CPU 구현 과제 최종 보고서

4조

제출일	2024. 06. 12	**이름**	김유진
과목	디지털 논리 회로	**이름**	안중선
		이름	정의진

[목차]

내용

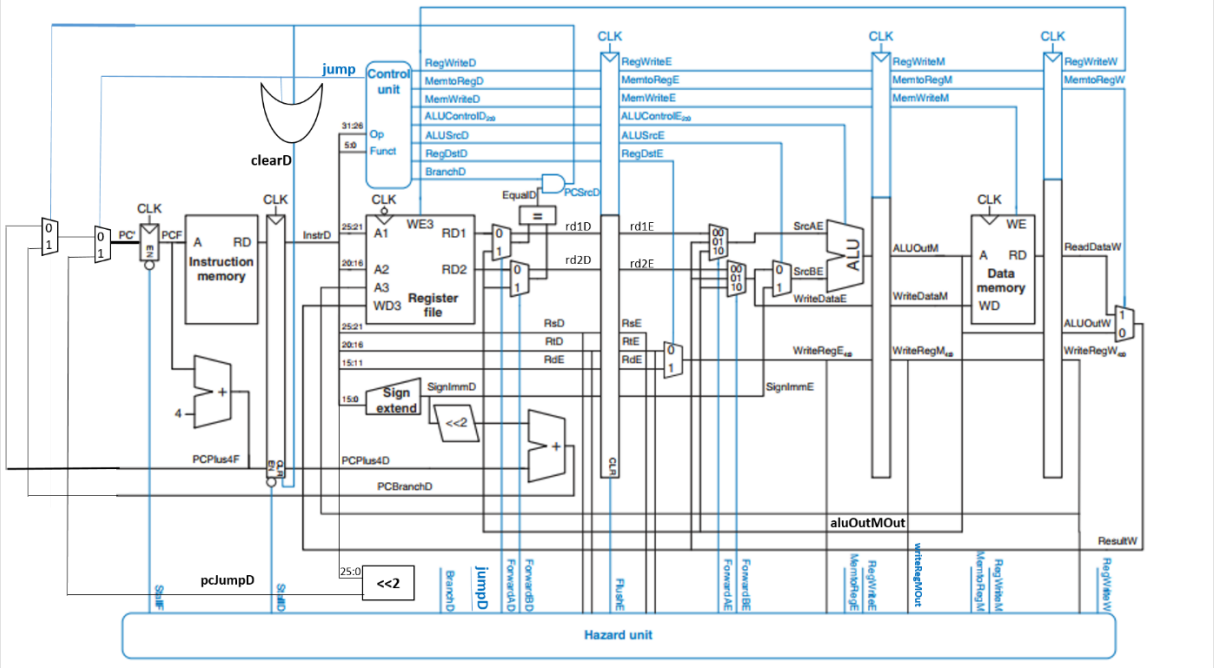
1. 서론

A. 프로젝트 개요(Project Overview)

이번 프로젝트의 목표는 파이프라인 구조를 갖춘 MIPS 프로세서를 설계하고 구현하는 것이다. MIPS(Microprocessor without Interlocked Pipeline Stages)는 간단하고 효율적인 명령어 집합 구조(ISA)를 가지고 있어, 교육 및 연구 목적으로 널리 사용되는 프로세서 아키텍처이다. 이 프로젝트에서는 MIPS 프로세서의 기본적인 동작 원리를 이해하고, 이를 바탕으로 파이프라인 구조를 구현하여 명령어 처리 속도를 향상시키는 것을 목표로 하였다. 파이프라인 구조를 통해 명령어들이 서로 다른 단계에서 동시에 처리되므로, 전체적인 처리 속도가 증가하게 된다. 이번 프로젝트에서는 파이프라인의 다섯 가지 주요 단계를 구현할 것이다.

1. **Fetch (명령어 인출) 단계**: 메모리에서 명령어를 가져옵니다.
2. **Decode (명령어 해독) 단계**: 명령어를 해독하여 필요한 제어 신호를 생성하고, 레지스터 파일에서 피연산자를 읽어옵니다.
3. **Execute (명령어 실행) 단계**: ALU 를 사용하여 연산을 수행합니다.
4. **Memory Access (메모리 접근) 단계**: 메모리 읽기 또는 쓰기 작업을 수행합니다.
5. **Write Back (결과 기록) 단계**: 연산 결과를 레지스터 파일에 저장합니다.

각 단계는 독립적으로 설계되어 있으며, 파이프라인 구조를 통해 병렬로 실행된다. 이를 통해 CPU 의 처리 성능을 크게 향상시킬 수 있다.



2. 설계 개요

A. 아키텍쳐 개요

이번 프로젝트에서 구현한 파이프라인 MIPS 프로세서는 명령어를 Fetch, Decode, Execute, Memory Access, WriteBack 하는 5 단계 파이프라인으로 구성된 간단한 CPU 이다. 각 단계는 독립적인 모듈로 구현되어 있으며, 명령어가 각 단계에서 병렬로 처리될 수 있도록 설계하였다. 전체 시스템 아키텍처는 다음과 같이 구성되어 있다.

시스템 구성 요소

DataPath
명렁어의 흐름

Fetch

-**ProgramCounter**: 현재 PC 값을 저장하고, 브랜치가 발생할 경우 브랜치 주소로 갱신한다.

-**FetchStage**: ProgramCounter에서 받아온 PC 값으로 명령어를 fetch하고, 다음 PC 값을 계산한다

Decode

-**DecodeRegister**: Fetch 단계에서 페치된 명령어와 PC 값을 임시 저장하고, 클럭에 동기되어 저장된 값을 출력한다.

-**DecodeStage**: DecodeRegister에서 받은 명령어를 해석하고, 레지스터 파일에서 데이터를 읽어오며, 브랜치 주소를 계산한다.

Excute

-**ExcuteRegister**: Decode 단계에서 해석된 데이터를 임시 저장하고, 클럭에 동기되어 저장된 값을 출력한다.

ExcuteStage: ExcuteRegister 에서 받은 데이터를 ALU 연산을 통해 처리한다.

Memory

MemoryRegister: Execute 단계에서 생성된 데이터를 임시 저장하고, 클럭에 동기되어 저장된 값을 출력한다.-

MemoryStage: MemoryRegister에서 받은 데이터를 통해 메모리 접근을 수행한다.

WriteBack

WriteBackRegister: Memory 단계에서 생성된 데이터를 임시 저장하고, 클럭에 동기되어 저장된 값을 출력한다

WriteBackStage: WriteBackRegister에서 받은 데이터를 최종 결과로 출력하고, 필요시 레지스터 파일을 갱신한다.

ControlUnit

OpcodeDecoder : DecodeStage로부터 Opcode를 받아서 그에 맞는 Control Signal을 DataPath로 전달하여 각 모듈에 분배하여 전체 시스템 Control

Control Signal(Output of ControlUnit)

branchD : DecodeStage - BranchDetect 모듈로 들어가서 PC_branch값 생성

flushD : branch시에 ExcuteRegister로 들어가 실행중인 명령어 flush 수행

RegWriteD : DecodeStage – Register_File로 들어가 WriteBackStage의 출력을 Register에 저장

MemWriteD : MemotyStage – Memory_Data로 들어가 write_enable 수행

MemToRegD : WriteBackStage로 들어가 alu_result와 MemReadData의 Mux select 신호 수행

immediateD : DecodeStage – Register_File로 들어가 srcdata2가 immediate value로 사용될 수 있도록 한다.

forwardD: DecodeStage - 데이터 포워딩 신호를 생성

alufunc : ExcuteStage – ALU로 들어가 ALU의 연산 제어 수행

HazardUnit

DataPath와 ControlUnit에서 input을 받아와서 Output으로 ControlUnit과 함께 Dataflow를 제어하는 signal을 출력한다.

Output signal

stallF, stallD:

각각 Fetch와 Decode의 Register 입력으로 들어가며 enable신호의 역할을 한다.

forwardA, forwardB:

명령어의 destadd가 다음 명령어의srcadd와 같다면 source1과 source2의 Data forward signal 역할을 한다.

flushD, flushE:

각각 Decode와 Excute의 Register 입력으로 들어가며 Register의 reset과 동일한 역할을 수행한다.

InstBranch:

Branch명령어가 들어왔으며 srcdata1과 srcdata2가 같지 않을 때 Branch signal로 동작하며 ProgramCounter와 DecodeStage의 입력으로 들어간다.

16bit Instruction Set

[4 : 4 : 4 : 4]

Opcode SrcAdd1 SrcAdd2(Imm) DestAdd

Opcode[3:0] : 제어신호(flag) 생성 – LDA(Imm), STA, IMM, BAF 총 3개의 제어신호 SET

LDA_imm = RegWrite, MemToReg, immediate 생성

- Memory_Data -> Register_File로 데이터를 옮기는 명령어

STA_imm = MemWrite 생성

- Register_File -> Memory_Data로 데이터를 옮기는 명령어

IMM = RegWrite, immediate 생성 (SrcAdd2 4bit를 Immediate Value로 사용 0 ~ 15)

- 즉시값을 사용하여 Register연산을 보다 효율적으로 이용
- IMM_add, IMM_sub, IMM_mul 로 연산 별로 세분화하였음.

BAF(Branch and Flush) = branch, flush 생성

- 명령어 이동을 가능하게하여 계산한 결과가 0이 아닐 때 LOOP문 구현

BAF_immsub : 레지스터와 immidate value끼리 뺄셈

BAF_regsub : 레시스터끼리 뺄셈 연산

SrcAdd1 : Register_File로 들어가 SrcData1를 출력하는 Register의 주소

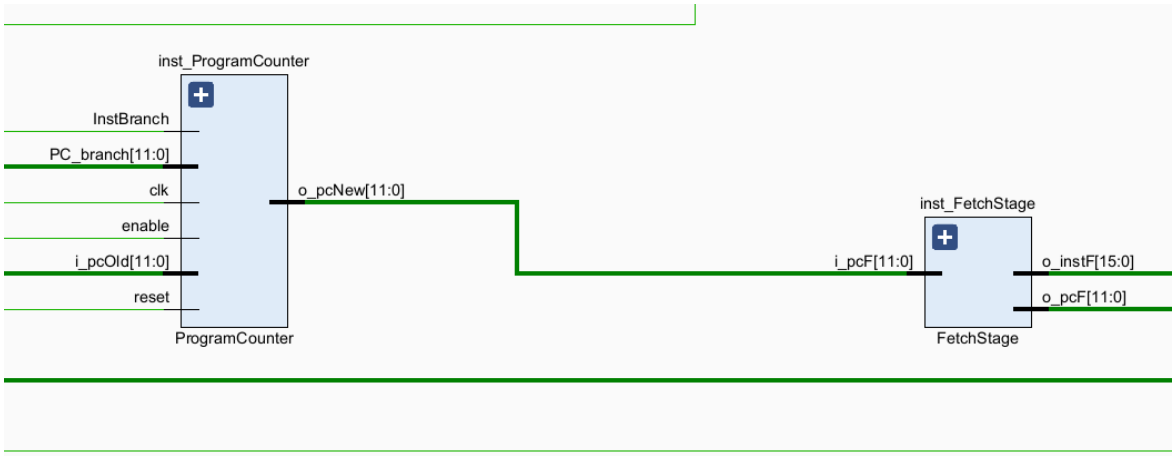
SrcAdd2 : RegisterFile로 들어가 SrcData2를 출력하는 레지스터의 주소 (Immediate 입력 시 SrcData2 =

DestAdd : Register_File과 Memory_Data에 들어가 데이터를 쓰고자하는 주소를 지정한다.

