해 분기 시 실행할 명령어 주소 값을 정하게 된다.

32bit 2-1 MUX는 BusA0, BusB0, PCAddressSel, WriteBackSel, LinkSel에 사용된다.

BusA0는 LUI 연산에 사용된다. ID 작업을 통해 만약 명령어가 LUI인 경우에는 Register 값을 사용하지 않기에 zero(0) 과 Immediate값을 ALU에서 연산하는 것이다.

BusB0는 레지스터 rs2를 사용하는 가 혹은 Immediate 값을 사용하는 가를 정한다. 해당과정은 ID 작업을 통해 만약 명령어가 Immediate 혹은 offset이 존재하는 지에 따라 결정하게 된다.

PCAddressSel은 다음 PC값으로 PC+4 혹은 분기된 PC값을 가져야 하는 지 결정하는 모듈이다.

WriteBackSel은 ALU연산 결과를 Register로 저장해야 하는지 혹은 Memory에서 Read 한 데이터를 저장해야 하는 지 정하는 모듈이다.

LinkSel은 PC+4을 Register에 저장해야 하는지 혹은 ALU Result을 저장해야 하는지 정하는 모듈에 해당한다.

32bit 4-1 MUX는 BusA1, BusB2에 사용한다.

4-1 MUX는 data forwarding에 매우 중요한 역할을 하는데, 4-1 MUX로 들어오는 Input은 현재 레지스터 rs1 혹은 rs2의 값과 이전 명령어의 EX 와 MA의 결과물에 해당하게 된다. 여기서 Hazard에 따라 ID-MA 간의 DATA HAZARD의 경우에는 MA의 결과물이 선택되고 ID-EX 간의 DATA HAZARD의 경우에는 EX의 결과물이 선택되게 된다.

이 작업이 rs1, rs2에 모두 적용되어야 하기에 BusA1, BusB1 둘 다 존재한다.

```
module InstructionMemoryFile(Address,Data,Clk,Rst);
    output      [31:0]  Data;
    input       [31:0]  Address;
    input               Clk,Rst;

    reg         [ 7:0]  imembank [0:63];  //  8x64  64B memory

initial begin $readmemh("Imem.txt",imembank);  end

assign Data = {imembank[Address+3'b11],imembank[Address+2'b10],
    imembank[Address+2'b01],imembank[Address]} ;

endmodule
```

```
module RegisterFile(data1, data2, read1, read2, writeReg, writeData, Clk, Rst, regWen);
    input       [31:0] writeData;
    input       [ 4:0] read1, read2, writeReg;  // Register Number
    input              Clk, regWen, Rst; //regWen => Write Reg signal

    output      [31:0] data1, data2;

    reg         [31:0] registerbank [0:31];

    initial begin
        $readmemh("Rmem.txt",registerbank);  //Register Reset
    end

    always @(posedge Clk)
    begin
        if(~Rst)
            begin
                $readmemh("Rmem.txt",registerbank);
            end
    end

    always @( * )                   // Writing at Negative Edge of clock
    begin
        registerbank[0] <= 32'd0;   // Regster 0번은 항상 0의값을 가진다.

        if(regWen)
            registerbank[writeReg] <= writeData;
    end

    assign data1 = registerbank[read1]; // Port for Rs1
    assign data2 = registerbank[read2]; // Port for Rs2

endmodule
```

Fig14. Instruction Memory File Module & RegisterFile Module

Instruction Memory File은 주어진 PC값에 따라 Instruction Memory에서 해당하는 주소의 명령어를 출력하는 모듈이다.

해당 메모리는 미리 입력해둔 Instruction sequence에 따라 명령어가 나오게 되며 32bit의 명령어 주소를 받지만 메모리 자체는 byte 체계를 가지기에 4바이트로 32비트를 전환해야 한다.

Register File은 주어진 Register number에 따라 Register File내 있는 레지스터의 값을 출력하게 된다.

해당 모듈은 Instruction Decode (ID) & Writeback (WB) 과정에서 사용되는 모듈이다. Register File 내 Writeback 단계는 Clock에 동기화되어 작동하여 Register 값을 갱신한다.

```
module Control(Opcode,funct3,ALUsrc,MemtoReg,RegWrite,MemRead,MemWrite,AddSel,Link,Branch1,Branch0,ALUOp,Lui);

output [1:0] ALUOp;
input  [6:0] Opcode;
input  [2:0] funct3;

output  reg ALUsrc,RegWrite,MemWrite,MemtoReg,MemRead,AddSel,Link,Lui;
output wire  Branch1,Branch0;

reg ALUOp1,ALUOp0;
reg    [1:0] Branch;
initial begin Branch =2'b00; end

assign {Branch1,Branch0}=Branch;

assign ALUOp= {ALUOp1,ALUOp0};


// ALUsrc -> Using  Immediate or NOT
// MemtoReg -> Using memory data for write register or not
// RegWrite -> Update Register or not
// MemRead -> Using memory , read data or not
// MemWrite -> Using memory, write data to register or not
// AddSel ->  Using for  jump process(branch address) based Reg or not
// Link -> jal, jr etc jump process
// Lui -> instruction for lui, lui instruction can make reg have 32bit immediate value
// Branch -> is it branch process or not
// ALUop1 ->
```

	R-type	I-type (load)	I-type (Arth)	UJ-type (Jalr)	S-type	SB-type	U-type	UJ-type (Jal)
ALUsrc	0	1	1	1	1	0	1	1
MemtoReg	1	0	1	1	X	X	1	1
RegWrite	1	1	1	1	0	X	1	1
MemRead	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	1	0	0	0
AddSel	X	X	X	1	X	0	X	0
Link	0	0	0	1	0	0	0	1
Lui	0	0	0	0	0	0	1	0
Branch	00	00	00	10	00	10 or 11	00	10
ALUOp	10	00	11	01	00	01	00	01

Fig 15. Control Module & Signals Table based on Instruction Type

Control 모듈은 Instruction Fetch 과정에서 받은 Instruction 정보를 토대로 EX, MA, WB 각 작업 단계에 필요한 신호의 값을 결정하는 모듈이다.

해당 모듈은 Instruction Decode 작업 과정에 해당하며 해당 과정에서 명령어에 따라 필요한 신호들을 결정하게 된다.

