CSED311 Computer Architecture

Assignment 4-1 Report: Pipelined-cycle CPU

TEAM 28

20180154 **전제민** jeminjeon@postech.ac.kr

20180916 **임경빈** kbini@postech.ac.kr

2023/05/01

1. Introduction

이번 Lab4-1을 수행하며 Pipelined Multi-cycle CPU를 RTL로 구현하였다. Pipelined Multi -cycle CPU는 명령어 수행을 효율적으로 하기 위해 하나의 명령어 수행을 여러 단계로 나누고 각 단계에서 동시에 다른 명령어를 처리하도록 설계된 CPU를 말한다. Lab4에서는 Non-control flow 명령어를 지원하도록 구현하였다.

이번 보고서에서는 Pipelined Multi-cycle CPU의 논리 회로 설계를 다루고자 한다.

- **Design 파트에서는** 각 모듈의 설계, 파이프라인 레지스터 설계, hazard detection 설계, data forwarding에 대한 설계에 대해 설명하였다.
- **Implementation 파트에서는** 각 모듈의 구체적인 구현을 설명하였다. 또한 hazard detection이 언제 어떻게 이루어지는지, data forwarding이 언제 어떻게 이루어지는지 구체적인 방안을 설명하였다.
- **Discussion 파트에서는** Pipeline 설계의 장점, Single-cycle CPU와 Pipelined CPU의 실행 사이클 수 차이를 비교하였다.
- **Conclusion 파트에서는** 이번 구현을 통해 새롭게 알거나 깨닫게 된 점을 간단히 정리하였다.

2. Design

(1) Designs of Each Module

- **CPU**: Pipelined CPU를 이루는 각 하위 모듈들을 연결시키는 상위 모듈이다. 또한 Control Unit에서 나온 control flag에 따라 MUX의 output을 지정하는 로직도 포함하고 있다. top.v에서 발생시키는 clk cycle과 reset signal을 전달하며 cpu.v.에서 output으로 나온 is_halted에 따라 cpu를 종료시킨다.

- **PC**: 클럭 사이클에 따라 동기화되는 Synchronous 모듈이며, next_pc를 입력받아 PC를 업데이트하는 역할을 하는 모듈이다.

- **InstMemory**: 클럭 사이클에 따라 동기화되는 Synchronous 모듈이며, 입력하는 PC에 따른 Instruction을 output으로 내보낸다.

- **DataMemory**: 클럭 사이클에 따라 동기화되는 Synchronous 모듈이며, MemRead 또는 MemWrite 컨트롤에 따라서 입력 받은 주소에 해당하는 데이터를 출력하거나 입력 받은 주소에 입력 데이터를 작성하는 역할을 하는 모듈이다. 이전 Multi-cycle CPU에서는 Instruction과 Data Memory가 통합되었으나 이번 Lab에서는 파이프라인 구현을 위해 다시 분리되었다.

- **RegisterFile**: 클럭 사이클에 따라 동기화되는 Synchronous 모듈이며, 입력 받은 번호에 해당하는 레지스터 값을 출력하거나, 또는 RegWrite 컨트롤 플래그에 따라 입력 받은 데이터를 destination 레지스터에 작성하는 역할을 한다. 또한 Ecall 인스트럭션이 들어올 것을 대비해 x17 레지스터의 값을 확인하는 역할도 수행한다.

- **ALU**: Asynchronous 모듈이며 두 개의 input을 이용해 연산을 한다. 이때 ALUControl 모듈에서 가져온 연산 코드를 이용해 적절한 연산이 이루어지도록 한다. 만약 Branch 인스트럭션이 들어왔을 경우 두 레지스터의 값을 비교하여 적절하게 bcond 플래그를 set하는 역할도 수행한다.

- **ALUControl**: Asynchronous 모듈이며 opcode를 이용해 인스트럭션마다 적절한 ALU 연산이 이루어지도록 어떤 연산을 해야 하는 지 결정하는 로직이 존재하는 모듈이다.