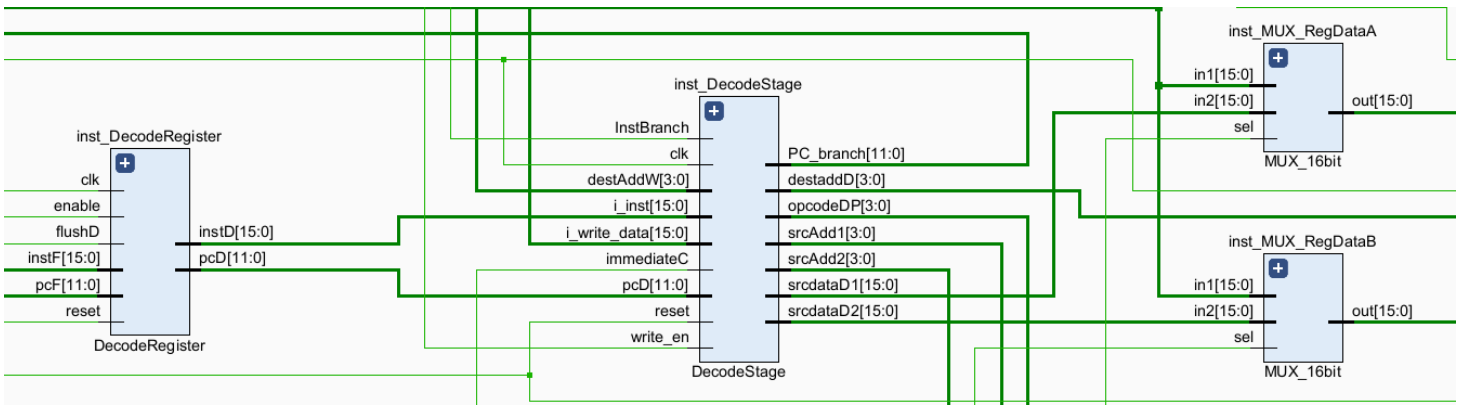
B. 데이터 및 제어 흐름

1. 명령어 처리 단계 Schmetic

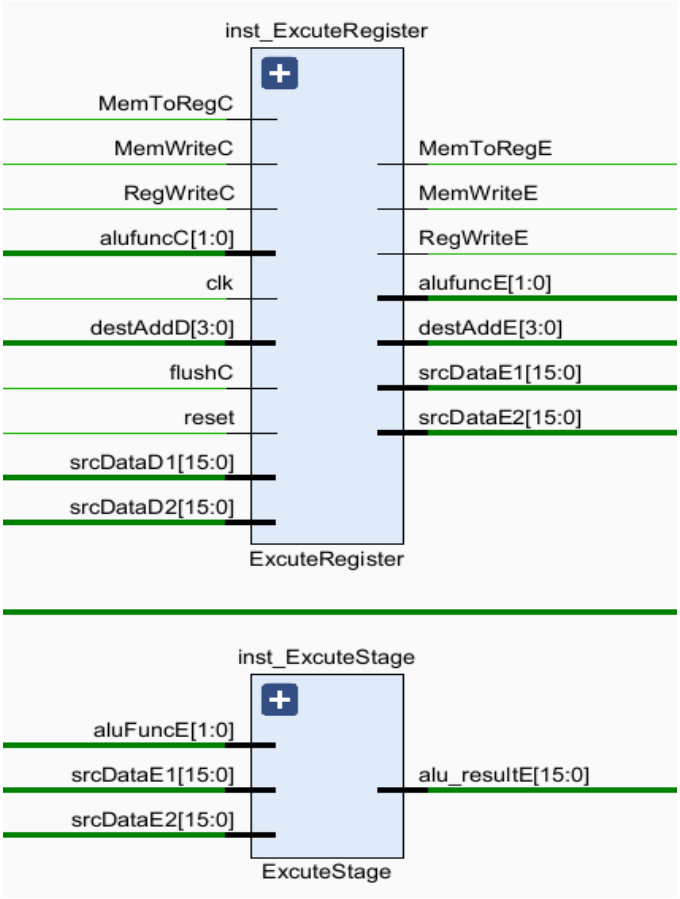
i. Fetch



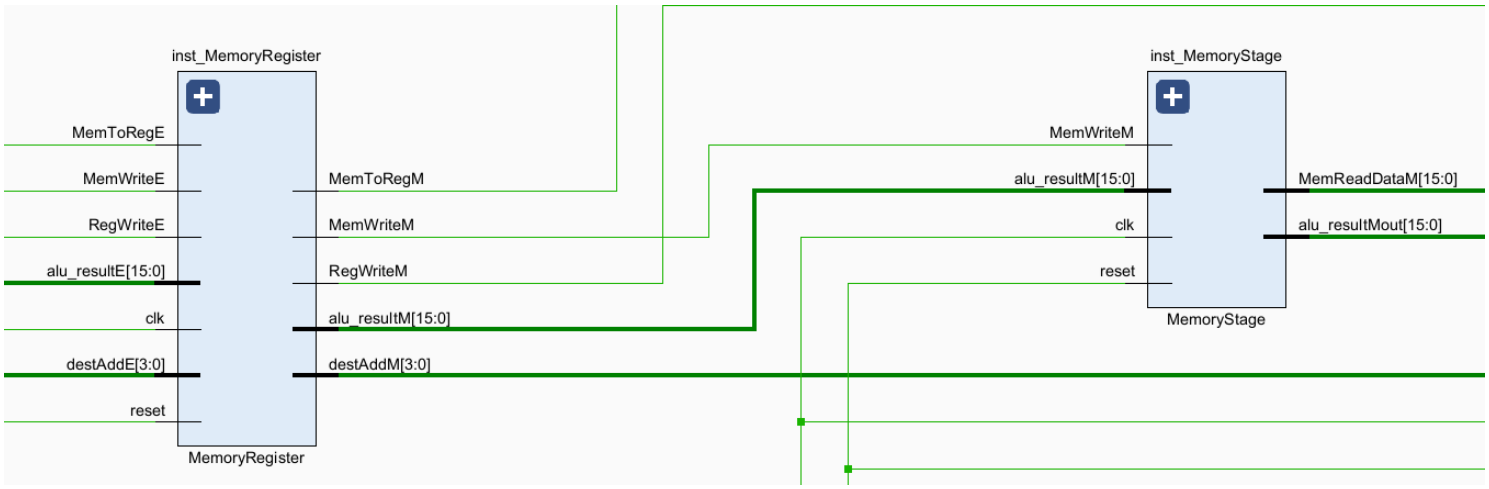
ii. Decode



iii. Execute



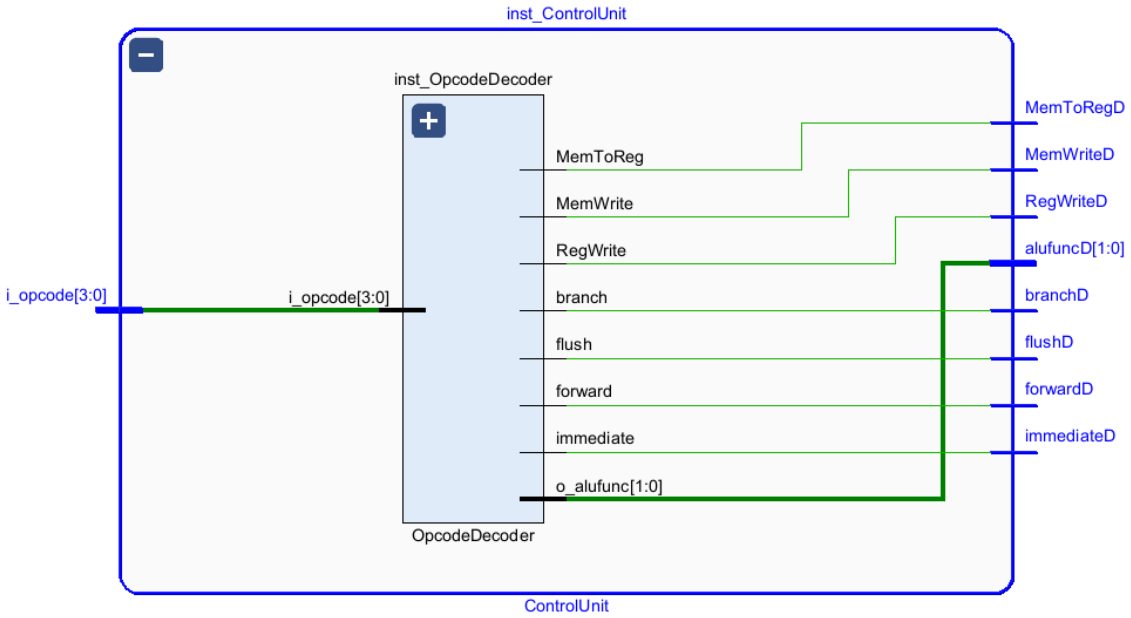
iv. Memory



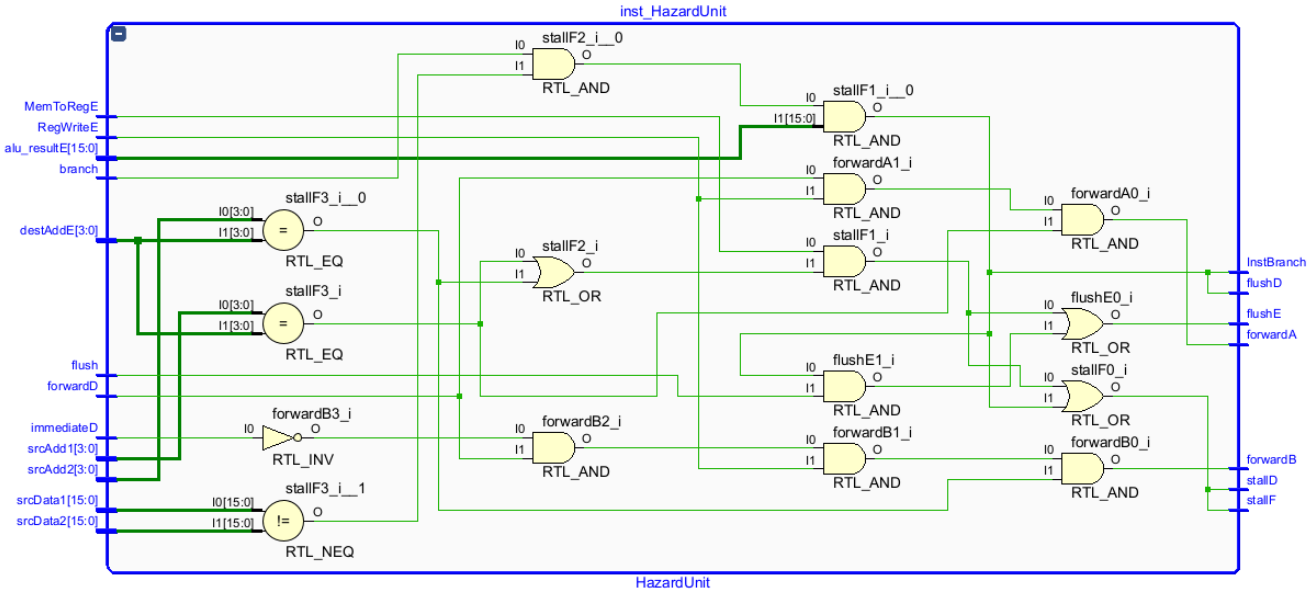
v. WriteBack

2. 제어 유닛 및 관련 구성 요소 Schemetic

i. Control Unit

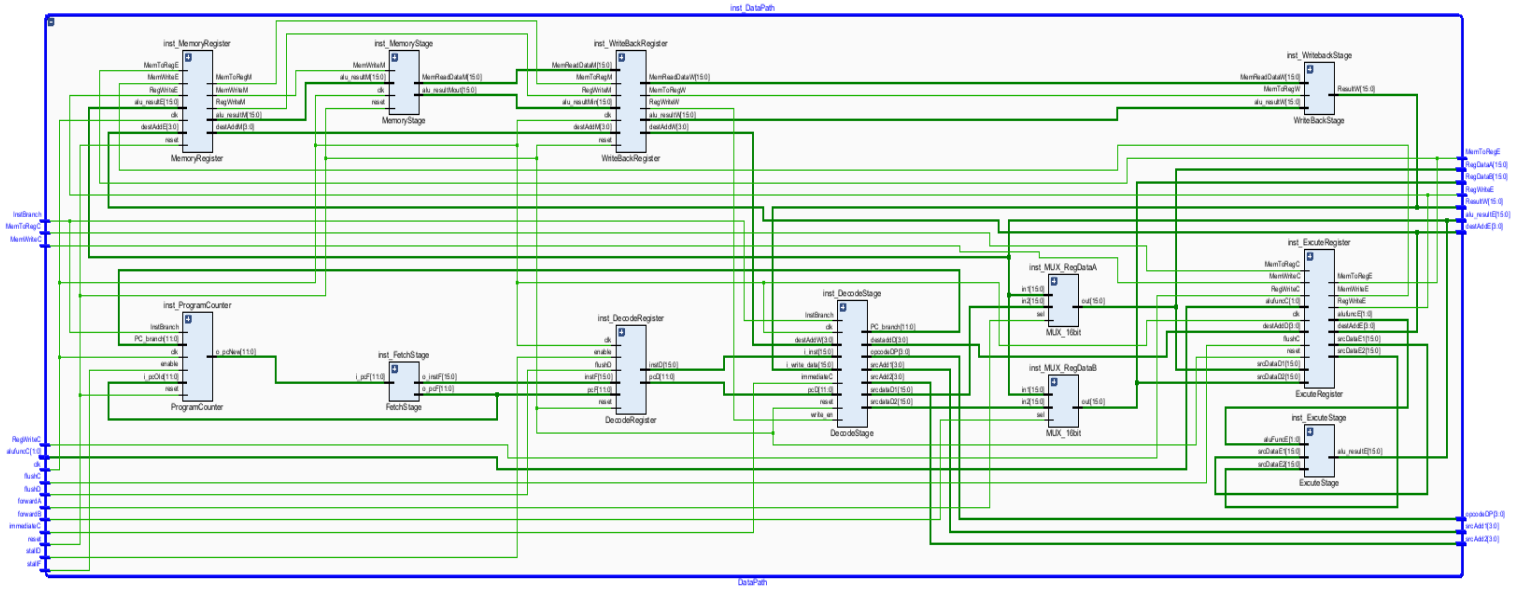


ii. Hazard Unit

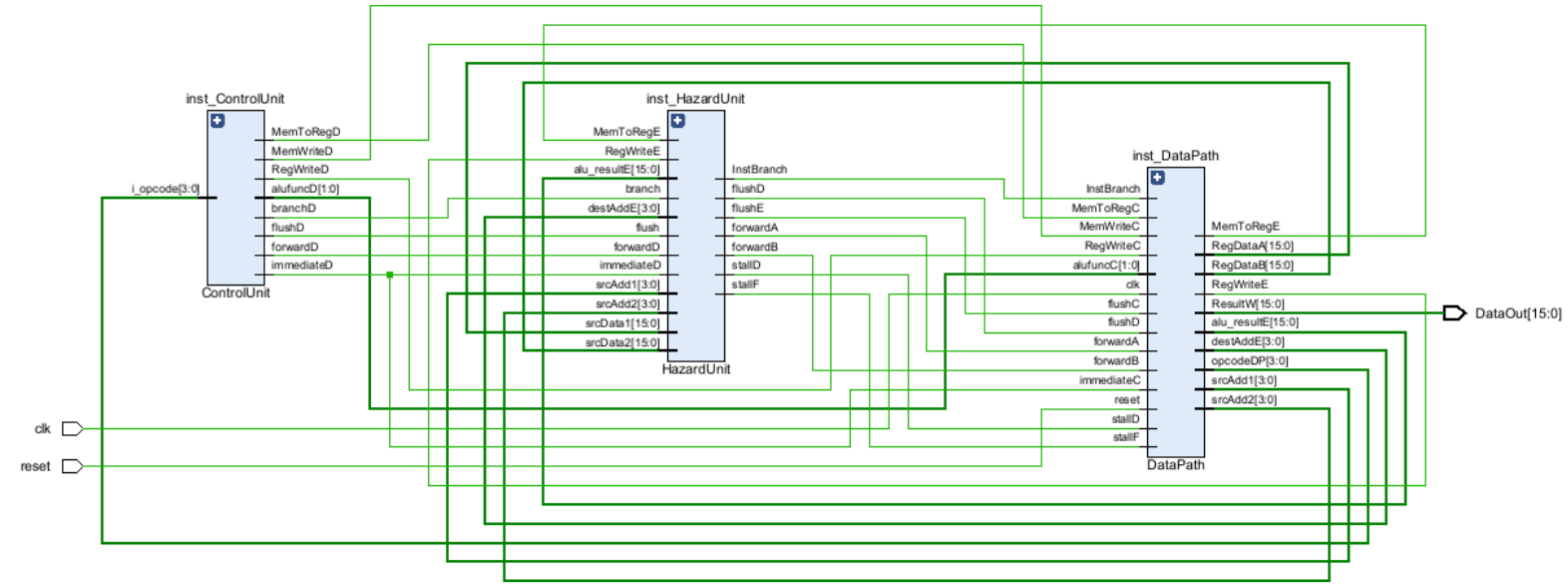


3. CPU 구성요소 Schemetic

i. Data Path



ii. CPU


inst_ControlUnit
i_opcode[3:0]
MemToRegD
MemWriteD
RegWriteD
alufuncD[1:0]
branchD
flushD
forwardD
immediateD
ControlUnit
inst_HazardUnit
MemToRegE
RegWriteE
alu_resultE[15:0]
branch
destAddE[3:0]
flush
forwardD
immediateD
srcAdd1[3:0]
srcAdd2[3:0]
srcData1[15:0]
srcData2[15:0]
InstBranch
flushD
flushE
forwardA
forwardB
stallD
stallF
HazardUnit
inst_DataPath
InstBranch
MemToRegC
MemWriteC
RegWriteC
alufuncC[1:0]
clk
flushC
flushD
forwardA
forwardB
immediateC
reset
stallD
stallF
MemToRegE
RegDataA[15:0]
RegDataB[15:0]
RegWriteE
ResultW[15:0]
alu_resultE[15:0]
destAddE[3:0]
opcodeDP[3:0]
srcAdd1[3:0]
srcAdd2[3:0]
DataPath
clk
reset
DataOut[15:0]

3. 모듈 상세 설명

- ALU 모듈

ALU 모듈은 두 개의 16 비트 입력값(ALU_srcdata_1, ALU_srcdata_2)과 2 비트 제어 신호(ALU_func)를 받아 덧셈, 뺄셈, 곱셈, 비교 연산을 수행하여 결과(ALU_result)를 출력하는 산술 논리 연산 장치이다. ALU_func 가 00 이면 덧셈, 01 이면 뺄셈, 10 이면 곱셈, 11 이면 첫 번째 입력이 두 번째 입력보다 작은지 비교하여 결과를 결정한다.

```
1   module ALU(
2       input       [1:0]   ALU_func,
3       input       [15:0]  ALU_srcdata_1,
4       input       [15:0]  ALU_srcdata_2,
5       output  reg [15:0]  ALU_result
6   );
7
8   parameter   ADD = 2'b00,
9               SUB = 2'b01,
10              MUL = 2'b10,
11              SLT = 2'b11;
12
13  always@(*)begin
14      ALU_result = 16'h0000;
15      case(ALU_func)
16              ADD: ALU_result = ALU_srcdata_1 + ALU_srcdata_2;
17              SUB: ALU_result = ALU_srcdata_1 - ALU_srcdata_2;
18              MUL: ALU_result = ALU_srcdata_1 * ALU_srcdata_2;
19              SLT: ALU_result = (ALU_srcdata_1 < ALU_srcdata_2) ? {{(15){1'b0}}, 1'b1} :  {(16){1'b0}};
20      endcase
21  end
22
23  endmodule
```

상세 설명

입력값:

- ALU_func (2 비트): 수행할 연산을 지정함.
- ALU_srcdata_1 (16 비트): 첫 번째 피연산자.
- ALU_srcdata_2 (16 비트): 두 번째 피연산자.

출력값:

- ALU_result (16 비트): 연산 결과.

동작 방식:

- ALU_func 가 00 이면, ALU_srcdata_1 과 ALU_srcdata_2 를 더하여 ALU_result 에 저장함.
- ALU_func 가 01 이면, ALU_srcdata_1 에서 ALU_srcdata_2 를 빼서 ALU_result 에 저장함.
- ALU_func 가 10 이면, ALU_srcdata_1 과 ALU_srcdata_2 를 곱하여 ALU_result 에 저장함.
- ALU_func 가 11 이면, ALU_srcdata_1 이 ALU_srcdata_2 보다 작으면 ALU_result 에 1 을, 그렇지 않으면 0 을 저장함.

.

- MUX_16bit 모듈

MUX 모듈은 두 개의 16비트 입력값(in1, in2)과 1비트 선택 신호(sel)를 받아 선택 신호에 따라 하나의 입력값을 출력하는 멀티플렉서이다. sel이 1이면 in1을, 0이면 in2를 출력한다

```
1     module MUX_16bit(
2         input  [15:0] in1,
3         input [15:0]  in2,
4         input         sel,
5         output      [15:0] out
6     );
7
8     assign out = (sel) ? in1 : in2;
9
10    endmodule
```

입력값

- in1 (16 비트): 첫 번째 입력 데이터.
- in2 (16 비트): 두 번째 입력 데이터.
- sel (1 비트): 선택 신호.

출력값

- out (16 비트): 선택된 입력 데이터.

동작 방식

- sel 이 1 이면, out 에 in1 을 출력한다.
- sel 이 0 이면, out 에 in2 를 출력한다.

- ProgramCounter 모듈

ProgramCounter 모듈은 클럭 신호(clk)와 리셋 신호(reset), 제어 신호(enable, InstBranch)를 받아 현재 프로그램 카운터 값을 관리하고 갱신하는 모듈이다. 리셋되면 0 으로 초기화되고, 브랜치 신호가 활성화되면 지정된 분기 주소(PC_branch)로 갱신된다.

```
1   module ProgramCounter(
2       input           clk,
3       input           reset,
4       input           enable,
5       input           InstBranch,
6       input  [11:0]   PC_branch,
7       input  [11:0]   i_pcOld,
8       output [11:0]   o_pcNew
9   );
10
11  reg     [11:0]      r_pcNew;
12
13  always@(posedge clk or negedge reset)begin
14      if(~reset)
15          r_pcNew <= 0;
16      else if(!enable)
17          r_pcNew <= i_pcOld;
18      else if(InstBranch)
19          r_pcNew <= PC_branch;
20  end
21
22  assign o_pcNew = r_pcNew;
23
24  endmodule
```

입력값

- clk (1 비트): 클럭 신호.
- reset (1 비트): 리셋 신호.
- enable (1 비트): 프로그램 카운터 활성화 신호.
- InstBranch (1 비트): 브랜치 명령어 신호.
- PC_branch (12 비트): 분기할 주소.
- i_pcOld (12 비트): 현재 프로그램 카운터 값.

출력값

- o_pcNew (12 비트): 갱신된 프로그램 카운터 값.

동작 방식

- 리셋 신호가 활성화되면, r_pcNew 를 0 으로 초기화한다.
- enable 신호가 비활성화되면, r_pcNew 에 i_pcOld 를 저장한다.
- InstBranch 신호가 활성화되면, r_pcNew 에 PC_branch 를 저장한다.
- o_pcNew 는 r_pcNew 의 값을 출력한다.