명령어에 따른 각 신호 값은 Fig 15 내 Signals Table based on Instruction Type의 값에 따라 정해진다.

```
module ImGen(Out,Instruction);

    input       [31:0] Instruction;
    output reg  [31:0] Out;

    wire [6:0] Opcode;
    wire [31:0] Ins;

    // Extracts and Extends the Immediate values from different types of Instruction
        always@(Instruction) begin
            begin
                case(Opcode)
                    7'b0000011,7'b0010011,7'b1100111 : Out = {{20{Ins[31]}},Ins[31:20]}; //signed extension
                    // 12 Bit Imm at Ins[31:20] for I-type (Load,Arith,Jalr)
                    7'b0100011  : Out = {{20{Ins[31]}},Ins[31:25],Ins[11:7]}; //signed extension
                    // 12 Bit Imm at Ins[31:25],Ins[11:7] for S-Type
                    7'b1100111  : Out = {{20{Ins[31]}},Ins[31],Ins[7],Ins[30:25],Ins[11:8]}; // signed extension
                    // 12 Bit Imm for SB-type
                    7'b1101111  : Out = {{12{Ins[31]}},Ins[31],Ins[19:12],Ins[20],Ins[30:21]}; //signed extension,  ohnage instruction address
                    //  20 Bit Imm for UJ-Type
                    7'b0110111  : Out = {Ins[31:12],12'h000};"
                    //  20 Bit Imm for J-Type (LUI)
                    default: Out= 32'hZZZZ;
                endcase // Opcode
            end
        end

    assign Opcode = Instruction[6:0];   // Extract Opcode
    assign Ins = Instruction;

endmodule
```

Fig 16. Immediate Generator Module

위 모듈은 Immediate 혹은 offset의 값을 받아 해당 값을 32bit에 맞춰 변경하는 역할을 수행한다. 일반적으로 Sign-extension 작업이 이루어진다.

```
module DataMemoryFile(ReadData,Address,WriteData,memWrite,memRead,Clk,Rst);

    input       [31:0]  Address;
    input       [31:0]  WriteData;
    input               Clk,memWrite,memRead,Rst;

    output      [31:0]  ReadData;

    reg        [7:0] dataMem [0:63];  //8x64 Bits = 64 Byte memory

    initial begin $readmemh("Dmem.txt",dataMem);  end

    assign ReadData =(memRead)  ?   {dataMem[Address+2'b11],dataMem[Address+2'b10],dataMem[Address+2'b01],dataMem[Address]}:32'hZZZZZZZZ;
              // Scoops 4 8 bit memory locations at a time in Little Endian
    always @( * )
    begin
        if(memWrite) begin
             {dataMem[Address+2'b11],dataMem[Address+2'b10],dataMem[Address+2'b01],dataMem[Address]} <= WriteData;
        end
    end
endmodule
```

Fig 17. Data Memory File Module

해당 모듈은 MA(Memory Access)단계에서 사용되는 모듈이며 LW, SW 작업이 진행된다.

해당 모듈로 들어오는 입력신호들은 Clock에 동기화된 EX-MA PipeReg를 통해 들어오기에 모듈 내부에서는 Clock에 동기화하여 read, write를 진행하지 않는다.

```
module ALUControl(ALUCnt,AluOp,funct3,funct7);      // Takes in I
output reg  [3:0] ALUCnt;
input       [2:0] funct3;
input       [6:0] funct7;
input       [1:0] AluOp;

/// AluOp -> divide by type of instruction

always@(AluOp,funct3,funct7)    begin
// funct7 -> [31:25]
// funct7 -> [14:12]

        case(AluOp)
            2'b00 : // LW or SW
                ALUCnt = 4'b0010;

            2'b01 :
                begin
                    case(funct3)
                        3'b000: ALUCnt=4'b0110; //Beq
                        3'b001: ALUCnt=4'b0110; //Bne
                        3'b100: ALUCnt=4'b0111; //Blt
                        3'b101: ALUCnt=4'b0111; //Bge
                        default: ALUCnt=4'b0010; //Jalr
                    endcase
                end

            2'b10 : // R-Type Function 3 and 7 defines ALU mode
                begin
                    case(funct3)
                        3'b000 :
                                case (funct7)
                                7'b0000000: ALUCnt = 4'b0010;   // ADD
                                7'b0100000: ALUCnt = 4'b0110;   // SUB
                                    default : ALUCnt = 4'bZZZZ;
                                endcase
                        3'b001 : ALUCnt = 4'b1101;  // SLL
                        3'b100 : ALUCnt = 4'b1100;  //XOR
                        3'b101 :
                                case (funct7)
                                7'b0000000: ALUCnt = 4'b1110;// SRL
                                7'b0100000: ALUCnt = 4'b1000;// SRA
                                    default : ALUCnt = 4'bZZZZ;
                                endcase
                        3'b110 : ALUCnt = 4'b0001;  //OR
                        3'b111 : ALUCnt = 4'b0000;  //AND

                        default:
                            ALUCnt = 4'bZZZZ;
                    endcase
                end
            2'b11:  // When Source 2 is Imm Data
                begin
                    case(funct3)
                        3'b000 : ALUCnt = 4'b0010;  // ADDI
                        3'b001 : ALUCnt = 4'b1101;  // SLLI
                        3'b100 : ALUCnt = 4'b1100;  //XORI
                        3'b101 :
                                case (funct7)
                                7'b0000000: ALUCnt = 4'b1110;// SRLI
                                7'b0100000: ALUCnt = 4'b1000;// SRAI
                                    default : ALUCnt = 4'bZZZZ;
                                endcase
                        3'b110 : ALUCnt = 4'b0001;  //ORI
                        3'b111 : ALUCnt = 4'b0000;  //ANDI

                        default:
                            ALUCnt = 4'bZZZZ;
                    endcase
                end
        endcase
    end
endmodule
```

Fig 18. ALU Controller

Instruction	ALUops	Func7	Func3	ALUcnt
AND	10		111	0000
OR	10		110	0001
ADD	10		000	0010
SUB	10		000	0110
SLT	01			0111
XOR	10		100	1100
sll	10		001	1101
srl	10	0000000	101	1110
sra	10	0100000	101	1000
LW / SW	00		010	0010
Beq	01		000	0110
Bne	01		001	0110
Bge	01		101	0111
blt	01		100	0111
jalr	01		000	0010
addi	11		000	0010
slli	11		001	1101
xori	11		100	1100
srli	11		101	1110
srai	11		101	1000
ori	11		110	0001
andi	11		111	0000

```
module ALU(Zero, ALUresult, A, B, AluOp);
    input       [31:0]  A,B;
    input       [ 3:0]  AluOp;
    output          Zero;
    output  reg [31:0]  ALUresult;

always@(AluOp,A,B)
    begin
        case (AluOp)
            4'b0000 : ALUresult<= A&B;          //AND
            4'b0001 : ALUresult<= A|B;      //OR
            4'b0010 : ALUresult<= A+B;          //ADD
            4'b0110 : ALUresult<= A-B;      //SUB
            4'b0111 : ALUresult<= (A<B)?32'd1:32'd0;    // SLT
            4'b1100 : ALUresult<= (A^B);    //XOR
            4'b1101 : ALUresult<= A<<B;     //sll
            4'b1110 : ALUresult<= A>>B;         //srl
            4'b1000 : ALUresult<= A>>>B;    //sra

            default : ALUresult <=0;
        endcase
    end

assign Zero = (ALUresult==0);

endmodule
```

Fig 19-20. Instruction에 따른 Signal 분류 & ALU Module

해당 모듈은 Control 모듈에서 정해진 ALUOP 과 instruction 내 존재하는 Funct3, funct7의 비트를 이용해 ALU에서 진행해야 할 연산 종류를 결정하게 된다.