- **ImmediateGenerator**: Asynchronous 모듈이며 인스트럭션을 읽어 타입에 따라 적절하게 Immediate 값을 만들어내는 모듈이다.

- **HazardControl**: 파이프라인을 구현함에 따라 발생할 수 있는 Data Hazard를 예방하기 위해 ID stage에서 읽어오고자 하는 레지스터가 이전 명령어에서 업데이트할 예정인 레지스터일 경우 이를 감지하여 Data forwarding을 수행하도록 하도록 컨트롤 플래그를 output으로 내보내는 모듈이다. 구체적인 로직은 아래

Hazard Detection Design에서 설명할 예정이다.

- **ForwardingUnit**: EX에서 사용할 rs1, rs2와 EX/MEM 레지스터에 있는 RD, MEM/WB 레지스터에 있는 RD를 비교하고 만약 Forwarding이 필요할 경우 이에 맞게 Forward 컨트롤 시그널을 output으로 내보내는 모듈이다. 구체적인 로직은 아래 Data Forwarding Design에서 설명할 예정이다.

(2) Pipeline register design

각 스테이지 사이에 이전 스테이지에서 결정된 값들을 저장하는 파이프라인 레지스터를 만들어 저장한다.

- IF/ID register: IF에서 PC를 이용해 fetch한 instruction을 저장한다.
- ID/EX register: instruction으로부터 가져온 접근할 레지스터 넘버, 레지스터 파일로부터 읽어온 레지스터 값, immediate, ALU control unit에 들어갈 input, control unit에서 나온 컨트롤 시그널들을 저장한다.
- EX/MEM register: ALU에서 계산한 값, destination register number, rs2로부터 나온 데이터 메모리에 작성할 데이터를 저장한다.
- MEM/WB register: destination register number, 레지스터에 저장할 데이터를 저장한다.

(3) Control signal design

ID stage에서 Instruction을 읽고 이에 해당하는 Control Signal을 파이프라인 레지스터에 저장한다. ID/EX 레지스터에는 EX, MEM, WB에서 사용하는 Control Signal을 모두 갖고 있고, EX/MEM 레지스터에는 MEM과 WB에서 사용하는 Control Signal을, MEM/WB 레지스터에서는 WB에서 사용하는 Control Signal을 저장한다. 파이프라인 단계가 넘어갈 때마다 현재 스테이지에서 처리하는 명령어에 맞는 control signal이 이동할 수 있도록 한다.

- **EX stage에서 사용하는 Control Signal**

  alu_op: ALU에서 수행할 연산을 전달하는 컨트롤 플래그이다.

  alu_src: ALU에서 operand로 사용할 값을 어디서 가져올지 결정하는 컨트롤 플래그이다.

- **MEM stage에서 사용하는 Control Signal**

  mem_write: 데이터 메모리에 값을 write할 경우 set되는 컨트를 플래그이다.

  mem_read: 데이터 메모리로부터 값을 읽어올 경우 set되는 컨트롤 플래그이다.

- **WB stage에서 사용하는 Control Signal**

  mem_to_reg : Load 명령어의 경우 데이터 메모리에서 읽어온 값을 레지스터에 작성할 것인지 여부를 나타낸다.

  reg_write : 레지스터에 값을 작성할 것인지 여부를 나타낸다.

pc_to_reg와 같은 컨트롤 시그널도 있으나 Control flow 명령어를 다루는 다음 Lab에서 사용할 예정이다.