- PC_Adder 모듈

PC_Adder 모듈은 12 비트 입력값(i_pcOld)을 받아 1 을 더한 결과를 출력하는 프로그램 카운터 증가기이다. 현재 PC 값을 받아 다음 명령어 주소를 계산한다.

```
1    module PC_Adder(
2        input  [11:0]   i_pcOld,
3        output [11:0]   o_pcNew
4    );
5
6    assign o_pcNew = i_pcOld + 12'd1;
7
8    endmodule
```

입력값

- i_pcOld (12 비트): 현재 프로그램 카운터 값.

출력값

- o_pcNew (12 비트): 1 증가된 프로그램 카운터 값.

동작 방식

- i_pcOld 값에 1 을 더하여 o_pcNew 에 출력한다.

- Memory Register 모듈

MemoryRegister 모듈은 실행 단계에서 생성된 제어 신호와 데이터를 저장하여 메모리 단계로 전달하는 역할을 한다.

```
1   module MemoryRegister(
2       input               clk, reset,
3       input               MemWriteE, MemToRegE, RegWriteE,
4       input       [3:0]   destAddE,
5       input       [15:0]  alu_resultE,
6       output reg  [3:0]   destAddM,
7       output reg          MemWriteM, MemToRegM, RegWriteM,
8       output reg  [15:0]  alu_resultM
9   );
10
11  always@(posedge clk or negedge reset)begin
12      if(~reset)begin
13          MemWriteM   <= 1'b0;
14          MemToRegM   <= 1'b0;
15          RegWriteM   <= 1'b0;
16          alu_resultM <= 16'h0000;
17          destAddM    <= 4'b0000;
18      end
19      else begin
20          MemWriteM   <= MemWriteE;
21          MemToRegM   <= MemToRegE;
22          RegWriteM   <= RegWriteE;
23          alu_resultM <= alu_resultE;
24          destAddM    <= destAddE;
25      end
26  end
27
28  endmodule
```

입력값

- clk (1 비트): 클럭 신호.
- reset (1 비트): 리셋 신호.
- MemWriteE (1 비트): 메모리 쓰기 제어 신호.
- MemToRegE (1 비트): 메모리 데이터 레지스터 제어 신호.
- RegWriteE (1 비트): 레지스터 쓰기 제어 신호.
- destAddE (4 비트): 목적지 주소.
- alu_resultE (16 비트): ALU 연산 결과.

출력값

- MemWriteM (1 비트): 메모리 쓰기 제어 신호.
- MemToRegM (1 비트): 메모리 데이터 레지스터 제어 신호.
- RegWriteM (1 비트): 레지스터 쓰기 제어 신호.
- destAddM (4 비트): 목적지 주소.
- alu_resultM (16 비트): ALU 연산 결과.

동작 방식

- 리셋 신호가 활성화되면, 모든 출력 신호를 0 으로 초기화한다.
- 클럭 상승 모서리에서 입력된 제어 신호(MemWriteE, MemToRegE, RegWriteE)와 데이터를(alu_resultE, destAddE) 출력 레지스터에 저장한다.
- 저장된 출력 신호와 데이터는 다음 단계인 메모리 단계에서 사용된다.

- Instruction Mem 모듈

Instruction_Mem 모듈은 프로그램 카운터(PC) 값을 입력받아 해당 주소의 명령어를 출력하는 역할을 하는 명령어 메모리이다.

```
module Instruction_Mem(
    input      [11:0] PCAdd_pc,
    output     [15:0] M_instruction
); //instruction memory(fetch)

    reg [15:0] instM[4095:0];

    //Data forward
    /*initial begin //4+2+1+2+1 = 10, r5 <- 10
        instM[0] = 16'b0110_0001_0100_0001; //r1 = r1 + 4
        instM[1] = 16'b0110_0001_0010_0010; //r2 = r1 + 2
        instM[2] = 16'b0110_0010_0001_0011; //r3 = r2 + 1
        instM[3] = 16'b0110_0011_0010_0100; //r4 = r3 + 2
        instM[4] = 16'b0110_0100_0001_0101; //r5 = r4 + 1
    end*/

    //Branch
    initial begin
        instM[0] = 16'b0110_0001_0100_0001; //r1 = r1 + 4
        instM[1] = 16'b0110_0010_0100_0010; //r2 = r2 + 4
        instM[2] = 16'b0110_0011_0001_0011; //r3 = r3 + 1
        instM[3] = 16'b0110_0100_0001_0100; //r4 = r4 + 1
        instM[4] = 16'b0111_0001_0001_0001; //r1 = r1 - 1
        instM[5] = 16'b1001_0001_0001_0101; //BNE r1, 1
        //if(r1=1)
        instM[6] = 16'b0110_0001_0100_0010; //r2 = r1 + 4
        instM[7] = 16'b0110_0010_0100_0011; //r3 = r2 + 4
        instM[8] = 16'b0110_1111_0000_1111; //unused
        instM[9] = 16'b0110_1111_0000_1111; //unused
        instM[10] = 16'b0010_0011_0010_0100; //r4 = r3 + r2
    end

    //Load and Store
    /*initial begin
        instM[0] = 16'b0000_0001_0100_0001; //r1 = mem[r1 + 4], (r1 = 4)
        instM[1] = 16'b0110_0010_0100_0010; //r2 = r2 + 4
        instM[2] = 16'b0110_0011_0001_0011; //r3 = r3 + 1
        instM[3] = 16'b0110_0001_1111_0001; //r1 = r1 + 15
        instM[4] = 16'b0001_0001_0000_0000; //mem[r1] = r1
    end*/

    //Datafoward with LDA
    /*initial begin
        instM[0] = 16'b0000_0000_1111_0001; //r1 = mem[15](15)
        instM[1] = 16'b0110_1000_0000_1000; //unused
        instM[2] = 16'b0110_1000_0000_1000; //unused
        instM[3] = 16'b0110_0001_0010_0010; //r2 = r1 + 2
        instM[4] = 16'b0110_0010_0001_0011; //r3 = r2 + 1
        instM[5] = 16'b0110_0011_0010_0100; //r4 = r3 + 2
        instM[6] = 16'b0110_0100_0001_0101; //r5 = r4 + 1
    end*/

    assign M_instruction = instM[PCAdd_pc];

endmodule
```

-

입력값

- PCAdd_pc (12 비트): 프로그램 카운터 값.

출력값

- M_instruction (16 비트): 프로그램 카운터 값에 해당하는 명령어.

동작 방식

- PCAdd_pc 를 사용하여 명령어 메모리(instM)에서 해당 주소의 명령어를 읽어 M_instruction 에 출력한다.
- 초기 블록(initial begin)에서 명령어 메모리(instM)를 특정 명령어들로 초기화한다.

- Memory Data 모듈

MemoryData 모듈은 16 비트 주소를 받아 메모리에서 데이터를 읽고, 쓰기 활성화 신호가 있을 경우 메모리에 데이터를 쓰는 역할을 하는 데이터 메모리이다.

```
1   module MemoryData(
2       input                clk, reset, write_en,
3       input      [15:0]    addrM,
4       input      [15:0]    write_dataM,
5       output     [15:0]    read_dataM
6   );//516B memory
7
8   integer i;
9
10  reg [15:0] mem[65536:0];
11
12  always@(negedge clk or negedge reset)begin
13      if(~reset)
14          for(i=0; i<65536; i=i+1)begin
15              mem[i] <= i;
16          end
17      else if(write_en)
18          mem[addrM] <= write_dataM;
19  end
20
21  assign read_dataM = mem[addrM];
22
23  endmodule
```

입력값

- clk (1 비트): 클럭 신호.
- reset (1 비트): 리셋 신호.
- write_en (1 비트): 쓰기 활성화 신호.
- addrM (16 비트): 메모리 주소.
- write_dataM (16 비트): 메모리에 쓸 데이터.

출력값

- read_dataM (16 비트): 메모리에서 읽은 데이터.

동작 방식

- 리셋 신호가 활성화되면, 메모리의 초기화 블록은 특정 초기값으로 메모리를 설정한다.
- 클럭의 상승 모서리에서, 쓰기 활성화 신호(write_en)가 켜져 있으면, addrM 주소의 메모리에 write_dataM 데이터를 쓴다.
- addrM 주소의 메모리 값을 read_dataM 에 출력한다

- Opcode Decoder 모듈

OpcodeDecoder 모듈은 4비트 OPCODE(i_opcode)를 입력받아 해당 오피코드에 따른 제어 신호들을 생성하여 실행 레지스터로 전달하는 역할을 한다.

```
module OpcodeDecoder(
    input   [3:0] i_opcode,
    output  reg branch,
    output  reg flush,
    output  reg RegWrite,
    output  reg MemToReg,
    output  reg MemWrite,
    output  reg immediate,
    output  reg forward,
    output  reg [1:0] o_alufunc
); //to excute register

reg [8:0] flag;

parameter
          LDA_imm       = 4'b0000,
          STA_imm       = 4'b0001,
          CAL_add       = 4'b0010,
          CAL_sub       = 4'b0011,
          CAL_mul       = 4'b0100,
          CAL_SLT       = 4'b0101,
          IMM_add       = 4'b0110,
          IMM_sub       = 4'b0111,
          IMM_mul       = 4'b1000,
          BAF_immsub    = 4'b1001,
          BAF_regsub    = 4'b1010;


always @(*) begin
    {o_alufunc, branch, flush, RegWrite, MemWrite, MemToReg, immediate, forward} = flag;
  always @(*) begin
      flag = 9'b00_0000000;
      case(i_opcode[3:0])
          LDA_imm     : flag =    9'b00_0010111;
          STA_imm     : flag =    9'b00_0001010;
          CAL_add     : flag =    9'b00_0010001;
          CAL_sub     : flag =    9'b01_0010001;
          CAL_mul     : flag =    9'b10_0010001;
          CAL_SLT     : flag =    9'b11_0010001;
          IMM_add     : flag =    9'b00_0010011;
          IMM_sub     : flag =    9'b01_0010011;
          IMM_mul     : flag =    9'b10_0010011;
          BAF_immsub  : flag =    9'b01_1100010;
          BAF_regsub  : flag =    9'b01_1100000;
      endcase
  end

  endmodule
```

입력값

- i_opcode (4 비트): 명령어의 오피코드.

출력값

- branch (1 비트): 브랜치 신호.
- flush (1 비트): 플러시 신호.
- RegWrite (1 비트): 레지스터 쓰기 신호.
- MemToReg (1 비트): 메모리에서 레지스터로 데이터 이동 신호.
- MemWrite (1 비트): 메모리 쓰기 신호.
- immediate (1 비트): immediate value 사용 신호.
- forward (1 비트): 포워딩 신호.
- o_alufunc (2 비트): ALU 연산 함수.

동작 방식

- 입력된 오퍼코드(i_opcode)에 따라 case 문을 사용하여 제어 신호들을 설정한다.
- 각 오퍼코드에 대해 다음과 같은 제어 신호들이 설정된다:
    - LDA_imm (4'b0000): immediate value 를 로드하여 레지스터에 쓰기.
    - STA_imm (4'b0001): immediate value 를 사용하여 메모리에 데이터 쓰기.
    - CAL_add (4'b0010): 레지스터 간 덧셈 연산.
    - CAL_sub (4'b0011): 레지스터 간 뺄셈 연산.
    - CAL_mul (4'b0100): 레지스터 간 곱셈 연산.
    - CAL_SLT (4'b0101): 레지스터 간 비교하여 작은 값 설정.
    - IMM_add (4'b0110): immediate value 와 레지스터 값 덧셈.
    - IMM_sub (4'b0111): immediate value 와 레지스터 값 뺄셈.
    - IMM_mul (4'b1000): immediate value 와 레지스터 값 곱셈.
    - BAF_immsub (4'b1001): immediate value 를 사용한 뺄셈 결과에 따른 브랜치.
    - BAF_regsub (4'b1010): 레지스터 값을 사용한 뺄셈 결과에 따른 브랜치.