Funct3은 Instruction의 [14:12] bits이고 Funct7은 Instruction의 [31:25] bits이다.

해당 비트들과 Control에서 정해진 ALUOP에 따라 연산 종류를 결정한다.

연산 종류를 결정하는 기준은 Fig 0 표를 기준으로 결정하게 된다.

해당 모듈은 ALU Control에서 나오는 신호를 기반으로 연산이 이뤄지게 된다.

피연산자 2개를 받고 ALU Control에서 오는 신호를 기반으로 연산을 수행하고 연산결과를 출력한다.

작업에 따른 전체 작업 Design

```
// StallIF -> instruction Fetch level Stalling
always@(posedge Clk or negedge Rst) begin
        if(~Rst)
            PCreg   <= 32'd0;
        else if(stallIF==1'b0)                // Check for IF Stall
            PCreg   <= PCin ;                 // Update PC at posedge CLK1
        else
            PCreg   <= PCreg;                // for stall process, doens't update the  PC
end



// IF
// process: UPDATE PC VALUE -> Using instruction memory , update instruction, -> set next PC (PC+4 or Branch or Jal)

assign PC=PCreg; // PCreg has the current PC value
assign PCSrc = (((zero_MEM^Branch_MEM_0)||Link_MEM) && Branch_MEM_1) ; // PCSrc -> Branch , jump process.

    Add     PCAddressIncrement(PC_4,PC,32'd4);                  // Adder for PC increment PC_4=PC+4

    InstructionMemoryFile IMF (PC,Instruction,Clk,Rst); // Instruction Memory

    Mux2    PCAddressSel(PCin,PC_4,OffsetAddress,PCSrc);      // Next Address Selection 32 bit wide 2X1 Mux
```

Fig 21. Instruction Fetch RTL Design

IF 작업에서의 RTL Design에 해당하게 된다.

해당 과정에서는 처음 PC값을 설정하는 부분이 존재한다.

현재 PC값에 4byte를 더해주는 PCAddressIncrement 모듈이 존재하고 해당 과정은 clock에 동기화되지 않고 작동한다.

또한 Instruction Memory File에서 PC값에 따라 Instruction을 가져오는 Instruction Memory File 모듈이 존재한다.

해당 과정 또한 clock에 동기화되지 않고 작동하게 된다.

다음으로 다음 PC값을 결정지을 PCAddressSel 작업이 존재한다. 해당 과정을 통해 branch가 일어날지 아니면 현재 PC값에 4를 더한 값을 주소값으로 가질지 정해진다.

```
always@(posedge Clk or negedge Rst) begin

        if(~Rst) begin
            IF_ID_pipereg <= 96'd0;
        end
        else begin
          if(stallID==1'b0) begin
            IF_ID_pipereg[31: 0] <= Instruction;      /
            IF_ID_pipereg[63:32] <= PC;               //
            IF_ID_pipereg[95:64] <= PC_4;            //
          end
          else begin
              if(flushIF==1'b1) begin
                  IF_ID_pipereg  <= 96'b0;
              end
              else begin
                  IF_ID_pipereg <= IF_ID_pipereg;
              end
          end
    end
end
```

```
assign Instruction_ID   = IF_ID_pipereg[31:0];     // Extract Instruction
assign Opcode           = IF_ID_pipereg[6:0];      // Extract Opcode
assign funct3_ID        = IF_ID_pipereg[14:12]; //Extract Funct3 -> for addi or subi, lw, s
assign Rs2_ID           = IF_ID_pipereg[24:20];  // Extract reg select bits for rs1
assign Rs1_ID           = IF_ID_pipereg[19:15]; // Extract reg select bits for rs2
assign Rd               = IF_ID_pipereg[11: 7];// Extract reg select bits for rd

// ControlWire for FlushID, , NOP -> for stalling ( NO OPERATION) -> Means no enter to proc


// for stall or flush process
assign ControlWire1= (NOP || flushID) ? 12'bXXXXXXXXXXX:{Branch1,Lui,ALUOp,ALUsrc,
        AddSel,Link,Branch0,MemWrite,MemRead,RegWrite,MemtoReg};

    Control ControlDecoder(Opcode,funct3_ID,ALUsrc,MemtoReg,RegWrite,
              MemRead,MemWrite,AddSel,Link,Branch1,Branch0,ALUOp,Lui);
    //  Decodes instructions in ID stage and forwards the control signals to other stages
    RegisterFile GPR(data1,data2,Rs1_ID,Rs2_ID,rd_WB,writeData_WB,Clk,Rst,RegWrite_WB);
    //  General Purpose Register File x0-x31, two read ports and a write port
    ImGen   ImmediateGen(Immediate_ID,Instruction_ID);
    //  Generates 32 bit Immediate value as per instruction
```

Fig 22. Instruction Decode Module

해당 모듈은 입력으로 Instruction , PC, PC+4 값을 가지게 된다. 이 값들은 Instruction Fetch 과정의 결과 값들로 clock에 동기화되어 입력으로 들어온다.

Instruction 값을 통해 Control 모듈의 입력을 주게 된다.

FlushIF의 입력을 통해 Branch에 따른 flush 작업이 진행된다. 해당 과정이 진행되면

Instruction Decode 과정내 모듈들은 전부 초기화 된다. 만약 Stall 이 발생하게 된다면 stall에는 이전에 실행하던 Instruction을 그대로 유지하여 같은 작업이 한 클럭 더 실행되게 된다. 이는 Bubble과 같은 작업이라 할 수 있다.

Control 모듈로 원하는 신호들을 다 정하게 된다. 해당 신호들은 ControlWire1 변수에 clock과 상관없이 배치된다.

ControlWire1의 값은 NOP 혹은 Flush에서 Invalid값으로 변하게 되는데, 이는 Stall 및 Flush에서 control 값을 Invalid로 바꿔 해당 연산을 무효화 하기 위한 방안이다.