-

-

-

-

-

- ## Register File 모듈

- Register_File 모듈은 16개의 16비트 레지스터 파일을 관리하며, 읽기 및 쓰기 연산을 수행하는 역할을 한다.

```
1   module Register_File(
2       input              clk,
3       input              reset,
4       input              i_write_en,
5       input              immediateC,
6       input       [3:0]  i_read_add1,
7       input       [3:0]  i_read_add2,
8       input       [3:0]  i_write_add,
9       input       [15:0] i_write_data,
10      output reg  [15:0] o_read_data1,
11      output reg  [15:0] o_read_data2
12  );
13
14  //resister initialize and write-back(excute)
15  reg [15:0] registers [0:15];
16
17  integer i;
18  always @(negedge clk or negedge reset) begin //negedge clk for 1clock cycle
19      if (~reset) begin
20          for (i = 0; i < 16; i = i + 1) begin
21              registers[i] <= 16'h0000;
22          end
23      end
24      else if (i_write_en) begin
25          registers[i_write_add] <= i_write_data;
26      end
27  end
28
29  always@(*)begin
30      if(~reset)begin
31          o_read_data1 = 0;
32          o_read_data2 = 0;
33      end
34      else if(immediateC)begin
35          o_read_data1 = registers[i_read_add1];
36          o_read_data2 = {12'd0, i_read_add2};
37      end
38      else begin
39          o_read_data1 = registers[i_read_add1];
40          o_read_data2 = registers[i_read_add2];
41      end
42  end
43
44
45  endmodule
```

-

-

입력값

- clk (1 비트): 클럭 신호.
- reset (1 비트): 리셋 신호.
- i_write_en (1 비트): 쓰기 활성화 신호.
- immediateC (1 비트): immediate value 사용 신호.
- i_read_add1 (4 비트): 첫 번째 읽기 주소.
- i_read_add2 (4 비트): 두 번째 읽기 주소.
- i_write_add (4 비트): 쓰기 주소.
- i_write_data (16 비트): 쓰기 데이터.

출력값

- o_read_data1 (16 비트): 첫 번째 읽기 데이터.
- o_read_data2 (16 비트): 두 번째 읽기 데이터.

동작 방식

1. **리셋 동작**:
    - 리셋 신호가 활성화되면, 모든 레지스터를 0 으로 초기화한다.
2. **쓰기 동작**:
    - 클럭 신호의 하강 모서리에서 쓰기 활성화 신호(i_write_en)가 켜져 있으면, i_write_add 주소의 레지스터에 i_write_data 값을 쓴다.
3. **읽기 동작**:
    - 리셋 신호가 활성화되면, o_read_data1 과 o_read_data2 를 0 으로 설정한다.
    - immediateC 신호가 켜져 있으면, o_read_data1 은 i_read_add1 주소의 레지스터 값을, o_read_data2 는 sign-extended immediate value 를 반환한다.
    - immediateC 신호가 꺼져 있으면, o_read_data1 과 o_read_data2 는 각각 i_read_add1 과 i_read_add2 주소의 레지스터 값을 반환한다.

- Branch Detectect 모듈

BranchDetect 모듈은 브랜치 명령어를 감지하여 프로그램 카운터(PC) 값을 업데이트하는 역할을 한다.

```
1   module BranchDetect(
2       input                   InstBranch,
3       input       [3:0]       destaddD,
4       input       [11:0]      pcD,
5       output reg  [11:0]      PC_branch
6   );
7
8   always@(*)begin
9       if(InstBranch)
10          PC_branch = pcD - destaddD;
11      else
12          PC_branch = pcD;
13  end
14
15  endmodule
```

입력값

- InstBranch (1 비트): 브랜치 명령어 신호.
- destaddD (4 비트): 브랜치 대상 주소 오프셋.
- pcD (12 비트): 현재 프로그램 카운터 값.

출력값

- PC_branch (12 비트): 업데이트된 프로그램 카운터 값.

동작 방식

- InstBranch 신호가 활성화되면, PC_branch 는 현재 PC 값(pcD)에서 브랜치 대상 주소 오프셋(destaddD)을 뺀 값을 갖는다.
- InstBranch 신호가 비활성화되면, PC_branch 는 현재 PC 값(pcD)을 그대로 유지한다.

- FetchStage 모듈

FetchStage 모듈은 현재 프로그램 카운터(PC) 값을 받아 다음 명령어를 페치하고, 다음 PC 값을 계산하는 역할을 한다.

```
module FetchStage(
    input   [11:0]      i_pcF,
    output  [11:0]      o_pcF,
    output  [15:0]      o_instF
);//PC behavior

PC_Adder    inst_PCAdder(
    .i_pcOld(i_pcF),
    .o_pcNew(o_pcF)
);

Instruction_Mem     inst_InstMem(
    .PCAdd_pc(i_pcF),
    .M_instruction(o_instF)
);

endmodule
```

입력값

- i_pcF (12 비트): 현재 프로그램 카운터 값.

출력값

- o_pcF (12 비트): 다음 프로그램 카운터 값.
- o_instF (16 비트): 페치된 명령어.

동작 방식

- i_pcF 값이 PC_Adder 모듈에 전달되어 1 증가된 다음 PC 값(o_pcNew)이 생성된다.
- i_pcF 값이 Instruction_Mem 모듈에 전달되어 해당 주소의 명령어(M_instruction)가 페치된다.
- PC_Adder 모듈은 현재 PC 값(i_pcF)을 1 증가시켜 o_pcF 를 계산한다.
- Instruction_Mem 모듈은 입력된 PC 값(i_pcF)에 해당하는 명령어를 메모리에서 읽어 o_instF 로 출력한다.

- ## Decode Register 모듈

DecodeRegister 모듈은 페치 단계에서 디코드 단계로 명령어와 프로그램 카운터(PC) 값을 전달하는 역할을 한다.

```
1     module DecodeRegister(
2         input                clk, reset, enable, flushD,
3         input        [11:0]  pcF,
4         input        [15:0]  instF,
5         output reg   [11:0]  pcD,
6         output reg   [15:0]  instD
7     );
8
9     always@(posedge clk or negedge reset)begin
10        if(~reset | flushD)begin
11            pcD   <= 0;
12            instD <= 0;
13        end
14        else if(!enable)begin
15            pcD   <= pcF;
16            instD <= instF;
17        end
18    end
19
20    endmodule
```

입력값

- clk (1 비트): 클럭 신호.
- reset (1 비트): 리셋 신호.
- enable (1 비트): 레지스터 사용 가능 신호.
- flushD (1 비트): 디코드 단계의 플러시 신호.
- pcF (12 비트): 페치 단계의 프로그램 카운터 값.
- instF (16 비트): 페치 단계의 명령어.

출력값

- pcD (12 비트): 디코드 단계로 전달되는 프로그램 카운터 값.
- instD (16 비트): 디코드 단계로 전달되는 명령어.

동작 방식

- 리셋 신호(reset)가 활성화되거나 플러시 신호(flushD)가 활성화되면, pcD 와 instD 를 0 으로 초기화한다.
- 클럭 신호의 상승 모서리에서, enable 신호가 비활성화되면 페치 단계의 프로그램 카운터 값(pcF)과 명령어(instF)를 디코드 단계로 전달하여 pcD 와 instD 에 저장한다.

- Decode Stage 모듈

DecodeStage 모듈은 명령어를 해석하고 제어 신호를 생성하며, 브랜치 주소를 계산하고 레지스터 파일에서 데이터를 읽어오는 역할을 한다

```
1   module DecodeStage(
2       input           clk,
3       input           reset,
4       input           write_en,
5       input  [3:0]    destAddW,
6       input  [15:0]   i_write_data,
7       input           immediateC,
8       input  [15:0]   i_inst,
9       input           InstBranch,
10      input   [11:0]  pcD,
11      output  [11:0]  PC_branch,
12      output [3:0]    opcodeDP,
13      output [3:0]    destaddD,
14      output [3:0]    srcAdd1, srcAdd2,
15      output [15:0]   srcdataD1, srcdataD2
16  );
17
18  BranchDetect    inst_BranchDetect(
19      .InstBranch(InstBranch),
20      .destaddD(destaddD),
21      .pcD(pcD),
22      .PC_branch(PC_branch)//out
23  );
24
25  Register_File   inst_ResisterFile(
26      .clk(clk),
27      .reset(reset),
28      .i_write_en(write_en),
29      .immediateC(immediateC),
30      .i_read_add1(srcAdd1),
31      .i_read_add2(srcAdd2),
32      .i_write_add(destAddW),
33      .i_write_data(i_write_data),
34      .o_read_data1(srcdataD1), //out
35      .o_read_data2(srcdataD2)
36  );
37
38  assign srcAdd1 = i_inst[11:8];
39  assign srcAdd2 = i_inst[7:4];
40  assign destaddD = i_inst[3:0];
41  assign opcodeDP = i_inst[15:12];
42
43  endmodule
44
```

입력값

- clk (1 비트): 클럭 신호.
- reset (1 비트): 리셋 신호.
- write_en (1 비트): 레지스터 파일 쓰기 활성화 신호.
- destAddW (4 비트): 쓰기 주소.
- i_write_data (16 비트): 쓰기 데이터.
- immediateC (1 비트): immediate value 사용 신호.
- i_inst (16 비트): 입력 명령어.
- InstBranch (1 비트): 브랜치 명령어 신호.
- pcD (12 비트): 현재 프로그램 카운터 값.

출력값

- PC_branch (12 비트): 브랜치 주소.
- opcodeDP (4 비트): 오피코드.
- destaddD (4 비트): 목적지 주소.
- srcAdd1 (4 비트): 첫 번째 소스 주소.
- srcAdd2 (4 비트): 두 번째 소스 주소.
- srcdataD1 (16 비트): 첫 번째 소스 데이터.
- srcdataD2 (16 비트): 두 번째 소스 데이터.

동작 방식

1. **브랜치 주소 계산**:
    - BranchDetect 모듈은 InstBranch 신호가 활성화되면, 현재 PC 값(pcD)에서 브랜치 오프셋(destaddD)을 빼서 PC_branch 값을 계산한다.
2. **레지스터 파일 읽기 및 쓰기**:
    - Register_File 모듈은 클럭 신호(clk)와 리셋 신호(reset)에 따라 동작하며, 쓰기 활성화 신호(write_en)가 켜져 있으면 지정된 쓰기 주소(destAddW)에 쓰기 데이터(i_write_data)를 쓴다.
    - Register_File 모듈은 읽기 주소(srcAdd1, srcAdd2)에 따라 레지스터 파일에서 데이터를 읽어와 srcdataD1 과 srcdataD2 에 저장한다.
3. **명령어 해석**:
    - 입력된 명령어(i_inst)의 각 필드를 해석하여 소스 주소(srcAdd1, srcAdd2), 목적지 주소(destaddD), 오피코드(opcodeDP)를 추출한다.

- Execute Register 모듈

ExcuteRegister 모듈은 디코드 단계에서 실행 단계로 데이터를 전달하는 역할을 한다. 클럭 신호와 리셋 신호에 따라 제어 신호와 데이터를 저장하고 전달한다.

```
module ExcuteRegister(
    input               clk, reset, flushC,
    input               RegWriteC, MemWriteC, MemToRegC,
    input       [1:0]   alufuncC,
    input       [15:0]  srcDataD1, srcDataD2,
    input       [3:0]   destAddD,
    output reg          RegWriteE, MemWriteE, MemToRegE,
    output reg  [1:0]   alufuncE,
    output reg  [15:0]  srcDataE1, srcDataE2,
    output reg  [3:0]   destAddE
);

always@(posedge clk or negedge reset)begin
    if(~reset || flushC)begin
        RegWriteE <= 1'b0;
        MemWriteE <= 1'b0;
        MemToRegE <= 1'b0;
        alufuncE  <= 2'b00;
        srcDataE1 <= 16'h0000;
        srcDataE2 <= 16'h0000;
        destAddE  <= 4'b0000;
    end
    else begin
        RegWriteE <= RegWriteC;
        MemWriteE <= MemWriteC;
        MemToRegE <= MemToRegC;
        alufuncE  <= alufuncC;
        srcDataE1 <= srcDataD1;
        srcDataE2 <= srcDataD2;
        destAddE  <= destAddD;
    end
end

endmodule
```

입력값

- clk (1 비트): 클럭 신호.
- reset (1 비트): 리셋 신호.
- flushC (1 비트): 실행 단계의 플러시 신호.
- RegWriteC (1 비트): 레지스터 쓰기 신호.
- MemWriteC (1 비트): 메모리 쓰기 신호.
- MemToRegC (1 비트): 메모리에서 레지스터로 데이터 이동 신호.
- alufuncC (2 비트): ALU 연산 함수.
- srcDataD1 (16 비트): 첫 번째 소스 데이터.
- srcDataD2 (16 비트): 두 번째 소스 데이터.
- destAddD (4 비트): 목적지 주소.

출력값

- RegWriteE (1 비트): 실행 단계로 전달되는 레지스터 쓰기 신호.
- MemWriteE (1 비트): 실행 단계로 전달되는 메모리 쓰기 신호.
- MemToRegE (1 비트): 실행 단계로 전달되는 메모리에서 레지스터로 데이터 이동 신호.
- alufuncE (2 비트): 실행 단계로 전달되는 ALU 연산 함수.
- srcDataE1 (16 비트): 실행 단계로 전달되는 첫 번째 소스 데이터.
- srcDataE2 (16 비트): 실행 단계로 전달되는 두 번째 소스 데이터.
- destAddE (4 비트): 실행 단계로 전달되는 목적지 주소.

동작 방식

- 리셋 신호(reset)가 활성화되거나 플러시 신호(flushC)가 활성화되면, 모든 출력 신호를 초기화한다.
- 클럭 신호의 상승 모서리에서 입력된 제어 신호(RegWriteC, MemWriteC, MemToRegC)와 데이터를(srcDataD1, srcDataD2, destAddD, alufuncC) 실행 단계로 전달하여 출력 신호(RegWriteE, MemWriteE, MemToRegE, srcDataE1, srcDataE2, destAddE, alufuncE)에 저장한다.

- Execute Stage 모듈

ExcuteStage 모듈은 ALU 연산을 수행하는 역할을 한다. 입력된 연산 함수와 소스 데이터를 사용하여 연산 결과를 생성한다.

```
1   module ExcuteStage(
2       input   [1:0]           aluFuncE,
3       input   [15:0]          srcDataE1, srcDataE2,
4       output  [15:0]          alu_resultE
5   );
6
7   ALU     instALU(
8       .ALU_func(aluFuncE),
9       .ALU_srcdata_1(srcDataE1),
10      .ALU_srcdata_2(srcDataE2),
11      .ALU_result(alu_resultE)
12  );
13
14  endmodule
```

입력값

- aluFuncE (2 비트): ALU 연산 함수.
- srcDataE1 (16 비트): 첫 번째 소스 데이터.
- srcDataE2 (16 비트): 두 번째 소스 데이터.

출력값

- alu_resultE (16 비트): ALU 연산 결과.

동작 방식

- 입력된 ALU 연산 함수(aluFuncE)와 소스 데이터(srcDataE1, srcDataE2)를 사용하여 ALU 모듈에서 연산을 수행한다.
- ALU 모듈은 지정된 연산 함수에 따라 덧셈, 뺄셈, 곱셈, 비교 연산을 수행하고, 결과를 alu_resultE 에 저장한다.

- Memory Register 모듈

MemoryRegister 모듈은 Execute 단계에서 Memory Access단계로 데이터를 전달하는 역할을 한다. 클럭 신호와 리셋 신호에 따라 제어 신호와 데이터를 저장하고 전달한다.

```
module MemoryRegister(
    input              clk, reset,
    input              MemWriteE, MemToRegE, RegWriteE,
    input       [3:0]  destAddE,
    input       [15:0] alu_resultE,
    output reg  [3:0]  destAddM,
    output reg         MemWriteM, MemToRegM, RegWriteM,
    output reg  [15:0] alu_resultM
);

always@(posedge clk or negedge reset)begin
    if(~reset)begin
        MemWriteM   <= 1'b0;
        MemToRegM   <= 1'b0;
        RegWriteM   <= 1'b0;
        alu_resultM <= 16'h0000;
        destAddM    <= 4'b0000;
    end
    else begin
        MemWriteM   <= MemWriteE;
        MemToRegM   <= MemToRegE;
        RegWriteM   <= RegWriteE;
        alu_resultM <= alu_resultE;
        destAddM    <= destAddE;
    end
end

endmodule
```

입력값

- clk (1 비트): 클럭 신호.
- reset (1 비트): 리셋 신호.
- MemWriteE (1 비트): 메모리 쓰기 신호.
- MemToRegE (1 비트): 메모리에서 레지스터로 데이터 이동 신호.
- RegWriteE (1 비트): 레지스터 쓰기 신호.
- destAddE (4 비트): 목적지 주소.
- alu_resultE (16 비트): ALU 연산 결과.

출력값

- MemWriteM (1 비트): 메모리 단계로 전달되는 메모리 쓰기 신호.
- MemToRegM (1 비트): 메모리 단계로 전달되는 메모리에서 레지스터로 데이터 이동 신호.
- RegWriteM (1 비트): 메모리 단계로 전달되는 레지스터 쓰기 신호.
- destAddM (4 비트): 메모리 단계로 전달되는 목적지 주소.
- alu_resultM (16 비트): 메모리 단계로 전달되는 ALU 연산 결과.

동작 방식

- 리셋 신호(reset)가 활성화되면, 모든 출력 신호를 초기화한다.
- 클럭 신호의 상승 모서리에서 입력된 제어 신호(MemWriteE, MemToRegE, RegWriteE)와 데이터를(alu_resultE, destAddE) 메모리 단계로 전달하여 출력 신호(MemWriteM, MemToRegM, RegWriteM, alu_resultM, destAddM)에 저장한다.

- Memory Stage 모듈

MemoryStage 모듈은 메모리 접근 단계에서 메모리 읽기 및 쓰기 동작을 수행하며, 메모리에서 데이터를 읽어오거나 데이터를 메모리에 쓰는 역할을 한다.

```
1   module MemoryStage(
2       input               clk, reset,
3       input               MemWriteM,
4       input       [15:0]  alu_resultM,
5       output      [15:0]  MemReadDataM,
6       output      [15:0]  alu_resultMout
7   );
8
9   MemoryData  inst_MemoryData(
10      .clk(clk),
11      .reset(reset),
12      .addrM(alu_resultM),
13      .write_en(MemWriteM),
14      .write_dataM(alu_resultM),
15      .read_dataM(MemReadDataM) //out
16  );
17
18  assign alu_resultMout = alu_resultM;
19
20  endmodule
```

입력값

- clk (1 비트): 클럭 신호.
- reset (1 비트): 리셋 신호.
- MemWriteM (1 비트): 메모리 쓰기 신호.
- alu_resultM (16 비트): 메모리 접근 주소 및 쓰기 데이터.

출력값

- MemReadDataM (16 비트): 메모리에서 읽은 데이터.
- alu_resultMout (16 비트): 메모리 접근 주소.

동작 방식

- MemoryData 모듈은 주어진 주소(alu_resultM)에 대해 메모리 읽기 또는 쓰기 동작을 수행한다.
    - MemWriteM 신호가 활성화되면, alu_resultM 주소에 alu_resultM 데이터를 쓴다.
    - MemWriteM 신호가 비활성화되면, alu_resultM 주소의 데이터를 읽어 MemReadDataM 에 저장한다.
- alu_resultM 값은 그대로 alu_resultMout 에 전달된다.

- WriteBack Register 모듈

WriteBackRegister 모듈은 메모리 단계에서 쓰기 단계로 데이터를 전달하는 역할을 한다. 클럭 신호와 리셋 신호에 따라 제어 신호와 데이터를 저장하고 전달한다.

```
1   module WriteBackRegister(
2       input                   clk, reset,
3       input                   MemToRegM, RegWriteM,
4       input       [3:0]       destAddM,
5       input       [15:0]      MemReadDataM,
6       input       [15:0]      alu_resultMin,
7       output  reg             MemToRegW, RegWriteW,
8       output  reg [3:0]       destAddW,
9       output  reg [15:0]      MemReadDataW, alu_resultW
10  );
11
12  always@(posedge clk or negedge reset)begin
13      if(~reset)begin
14          MemToRegW    <= 1'b0;
15          RegWriteW    <= 1'b0;
16          MemReadDataW <= 16'h0000;
17          alu_resultW  <= 16'h0000;
18          destAddW     <= 4'b0000;
19      end
20      else begin
21          MemToRegW    <= MemToRegM;
22          RegWriteW    <= RegWriteM;
23          MemReadDataW <= MemReadDataM;
24          alu_resultW  <= alu_resultMin;
25          destAddW     <= destAddM;
26      end
27  end
28
29  endmodule
```

입력값

- clk (1 비트): 클럭 신호.
- reset (1 비트): 리셋 신호.
- MemToRegM (1 비트): 메모리에서 레지스터로 데이터 이동 신호.
- RegWriteM (1 비트): 레지스터 쓰기 신호.
- destAddM (4 비트): 목적지 주소.
- MemReadDataM (16 비트): 메모리에서 읽은 데이터.
- alu_resultMin (16 비트): ALU 연산 결과.

출력값

- MemToRegW (1 비트): 쓰기 단계로 전달되는 메모리에서 레지스터로 데이터 이동 신호.
- RegWriteW (1 비트): 쓰기 단계로 전달되는 레지스터 쓰기 신호.
- destAddW (4 비트): 쓰기 단계로 전달되는 목적지 주소.
- MemReadDataW (16 비트): 쓰기 단계로 전달되는 메모리에서 읽은 데이터.
- alu_resultW (16 비트): 쓰기 단계로 전달되는 ALU 연산 결과.

동작 방식

- 리셋 신호(reset)가 활성화되면, 모든 출력 신호를 초기화한다.
- 클럭 신호의 상승 모서리에서 입력된 제어 신호(MemToRegM, RegWriteM)와 데이터를(MemReadDataM, alu_resultMin, destAddM) 쓰기 단계로 전달하여 출력 신호(MemToRegW, RegWriteW, MemReadDataW, alu_resultW, destAddW)에 저장한다.

- WriteBack Stage 모듈

WriteBackStage 모듈은 메모리에서 읽은 데이터와 ALU 연산 결과 중에서 하나를 선택하여 최종 결과를 출력하는 역할을 한다.

```
1   module WriteBackStage(
2       input               MemToRegW,
3       input       [15:0]  MemReadDataW, alu_resultW,
4       output      [15:0]  ResultW
5   );
6
7   assign ResultW = MemToRegW ? MemReadDataW : alu_resultW;
8
9   endmodule
```

입력값

- MemToRegW (1 비트): 메모리에서 레지스터로 데이터 이동 신호.
- MemReadDataW (16 비트): 메모리에서 읽은 데이터.
- alu_resultW (16 비트): ALU 연산 결과.

출력값

- ResultW (16 비트): 최종 결과 데이터.

동작 방식

- MemToRegW 신호가 활성화되면, 메모리에서 읽은 데이터(MemReadDataW)를 최종 결과(ResultW)로 출력한다.
- MemToRegW 신호가 비활성화되면, ALU 연산 결과(alu_resultW)를 최종 결과(ResultW)로 출력한다.