또한 RegisterFile에서는 Instruction내 rs1, rs2 값에 따라 해당 레지스터의 값을 출력하게 된다.

ImGen 모듈에서는 Immediate Generator 역할이 수행되며 해당 작업을 통해 Instruction 내 존재하는 OFFSET 과 IMMEDIATE가 32 bit 체계로 출력된다.

```
always @(posedge Clk or negedge Rst)
 begin
    if(~Rst)
        ID_EX_pipereg <= 197'd0;
    else
    begin
        ID_EX_pipereg[ 31: 0 ] <= IF_ID_pipereg[63:32];       // Forward PC.
        ID_EX_pipereg[ 63:32 ] <= data1 ;                     //  Forward Rs1 Data
        ID_EX_pipereg[ 95:64 ] <= data2 ;                     //  Forward Rs2 Data
        ID_EX_pipereg[127:96 ] <= Immediate_ID ;              //  Forward Immediate
        ID_EX_pipereg[159:128] <= IF_ID_pipereg[95:64];       //  Forward PC+4
        ID_EX_pipereg[164:160] <= Rd ;                        //  Forward Rd Select
        ID_EX_pipereg[176:165] <= ControlWire1 ;              //  Forward Control S
        ID_EX_pipereg[186:177] <= {Instruction_ID[14:12],Instruction_ID[31:25]} ;/
        ID_EX_pipereg[196:187] <= {Rs1_ID,Rs2_ID};    // Store for Forwarding
    end
 end

assign Imm32=ID_EX_pipereg[127:96];
assign PC_EX=ID_EX_pipereg[ 31: 0];

assign data1_Ex = ID_EX_pipereg[ 63:32 ];
assign data2_Ex = ID_EX_pipereg[ 95:64 ];
```

```
assign ALUsrc_Ex    =ID_EX_pipereg[172];
assign AluOp_Ex     =ID_EX_pipereg[174:173];
assign Lui_Ex       =ID_EX_pipereg[175];
assign Branch1_Ex    =ID_EX_pipereg[176];
assign RegWrite_EX  =ID_EX_pipereg[166];
assign Rd_EX        =ID_EX_pipereg[164:160];

assign funct3=ID_EX_pipereg[186:184];
assign funct7=ID_EX_pipereg[183:177];

assign ControlWire2= (flushEX) ? 9'bxxxxxxxxx:
        {Branch1_Ex,Zero,ID_EX_pipereg[171:165]};
    // flush EX deasserts control introducing Bubbles/No operat

    // PC_EX -> current PC value.  imme -> 1bit가 생략되어있다.
    Add AddressAdder(BrAdd,PC_EX,{Imm32[30:0],1'b0});
    // Adder for Computing Branch Addresses (Imm32 bits are lef
    // InA -> data1_Ex -> register rs1 value.  ALUAddress ->  A
    Mux4 BusA1(A1,data1_Ex,ALUAddress,writeData_WB,32'd0,InA);
    Mux4 BusB1(B1,data2_Ex,ALUAddress,writeData_WB,32'd0,InB);/

    Mux2 BusA0(A,A1,32'd0,Lui_Ex);        // Mux : Loads Rs1 Dat
    Mux2 BusB0(B,B1,Imm32,ALUsrc_Ex);    // Mux : Selects betwee

    ALU  ALUUnit(Zero,ALUresult,A,B,ALUCnt);              // ALU
    ALUControl ALUCtrl(ALUCnt,AluOp_Ex,funct3,funct7);  // 2nd
```

Fig 23. Execute Module

해당 과정에서는 이전의 IF_ID PipeRegister 값을 토대로 rs1의 data, rs2의 data, 현재 PC값 등 등 Execute 단계 및 MA, WB 단계에 필요한 데이터들을 입력으로 받게 된다.

다음으로 이렇게 받은 데이터를 기준으로 Controlwire2를 구성하게 된다.

Controlwire2는 ID단계내 Controlwire1 단계와 동일하게 NOP(NOT OPERATION) 및 FLUSH 단계를 위해 존재하고 또한 다음 단계에 필요한 신호들을 전달하기 위해 존재한다.

Execute 단계에서는 먼저 분기가 발생할 수 있기에 분기 주소를 구해준다.

분기 주소는 일반적으로 Jal, Beq, Blt 등 명령어에 의해 발생하는데 해당 명령어들은 OFFSET의 LSB가 생략되어 있다. 그렇기에 BranchAddr를 구할 때는 2를 곱해주어 LSB 값을 메꿔준다.

다음으로 ALU Unit에 들어갈 피연산자들을 구해야 한다. BusA1, BusB1은 이전에 말했듯 DATA FORWARD를 고려한 하드웨어이다.

해당 과정을 지나면 Lui 혹은 ALUsrc 신호에 따라 피연산자들이 결정된다.

피연산자가 결정되면 ALU CONTROL에 따라 ALU내 연산이 결정되게 된다.

ALU는 ALU CONTROL의 출력 신호 값에 따라 피연산자들의 연산을 수행하게 된다.

```
always@(posedge Clk or negedge Rst)
begin
    if(~Rst)
        EX_MEM_pipereg <=142'd0;
    else
        begin
        EX_MEM_pipereg [ 31: 0 ] <= ALUresult;                    // ALU Result
        EX_MEM_pipereg [ 63: 32] <= BrAdd;                        // PC+Offset ,Branch Address
        EX_MEM_pipereg [ 95: 64] <= B1;                           // Rs2 data to write to Memory
        EX_MEM_pipereg [127: 96] <= ID_EX_pipereg[159:128];       // PC+4
        EX_MEM_pipereg [132:128] <= Rd_EX;                        // RD_EX
        EX_MEM_pipereg [141:133] <= ControlWire2;                 // Control Signals for further stages
    end
end


assign ControlWire3 = EX_MEM_pipereg[134:133];

assign BranchOffset = EX_MEM_pipereg [63:32];    // Branch address = PC+ Shifted Immediate
assign ALUAddress   = EX_MEM_pipereg [31: 0];    // ALU address     = Reg + Immediate , ALU result value
assign WriteData_MEM= EX_MEM_pipereg [95:64];  // RS2 Data for writing to memory ,RS1 -> based address in SW instruction
assign PCLink =EX_MEM_pipereg [127: 96]; // PC+4 value
assign Rd_MEM =EX_MEM_pipereg [132:128];   // RD_REG NUMBER OF CURRENT Instruction

assign Branch_MEM_1 = EX_MEM_pipereg[141]; // Branch or not
assign zero_MEM     = EX_MEM_pipereg[140];  // Used for Branches
assign AddSel_MEM   = EX_MEM_pipereg[139];  // change PC value using pc or reg
assign Link_MEM     = EX_MEM_pipereg[138];   // Set for Unconditional Jumps
assign Branch_MEM_0 = EX_MEM_pipereg[137];  //  Set for Branches
assign memWrite_MEM = EX_MEM_pipereg[136];
assign memRead_MEM  = EX_MEM_pipereg[135];
assign RegWrite_MEM = EX_MEM_pipereg[134];


    Mux2 AddressSel(OffsetAddress,BranchOffset,ALUAddress,AddSel_MEM); // Selects between PC Offset/Reg Offset
    Mux2 LinkSel(writeBack_MEM,ALUAddress,PCLink,Link_MEM);           // Selects between writing back PC+4/ALUOut
    DataMemoryFile DMF(ReadData,ALUAddress,WriteData_MEM,memWrite_MEM,memRead_MEM,Clk,Rst);
```

Fig 24. MA RTL DESIGN SECTION

Memory Access작업의 RTL design을 살펴보면 Execute 단계의 결과물을 토대로 작업이 진행됨을 확인할 수 있다.

AddressSel에서는 분기할 때 주소를 어떻게 지정할지 결정하게 된다.

BranchOffset은 현재 PC값에 OFFSET을 더한 주소 값에 해당하며 ALUaddress는 Register를 기준으로 OFFSET을 더한 값을 의미한다.

SB-type 명령어나 UJ-type 명령어는 주로 BranchOffset을 사용하지만 Jalr같은 레지스터 기반으로 명령어 주소를 결정하는 명령어는 ALUaddress를 사용하게 된다.

Jal 혹은 Jalr 명령어와 같이 현재 PC값에 +4한 값을 레지스터에 저장해주어야 한다. 이때 rd라는 저장 레지스터에 저장하는데, ALU Result를 저장허간 PC+4를 저장하게 되는 것이다.

이를 수행하는 모듈이 LinkSel이다.

DataMemoryFile은 Data read 및 Data write가 존재할 때, 사용된다. Read 동작 및 write 동작 모두 clock에 동기화되지 않는다.

하지만 EX-MA로 넘어오는 데이터가 CLOCK에 동기화되어 넘어오기에 전체 과정은 clock에 동기화되어 작동한다고 할 수 있다.

```
always @(posedge Clk or negedge Rst)
begin
    if(~Rst)
    MEM_WB_pipereg <= 70'd0;
    else
     begin
        MEM_WB_pipereg[31:0 ] <= ReadData; // Data read from Memory
        MEM_WB_pipereg[63:32] <= writeBack_MEM; // Data from ALU / Link Reg
        MEM_WB_pipereg[68:64] <= Rd_MEM; // Write Data Select Register
        MEM_WB_pipereg[70:69] <= ControlWire3; // Control Signals
    end
end


assign RegWrite_WB  = MEM_WB_pipereg[70];
assign MemtoReg_WB  = MEM_WB_pipereg[69];     // MemtoReg -> using memory for writing Register
assign readData_WB  = MEM_WB_pipereg[31: 0]; // from meemory data by processing LW
assign writeBack_WB = MEM_WB_pipereg[63:32];  // From ALU result
assign rd_WB        = MEM_WB_pipereg[68:64]; // number of destination Register

    Mux2 WriteBackSel(writeData_WB,readData_WB,writeBack_WB,MemtoReg_WB);
```

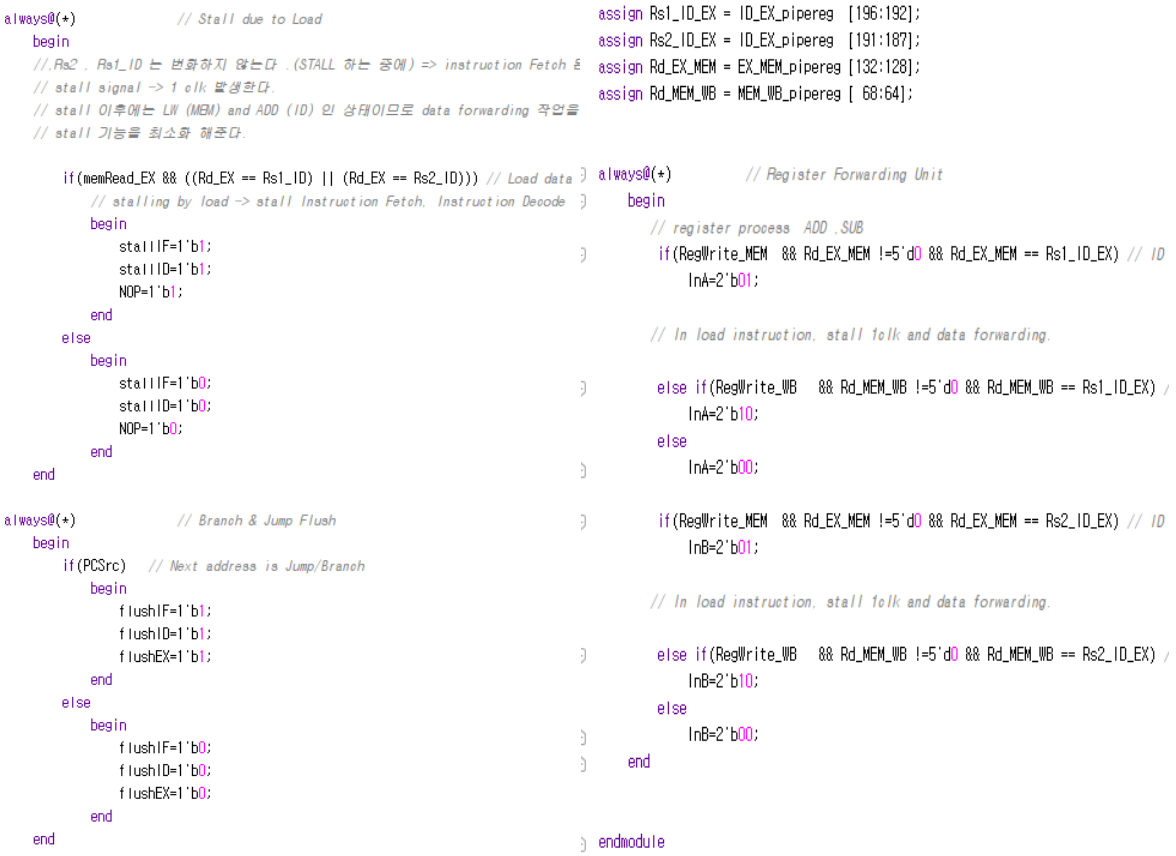
Fig 25. Writeback RTL DESIGN SECTION

해당 RTL DESIGN에서는 MEM_WB PIPEREG 내 데이터를 통해 모듈의 작업이 진행된다.