- Hazard Unit 모듈

HazardUnit 모듈은 데이터 해저드와 제어 해저드를 감지하고, 파이프라인의 스톨 및 포워딩을 제어하는 역할을 한다

```
module HazardUnit(
    input               branch,
    input               flush,
    input               RegWriteE,
    input               MemToRegE,
    input               immediateD,
    input               forwardD,
    input    [3:0]      srcAdd1, srcAdd2, destAddE,
    input    [15:0]     srcData1, srcData2, alu_resultE,
    output   reg        stallF, stallD,
    output   reg        forwardA, forwardB,
    output   reg        flushD, flushE,
    output   reg        InstBranch
);

reg lwstall;

always@(*)begin
    if(branch && (srcData1 != srcData2) && alu_resultE)
        InstBranch = 1'b1;
    else
        InstBranch = 1'b0;
end

always@(*)begin
    if(MemToRegE && ((srcAdd1 == destAddE)||(srcAdd2 == destAddE)))
        lwstall = 1'b1;
    else
        lwstall = 1'b0;
end

always@(*)begin
    if(forwardD && RegWriteE && (srcAdd1 == destAddE))
        forwardA = 1'b1;
    else
        forwardA = 1'b0;
end

always@(*)begin
    if(!immediateD && forwardD && RegWriteE && (srcAdd2 == destAddE))
        forwardB = 1'b1;
    else
        forwardB = 1'b0;
end

always@(*)begin
    if(lwstall || InstBranch)begin
        stallF = 1'b1;
        stallD = 1'b1;
    end
    else begin
        stallF = 1'b0;
        stallD = 1'b0;
    end
end

always@(*)begin
    if(lwstall || (InstBranch &&flush))
        flushE = 1'b1;
    else
        flushE = 1'b0;
end

always@(*)begin
    if(InstBranch)begin
        flushD = 1'b1;
    end
    else begin
        flushD = 1'b0;
    end
end

endmodule
```

입력값

- branch (1 비트): 브랜치 명령어 신호.
- flush (1 비트): 플러시 신호.
- RegWriteE (1 비트): 실행 단계의 레지스터 쓰기 신호.
- MemToRegE (1 비트): 실행 단계의 메모리에서 레지스터로 데이터 이동 신호.
- immediateD (1 비트): 디코드 단계의 immediate value 사용 신호.
- forwardD (1 비트): 디코드 단계의 포워딩 신호.
- srcAdd1 (4 비트): 첫 번째 소스 주소.
- srcAdd2 (4 비트): 두 번째 소스 주소.
- destAddE (4 비트): 실행 단계의 목적지 주소.
- srcData1 (16 비트): 첫 번째 소스 데이터.
- srcData2 (16 비트): 두 번째 소스 데이터.
- alu_resultE (16 비트): 실행 단계의 ALU 연산 결과.

출력값

- stallF (1 비트): 페치 단계의 스톨 신호.
- stallD (1 비트): 디코드 단계의 스톨 신호.
- forwardA (1 비트): 첫 번째 포워딩 신호.
- forwardB (1 비트): 두 번째 포워딩 신호.
- flushD (1 비트): 디코드 단계의 플러시 신호.
- flushE (1 비트): 실행 단계의 플러시 신호.
- InstBranch (1 비트): 브랜치 명령어 신호.

동작 방식

1. **브랜치 감지**:
    - branch 신호가 활성화되고, srcData1 과 srcData2 가 다르며, alu_resultE 가 참인 경우 InstBranch 신호를 1 로 설정한다.
    - 그렇지 않은 경우 InstBranch 신호를 0 으로 설정한다.
2. **로드-유효성 스톨 감지**:
    - MemToRegE 신호가 활성화되고, srcAdd1 또는 srcAdd2 가 destAddE 와 같으면 lwstall 신호를 1 로 설정한다.
    - 그렇지 않은 경우 lwstall 신호를 0 으로 설정한다.
3. **포워딩 신호 설정**:
    - forwardD 신호가 활성화되고, RegWriteE 신호가 활성화되며, srcAdd1 가 destAddE 와 같으면 forwardA 신호를 1 로 설정한다.
    - 그렇지 않은 경우 forwardA 신호를 0 으로 설정한다.

- immediateD 신호가 비활성화되고, forwardD 신호가 활성화되며, RegWriteE 신호가 활성화되고, srcAdd2 가 destAddE 와 같으면 forwardB 신호를 1 로 설정한다.
- 그렇지 않은 경우 forwardB 신호를 0 으로 설정한다.

4. **스톨 신호 설정**:
   - lwstall 신호가 활성화되거나 InstBranch 신호가 활성화된 경우 stallF 와 stallD 신호를 1 로 설정한다.
   - 그렇지 않은 경우 stallF 와 stallD 신호를 0 으로 설정한다.
5. **플러시 신호 설정**:
   - lwstall 신호가 활성화되거나 InstBranch 신호와 flush 신호가 동시에 활성화된 경우 flushE 신호를 1 로 설정한다.
   - 그렇지 않은 경우 flushE 신호를 0 으로 설정한다.
   - InstBranch 신호가 활성화된 경우 flushD 신호를 1 로 설정한다.
   - 그렇지 않은 경우 flushD 신호를 0 으로 설정한다.

- Control Unit 모듈

ControlUnit 모듈은 입력된 오피코드(i_opcode)를 해석하여 각종 제어 신호를 생성하는 역할을 한다.

```
module ControlUnit(
    input  [3:0]     i_opcode,
    output          branchD, flushD, RegWriteD, MemWriteD, MemToRegD, immediateD, forwardD,
    output [1:0]    alufuncD
);

OpcodeDecoder   inst_OpcodeDecoder(
    .i_opcode(i_opcode),
    .branch(branchD),
    .flush(flushD),
    .RegWrite(RegWriteD),
    .MemWrite(MemWriteD),
    .MemToReg(MemToRegD),
    .immediate(immediateD),
    .forward(forwardD),
    .o_alufunc(alufuncD)
);

endmodule
```

입력값

- i_opcode (4 비트): 명령어의 오피코드.

출력값

- branchD (1 비트): 브랜치 신호.
- flushD (1 비트): 플러시 신호.
- RegWriteD (1 비트): 레지스터 쓰기 신호.
- MemWriteD (1 비트): 메모리 쓰기 신호.
- MemToRegD (1 비트): 메모리에서 레지스터로 데이터 이동 신호.
- immediateD (1 비트): immediate value 사용 신호.
- forwardD (1 비트): 포워딩 신호.
- alufuncD (2 비트): ALU 연산 함수.

동작 방식

- ControlUnit 모듈은 OpcodeDecoder 모듈을 인스턴스화하여 i_opcode 를 해석하고, 각종 제어 신호들을 생성한다.
- OpcodeDecoder 모듈에서 생성된 제어 신호들은 각각 branchD, flushD, RegWriteD, MemWriteD, MemToRegD, immediateD, forwardD, alufuncD 로 출력된다.

- Data Path 모듈

DataPath 모듈은 CPU의 각 단계를 포함하여 명령어의 Fetch, decode, execute, memory access, Write back 단계를 처리하고, 각 단계의 데이터를 연결하는 역할을 한다.

입력값

- clk (1 비트): 클럭 신호.
- reset (1 비트): 리셋 신호.
- stallF (1 비트): 페치 단계의 스톨 신호.
- stallD (1 비트): 디코드 단계의 스톨 신호.
- forwardA (1 비트): 포워딩 신호 A.
- forwardB (1 비트): 포워딩 신호 B.
- InstBranch (1 비트): 브랜치 명령어 신호.
- flushD (1 비트): 디코드 단계의 플러시 신호.
- flushC (1 비트): 실행 단계의 플러시 신호.
- RegWriteC (1 비트): 실행 단계의 레지스터 쓰기 신호.
- MemWriteC (1 비트): 실행 단계의 메모리 쓰기 신호.
- MemToRegC (1 비트): 실행 단계의 메모리에서 레지스터로 데이터 이동 신호.
- immediateC (1 비트): 실행 단계의 immediate value 사용 신호.
- alufuncC (2 비트): 실행 단계의 ALU 연산 함수.

출력값

- RegWriteE (1 비트): 디코드 단계의 레지스터 쓰기 신호.
- MemToRegE (1 비트): 디코드 단계의 메모리에서 레지스터로 데이터 이동 신호.
- srcAdd1 (4 비트): 첫 번째 소스 주소.
- srcAdd2 (4 비트): 두 번째 소스 주소.
- destAddE (4 비트): 목적지 주소.
- opcodeDP (4 비트): 명령어의 오피코드.
- ResultW (16 비트): 쓰기 단계의 결과 데이터.
- RegDataA (16 비트): 첫 번째 레지스터 데이터.
- RegDataB (16 비트): 두 번째 레지스터 데이터.
- alu_resultE (16 비트): 실행 단계의 ALU 결과.

동작 방식

1. **Fetch 단계**:
    - ProgramCounter 모듈은 현재 PC 값을 관리하고 업데이트한다.
    - FetchStage 모듈은 현재 PC 값에 해당하는 명령어를 가져온다.
2. **Decode 단계**:
    - DecodeRegister 모듈은 페치 단계의 명령어와 PC 값을 저장하고 전달한다.
    - DecodeStage 모듈은 명령어를 해석하고, 제어 신호를 생성하며, 레지스터 파일에서 데이터를 읽어온다.
3. **Execute 단계**:
    - ExcuteRegister 모듈은 디코드 단계의 데이터를 저장하고 전달한다.
    - ExcuteStage 모듈은 ALU 를 사용하여 연산을 수행한다.
4. **Memory Access 단계**:
    - MemoryRegister 모듈은 실행 단계의 데이터를 저장하고 전달한다.
    - MemoryStage 모듈은 메모리 접근을 수행한다.
5. **Write Back 단계**:
    - WriteBackRegister 모듈은 메모리 단계의 데이터를 저장하고 전달한다.
    - WriteBackStage 모듈은 데이터를 레지스터 파일에 기록한다.

- **Top 모듈**

CPU 모듈은 전체 모듈들의 최상위 모듈로, 각 단계의 하위 모듈을 연결하고 전체적인 데이터 흐름과 제어 신호를 관리하는 역할을 한다.

```
module Team4_CPU(
    input               clk, reset,
    output  [15:0]      DataOut
);

wire                branchC, flushC, RegWriteC, MemWriteC, MemToRegC,
                    immediateC, forwardC, MemToRegE, RegWriteE,
                    stallF, stallD, flushD,
                    forwardA, forwardB,
                    flushE, InstBranch;
wire    [1:0]       alufuncC;
wire    [3:0]       opcodeDP, srcAdd1, srcAdd2, destAddE;
wire    [15:0]      RegDataA, RegDataB, alu_resultE;

DataPath        inst_DataPath(
    .clk(clk),
    .reset(reset),
    .flushC(flushE),
    .flushD(flushD),
    .RegWriteC(RegWriteC),
    .MemWriteC(MemWriteC),
    .MemToRegC(MemToRegC),
    .immediateC(immediateC),
    .alufuncC(alufuncC),
    .stallF(stallF),
    .stallD(stallD),
    .forwardA(forwardA),
    .forwardB(forwardB),
    .InstBranch(InstBranch),
    .opcodeDP(opcodeDP), //out
    .ResultW(DataOut),
    .RegWriteE(RegWriteE),
    .MemToRegE(MemToRegE),
    .srcAdd1(srcAdd1),
    .srcAdd2(srcAdd2),
    .destAddE(destAddE),
    .RegDataA(RegDataA),
    .RegDataB(RegDataB),
    .alu_resultE(alu_resultE)
);

ControlUnit     inst_ControlUnit(
    .i_opcode(opcodeDP),
    .branchD(branchC), //out
    .flushD(flushC),
    .RegWriteD(RegWriteC),
    .MemWriteD(MemWriteC),
    .MemToRegD(MemToRegC),
    .immediateD(immediateC),
    .forwardD(forwardC),
    .alufuncD(alufuncC)
);

HazardUnit      inst_HazardUnit(
    .branch(branchC),
    .flush(flushC),
    .RegWriteE(RegWriteE),
    .MemToRegE(MemToRegE),
    .immediateD(immediateC),
    .forwardD(forwardC),
    .srcAdd1(srcAdd1),
    .srcAdd2(srcAdd2),
    .destAddE(destAddE),
    .srcData1(RegDataA),
    .srcData2(RegDataB),
    .alu_resultE(alu_resultE),
    .stallF(stallF), //out
    .stallD(stallD),
    .forwardA(forwardA),
    .forwardB(forwardB),
    .flushE(flushE),
    .flushD(flushD),
    .InstBranch(InstBranch)
);

endmodule
```