WriteBackSel은 Register file로 전달되어 Writeback될 데이터가 무엇인지 결정짓게 된다.

MemtoReg Signal이 존재할 경우 Write data는 메모리에서 읽어온 데이터일 것이고 그것이 아니라면 ALU의 결과값에 해당하게 될 것이다.

Writeback signal이 전달될 Register File은 clock에 동기화되지 않고 신호가 오면 바로 해당 작업을 수행하게 되는데, 이 또한 MEM_WB PIPEREG에 모든 신호 및 데이터들이 동기화되어 있기에 바로 작업해도 무관한 것이다.



```
always@(*)              // Stall due to Load
    begin
    //,Rs2 , Rs1_ID 는 변화하지 않는다 .(STALL 하는 중에) => instruction Fetch 은
    // stall signal -> 1 clk 발생한다.
    // stall 이후에는 LW (MEM) and ADD (ID) 인 상태이므로 data forwarding 작업을
    // stall 기능을 최소화 해준다.

        if(memRead_EX && ((Rd_EX == Rs1_ID) || (Rd_EX == Rs2_ID))) // Load data
            // stalling by load -> stall Instruction Fetch, Instruction Decode
            begin
                stallIF=1'b1;
                stallID=1'b1;
                NOP=1'b1;
            end
        else
            begin
                stallIF=1'b0;
                stallID=1'b0;
                NOP=1'b0;
            end
    end

always@(*)              // Branch & Jump Flush
    begin
        if(PCSrc)   // Next address is Jump/Branch
            begin
                flushIF=1'b1;
                flushID=1'b1;
                flushEX=1'b1;
            end
        else
            begin
                flushIF=1'b0;
                flushID=1'b0;
                flushEX=1'b0;
            end
    end
```

```
assign Rs1_ID_EX = ID_EX_pipereg  [196:192];
assign Rs2_ID_EX = ID_EX_pipereg  [191:187];
assign Rd_EX_MEM = EX_MEM_pipereg [132:128];
assign Rd_MEM_WB = MEM_WB_pipereg [ 68:64];


always@(*)          // Register Forwarding Unit
    begin
        // register process  ADD ,SUB
        if(RegWrite_MEM  && Rd_EX_MEM !=5'd0 && Rd_EX_MEM == Rs1_ID_EX) // ID
            InA=2'b01;

        // In load instruction, stall 1clk and data forwarding.

        else if(RegWrite_WB   && Rd_MEM_WB !=5'd0 && Rd_MEM_WB == Rs1_ID_EX)
            InA=2'b10;
        else
            InA=2'b00;

        if(RegWrite_MEM  && Rd_EX_MEM !=5'd0 && Rd_EX_MEM == Rs2_ID_EX) // ID
            InB=2'b01;

        // In load instruction, stall 1clk and data forwarding.

        else if(RegWrite_WB   && Rd_MEM_WB !=5'd0 && Rd_MEM_WB == Rs2_ID_EX)
            InB=2'b10;
        else
            InB=2'b00;
    end


endmodule
```

Fig 26. Stall & Data Forward & Flush

위 RTL DESGIN은 Stall, Data Forward 그리고 Flush를 나타낸다.

먼저 Stall이 발생하기 위한 조건을 살펴보면 ID-EX, ID-MEM간의 Data Hazard 발생을 의미한다.

이를 확인하기 위해 memRead_EX (ID-EX PipeReg) 의 값과 Rd_EX(ID-EX PipeReg)와 Rs1,2_ID (IF-ID PipeReg)를 비교해야 한다.

만약 이 신호들의 논리연산이 Load 명령에 따른 Data Hazard를 발생시킨다면 Stall을 발생시켜 다음 명령어의 작업을 멈추고 현재 명령어 작업에서 LOAD 작업을 진행시킨다.

다음으로 Stall 이후 load된 데이터를 data forward를 통해 다음 명령어의 작업 단계Execute 단계에 있는 Mux로 보내게 된다.

만약 ID-EX간의 data Hazard가 발생한 경우 해당 경우에서는 execute 과정의 결과를 바로 Data forward를 통해 다음 Execute의 피연산자로 들어가게 되어 Data Hazard를 방지한다.

만약 분기가 발생한 상황이라면 분기 이전에 파이프라인 내 실행중인 명령어를 초기화 하기 위해 각 단계내 하드웨어로 들어가는 Controlwire1, Controlwire0를 Invalid로 만들어주고 PC값을 다음 클럭에서 새로운 분기 주소 값으로 받게 된다.

4. Custom RTL Design에 대한 시뮬레이션

A. Non-Hazard Instruction Sequence

Hazard가 발생하지 않는 상황에서 명령어의 처리가 5단계로 나눠 처리되는 지 확인한다. 이 시뮬레이션을 통해 개별적 명령어에 대한 처리가 정확히 이뤄지는 지 확인 가능하다.

```
00402083 // lw $1 , ($0)4
00202423 // sw $2, ($0)8
00000233 // add $4, $0, $0
002182B3 // add $5, $2, $3
40218333 // sub $6, $2, $3
00208393 // addi $7, $1, 2
```

Fig 27. Simulation Instructions

해당 Instruction Sequence는 서로에 대한 Hazard가 발생하지 않는 명령어들이다.

이를 먼저 실행하고 시뮬레이션으로 확인하여 제대로 RTL Design이 이뤄지는 지 확인한다.

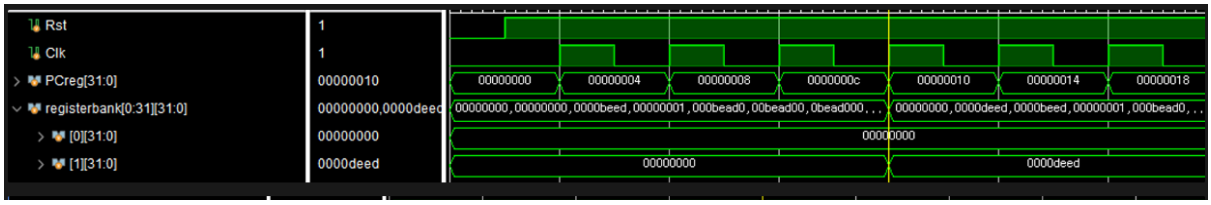


Fig 28. 명령어 실행에 대한 Program Counter 증가

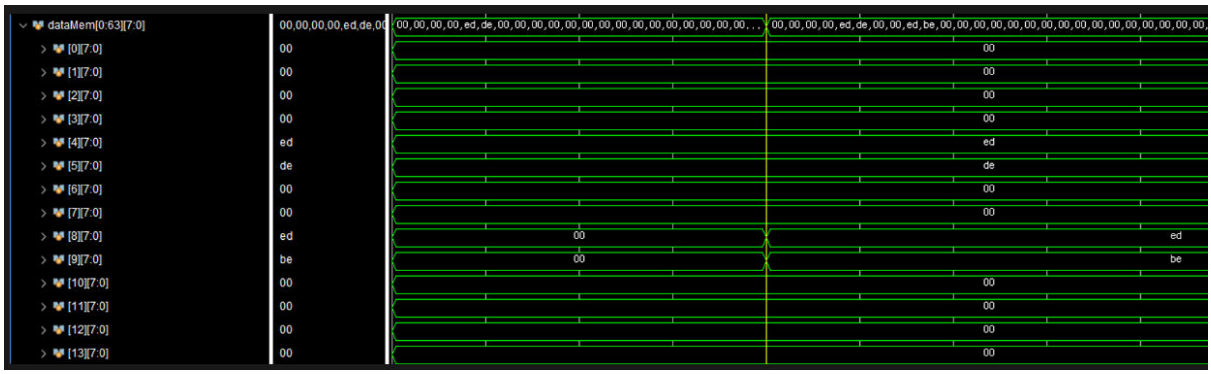


Fig 29. SW 명령어에 따른 Memory File 내 Data의 변화

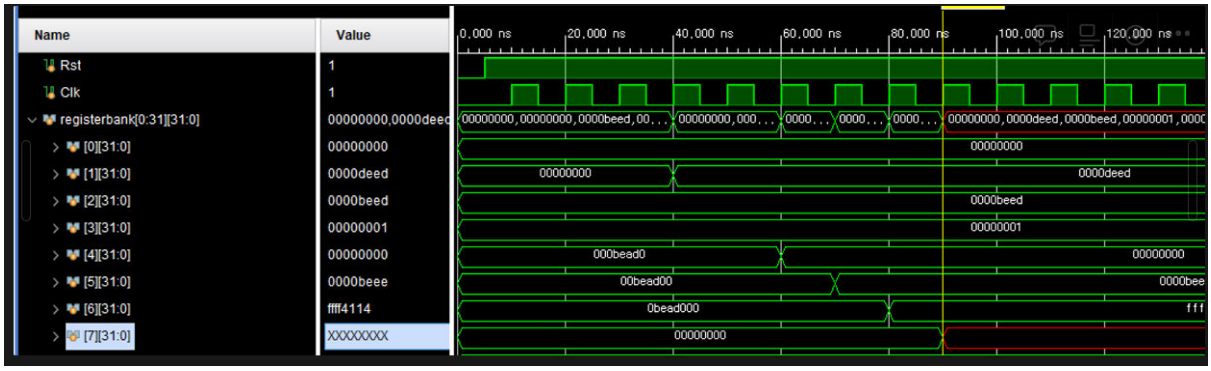


Fig 30. Load 및 산술 연산 명령어에 따른 Register File내 Data의 변화

Fig 28, 29, 30을 참고하면 각각의 명령어 실행마다 한 클럭의 차이로 Writeback 작업이 이뤄지는 것을 확인할 수 있고 순차적 명령어 실행에 대한 예측가능한 결과와 시뮬레이션의 결과가 일치함을 알 수 있다.

B. Hazard Instruction Sequence

```
83204000 // lw $1,($0)4
63040000   // beq $0, $0, 4, PC+8// 0000_0000_0000_0000_0000_0100_0110_0011 flush
23242000 // sw $2,($0)8
33020000 // add $4,$0,$0
B3824100 // add $5,$4,$3  data forward
03214000    // lw $2 , ($0)4 stall and data forward
33832140 // sub $6,$2,$3
93832000 // addi $7,$1,2
```

Fig 31. Simulation Instructions

해당 Instruction Sequence는 Data Hazard, Control Hazard를 가지고 있다.

먼저 Beq 명령어의 실행에 따른 Control Hazard가 발생하며, 해당 Control Hazard에 따른 Flush가 발생해야 한다.

Flush를 통해 분기 이전에 실행되던 명령어에 대한 초기화가 진행된다.

다음으로 분기에 따른 SW 명령어의 실행은 이뤄지지 않는다.

산술 연산 명령어에서 Data Hazard 와 Load 명령어에 따른 Data Hazard가 발생한다.

산술 연산 명령어에서는 Stall 없이 Data forward를 통해 Hazard 처리가 이뤄난다. 하지만 Load 과정내 일어나는 Data Hazard는 한 클럭을 Stall 해주고 Data forward를 통해 Hazard를 처리해준다.

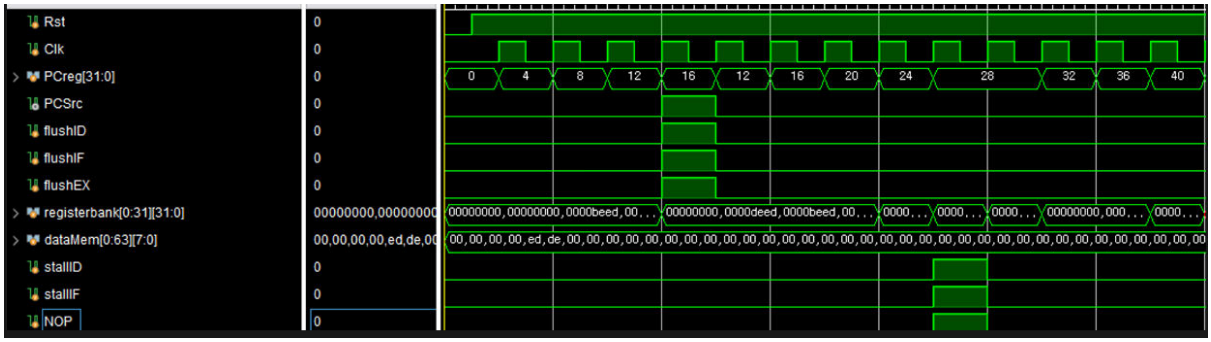


Fig 32. Hazard 처리에 따른 PC 값 변화 및 Stall, Flush Signal

Fig 32에서 보면 PCsrc에 따라 분기 결과가 결정되고 해당 결과에 따라 PC값이 변화하는 것을 확인할 수 있다.