(*코드는 각 모듈들을 인스턴스화하여 사용하였으므로 이에 대한 자세한 설명은 생략한다)

4. 테스트 및 검증

A. 테스트 벤치 설계

```
1   `timescale 1ns/1ps
2   module  Team4_CPU_tb();
3
4   reg             clk, reset;
5   wire    [15:0]  DataOut;
6
7   Team4_CPU   inst_CPU(
8       .clk(clk),
9       .reset(reset),
10      .DataOut(DataOut)
11  );
12
13  initial begin
14      clk = 0; reset = 1;
15      #10
16      reset = 0;
17      #10
18      reset = 1;
19      #1500
20      $finish;
21  end
22
23  always begin
24      #50
25      clk = ~clk;
26  end
27
28  endmodule
```

Output : WriteBackStage의 Stage

Input : clk, reset, enable

초기화 블록 (initial block)

- 초기화 블록에서는 클럭과 리셋 신호의 초기값을 설정하고, 리셋 신호를 제어하여 초기화를 수행
  - 초기화 단계에서 clk 는 0, reset 은 1 로 설정.
  - 10 ns 후 reset 을 0 으로 설정하여 리셋을 해제
  - 다시 10 ns 후 reset 을 1 로 설정
  - 1500 ns 후 시뮬레이션을 종료

반복 블록 (always block)

- 클럭 신호는 반복 블록 안에서 50 ns 마다 반전됨.
  100ns clk cycle 로 구성

B. Instruction Set

A. LDA(Data forward)

```
//Datafoward with LDA
initial begin
    instM[0] = 16'b0000_0000_1111_0001; //r1 = mem[15](15)
    instM[1] = 16'b0110_1000_0000_1000; //unused
    instM[2] = 16'b0110_1000_0000_1000; //unused
    instM[3] = 16'b0110_0001_0010_0010; //r2 = r1 + 2
    instM[4] = 16'b0110_0010_0001_0011; //r3 = r2 + 1
    instM[5] = 16'b0110_0011_0010_0100; //r4 = r3 + 2
    instM[6] = 16'b0110_0100_0001_0101; //r5 = r4 + 1
end
```

ALU 의 결과 r0 + 15 = 15, 따라서 Memory 의 15 번지로 가서 data15 를 가져와서 r1 에 저장한다. r1 = mem[15] 그 이후의 stal 이 필요하나 구현하지 못해서 unused instruction 사용하여 2clock delay 후에 가져온 r1 을 사용해 Dataforward 검증

B. Branch

```
//Branch
initial begin
    instM[0] = 16'b0110_0001_0100_0001; //r1 = r1 + 4
    instM[1] = 16'b0110_0010_0100_0010; //r2 = r2 + 4
    instM[2] = 16'b0110_0011_0001_0011; //r3 = r3 + 1
    instM[3] = 16'b0110_0100_0001_0100; //r4 = r4 + 1
    instM[4] = 16'b0111_0001_0001_0001; //r1 = r1 - 1
    instM[5] = 16'b1001_0001_0001_0101; //BNE r1, 1
    //if(r1=1)
    instM[6] = 16'b0110_0001_0100_0010; //r2 = r1 + 4
    instM[7] = 16'b0110_0010_0100_0011; //r3 = r2 + 4
    instM[8] = 16'b0110_1111_0000_1111; //unused
    instM[9] = 16'b0110_1111_0000_1111; //unused
    instM[10] = 16'b0010_0011_0010_0100; //r4 = r3 + r2
end
```

r1 에 4 를 저장하고 BNE r1 -1 을 이용해 4 번의 loop 구현 loop 문의 결과 :

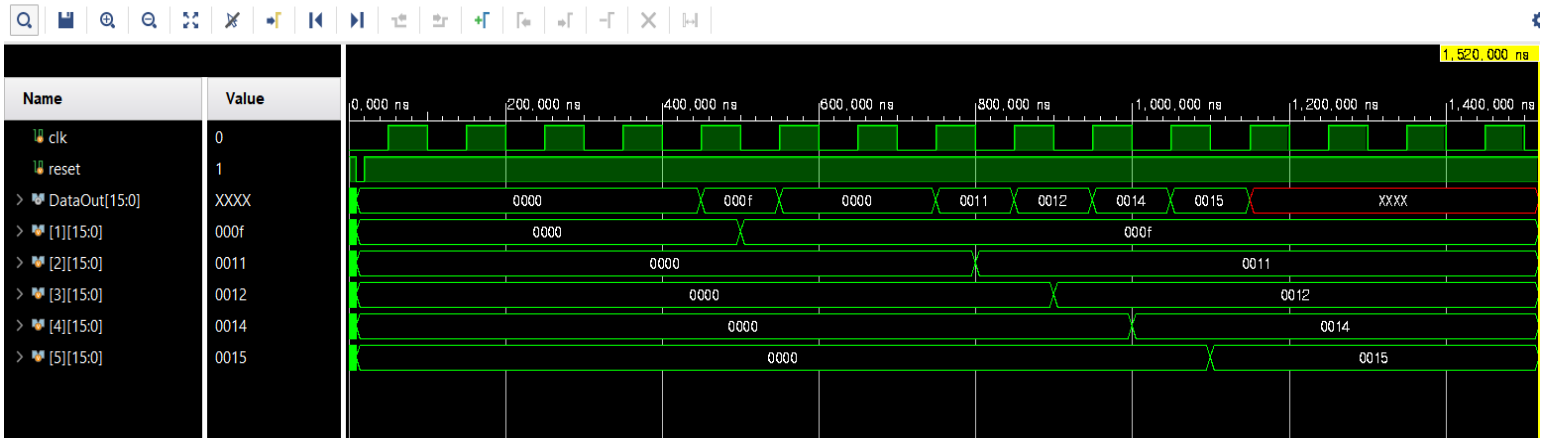
r2 = 4 + 4 + 4 +4 / r3 = 1 + 1 + 1 +1 / r4 = 1 + 1 + 1 + 1

각각 r2 에 0010, r3, r4 에 0004 저장

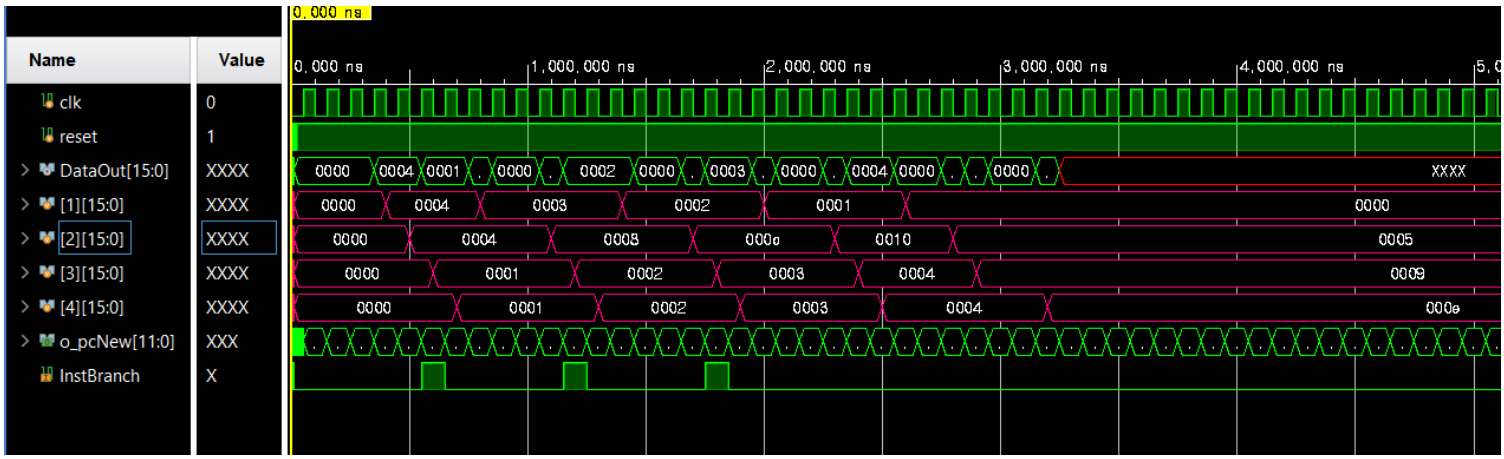
Loop 문을 빠져나온 후에 연산 값 Register 에 저장 완료

->r2 = r1(1) + 4 = 5 / r3 = r2(5) + 4 = 9 / r4 = r3(9) + r2(5) = 14

C. TestBench Waveform



(1) LDA Waveform



(2) Branch Waveform

5. Synthesis 결과

Area summary

Summary

Resource	Utilization	Available	Utilization %
LUT	2814	63400	4.44
FF	8690	126800	6.85
DSP	1	240	0.42
IO	18	210	8.57

Area hierarchy

Hierarchy

Name	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	DSPs (240)	Bonded IOB (210)	BUFGCTRL (32)
Team4_CPU	2814	8690	1136	512	1	18	1
inst_HazardUnit (HazardUnit)	0	0	0	0	0	0	0
inst_DataPath (DataPath)	2814	8690	1136	512	1	0	0

Timing summary

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	inf	Worst Hold Slack (WHS):	inf	Worst Pulse Width Slack (WPWS):	NA
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	NA
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	NA
Total Number of Endpoints:	25866	Total Number of Endpoints:	25866	Total Number of Endpoints:	NA

There are no user specified timing constraints.

6. 트러블슈팅 및 해결 과정

Trouble1: PC값이 2clk주기로 pc + 1 되어 정상적으로 움직이지 않는 상황 발생

Situation Simulation 결과 PC값을 전달해주는 모듈이 DecodeRegister여서 발생하던 문제임을 확인 할 수 있었고 Decode단에서 PC값을 feedback 해주는 것이 문제라는 것을 알 수 있었음.

Solution : FetchStage에서 PC값을 Fetch Mux로 보내주도록 수정

Trouble2 : ProgramCounter에서 Branch기능이 구현이 되지 않는 현상 발생

Situation. Simulation결과 동작시에 ProgramCounter에 Stall이 걸려있어서 BranchPC값을 받지 못해 Branch기능이 구현되지 않는 현상임을 알 수 있었음.

MUX(PC_branch, pc_F)가 Combinational Circuit으로 구현이 되어있어서 HazardUnit이 주는 신호(stall, InstBranch)를 디버깅하는 것이 쉽지 않았음

Solution : MUX를 제거하고 FetchRegister Logic 수정으로 해결하였음. Combinational인 MUX의 디버깅이 쉽지 않아 FetchRegister로 PC값을 받도록 하고 우선순위 if문을 이용해서 stall(!enable) 일때도 InstBranch신호를 받아 PC_branch값을 출력할 수 있도록 하였다.

7. 느낀점 및 고찰

CPU의 Dataflow 구현이 간단하지 않다는 것을 느꼈고 단순 verilog 학습도 중요하지만 하드웨어적인 지식도 충분히 학습이 되어야 더 큰 단위의 프로젝트들을 진행할 수 있음을 깨달았다.

설계 과정에서 16bit processor로는 구현할 수 있는 기능이 제한적이고 구현할 수 있는 기능들도 여러 제약이 따르게 될 수 밖에 없었다. 현재 구현해본 16bit CPU를 기반으로 32bit processor로 확장해서 여러 기능들을 구현해 보는 것도 도움이 되겠다고 느낀다.