Stall의 발생 또한 확인 가능하며 해당 Stall은 Load 명령 이후 산술 명령어에 따라 발생한 것이다.

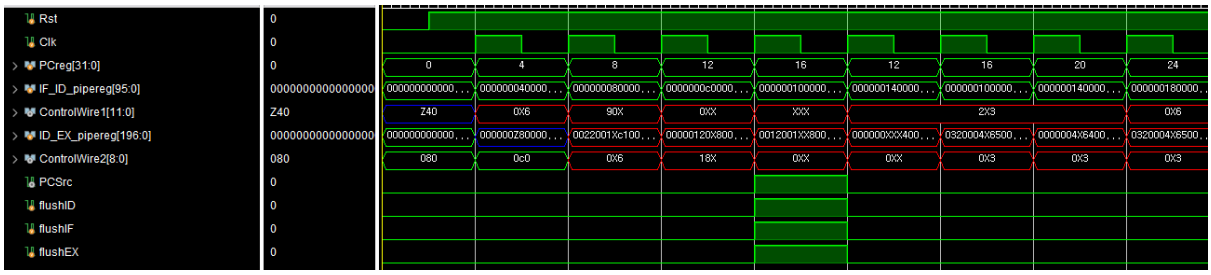


Fig 33. Branch Instruction에 따른 Flush 발생 및 PCsrc

Fig 33을 보면 Branch Instruction의 발생에 따라 Flush가 발생함을 알 수 있다.

Flush 발생은 MA 단계에서 확인 가능하다.

그 이유는 PCSrc 값은 EX 데이터의 결과값에 의존하기 때문이다. 해당 작업을 통해 Flush가 제대로 작동함을 알 수 있다.

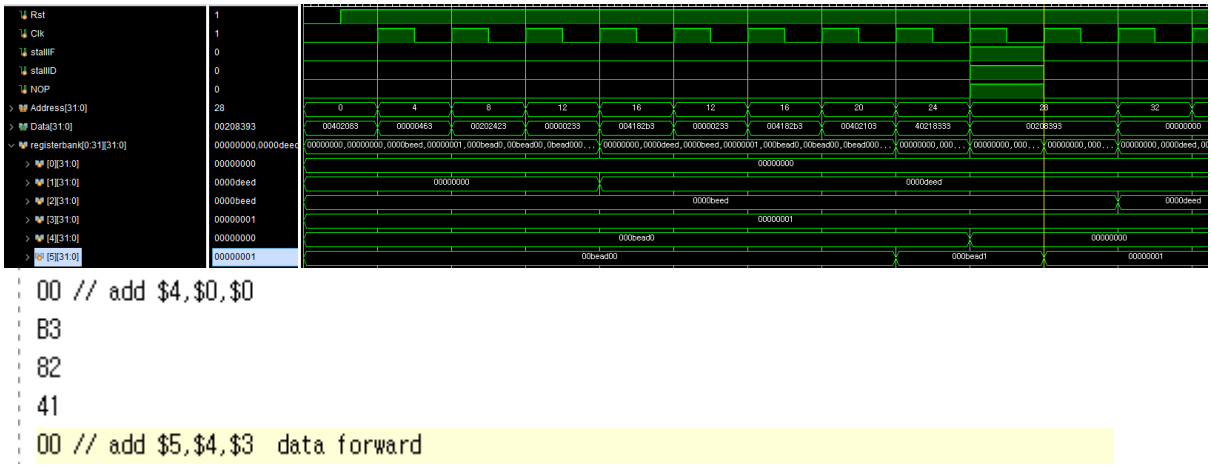


Fig 34. 산술 연산 명령어에 따른 Data Hazard 처리

해당 Fig 을 보면 산술 연산 명령어에 따른 Data Hazard 처리 결과를 볼 수 있다.

해당 결과에서 보면 원래 Register File 내 4번 레지스터의 값은 0 이 아닌 수를 가지고 있다. 하지만 add $4 $0 $0 의 명령어의 실행 결과를 통해 4번 레지스터의 값은 0 이 된다.

이를 Data Forward 방식을 반영하며 add $5, $4, $3의 결과는 000bead1이 아닌 00000001의 값을 가지게 된다.

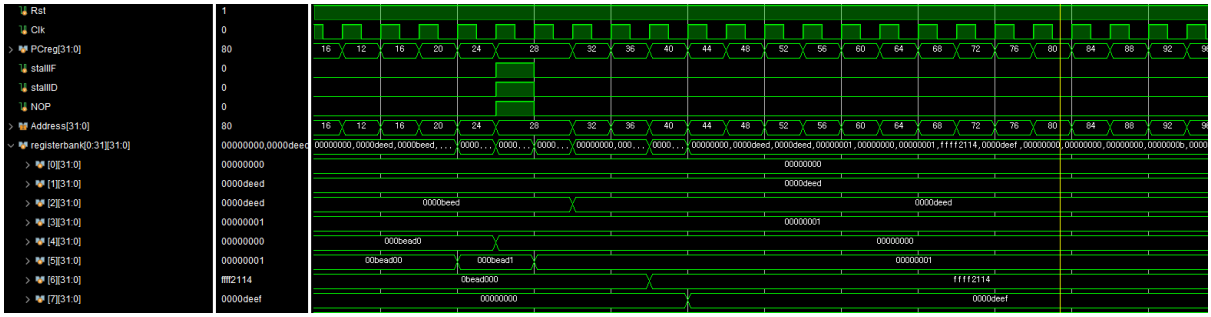


Fig 35. Load 명령어에 따른 Data Hazard 처리

Load 명령어에 의해 나타나는 Data Hazard 처리는 STALL 및 Data forward를 통해 처리가능하다. 먼저 STALL 처리 먼저보게 되면 Stall Signal이 존재할 때, PC값의 변화가 없는 것을 확인할 수 있다.

이에 따라 추가적 명령어 실행을 막게된다. 또한 NOP를 통해 Controlwire1~2 신호를 제거하여 단계내 하드웨어에서 아무 일도 발생하지 않도록 진행한다.

Stall이 되었기에 WriteBack 단계가 밀려 2번 레지스터의 저장 이후 두 클럭 뒤에 레지스터 6번의 저장이 이뤄짐을 확인할 수 있다.

해당 시뮬레이션들을 통해 Instruction Sequence에 따라 발생가능한 Hazard에 대한 처리가 제대로 됨을 확인할 수 있다.

참고 자료

- https://msyksphinz-self.github.io/riscv-isadoc/html/
- https://oilbeen.tistory.com/20
- https://passlab.github.io/CSCE513/notes/lecture08_RISCV_Impl_pipeline.pdf