(4) Hazard detection design

Hazard detection 로직은 HazardControl에서 정의하였다. 우선 instruction을 가져와서 rs1, rs2, opcode를 구한다. opcode에 따라 instruction이 rs1을 사용하고 레지스터 넘버가 0이 아님을 나타내는 use_rs1() function 값을 결정한다. Arithmetic, Store, Branch 타입의 경우 rs1과 rs2를 둘다 사용하는데, 만약 이들의 넘버가 0이 아닌 경우 use_rs1과 use_rs2를 모두 1로 set하는 방식이다. Arithmetic_imm, Load, JALR은 rs1만 사용되므로 rs1이 0이 아닌 경우 use_rs1은 1로 set 된다. JAL은 레지스터 값을 사용하지 않으므로 use_rs1과 use_rs2 값이 모두 0이다. 만약 rs1 또는 rs2를 rs라고 하였을 때 Instruction에서 읽어온 rs가 EX 또는 MEM 단계에서 사용하는 rd와 같고, use_rs가 1이고, 해당 스테이지에서 Reg_write가 1일 경우 is_stall 플래그를 set하여 내보낸다. 구체적인 Implementation은 3. Implementation 파트에서 상세히 기술하였다.

(5) Data forwarding design

Data Forwarding 로직은 ForwardingUnit에서 정의하였다. EX에서 사용할 rs1, rs2 그리고 EX/MEM과 MEM/WB 레지스터에 각각 저장되어 있는 rd 레지스터 넘버와 reg_write 컨트롤 시그널을 input으로 받는다. 만약 rs1과 rs2가 0이 아니고, 각 파이프라인 레지스터에 저장되어 있는 rd과 같으며 해당 파이프라인에서 수행 중인 인스트럭션이 rd에 값을 write할 예정일 경우, 즉 reg_write가 1인 경우 이에 해당하는 Forward 플래그를 output으로 내보내 적절한 데이터 포워딩이 이루어질 수 있도록 하였다. 이때 MEM단계, WB 단계 순서로 매치 여부를 살펴서 youngest instruction을 우선적으로 체크하도록 하였다. 구체적인 Implementation은 3. Implementation 파트에서 상세히 기술하였다.

3. Implementation

5- stage pipelined cpu를 구현하였다. 모든 인스트럭션들은 IF, ID, EX, MEM, WB stage를 거친다. 이 단계들은 파이프라인 레지스터들을 이용해 구현하였다.

(1) Cpu implementation

우리가 구현한 pipeline cpu의 Top module 이다. 파이프라인 레지스터들을 구현하였고 파이프라인 레지스터들 간의 연결, 데이터가 어떻게 이동하는지를 구현하였다. 또한 하위 모듈들인 PC모듈, Memory 모듈, RegisterFile 모듈, ControlUnit 모듈, HazardControlUnit 모듈, ForwardingUnit모듈, Immediate Generator 모듈, ALUControlUnit 모듈, ALU 모듈, DataMemory 등 하위모듈들의 연결을 여기서 구현하였다.

Reset signal, clock signal을 input으로 받는다. Output으로는 is_halted를 가진다. 그리고 reset signal, clock signal을 다른 모듈들로 전달함으로써 레지스터 값들을 초기화한다.

(2) Pipeline register implementation

Pipeline cpu구현에 사용된 Pipeline register들은 다음과 같다.

<IF_ID pipeline registers>

- IF_ID_inst : current pc 로 Instruction memory에서 fetch한 인스트럭션이 저장된다.

<ID_EX pipeine registers>

Control unit의 출력이 write되는 pipeline register 이다.

- ID_EX_alu_op : Control unit에서 instruction을 통해 alu의 어떤 오퍼레이션을 해야 되는지 알려준다.
- ID_EX_alu_src : Control unit에서 IF_ID_inst을 통해 결정된다. 이후에 EX_stage에서 ID_EX_alu_src Mux에서 사용된다.
- ID_EX_mem_write : Control unit에서 IF_ID_inst를 통해 결정된다. 메모리에 write하는지 여부를 나타낸다.
- ID_EX_mem_read : Control unit에서 IF_ID_inst를 통해 결정된다. 메모리에서 read 하는지 여부를 나타낸다.

- ID_EX_mem_to_reg : Control unit에서 IF_ID_inst를 통해 결정된다. Reg_write_data가 메모리에서 읽은 데이터를 register에 write해야 되는지 여부를 나타낸다.
- ID_EX_reg_write : Control unit에서 IF_ID_inst를 통해 결정된다. Destination Register에 write하는지 여부를 나타낸다.

Control unit과 관계없는 pipeline register이다.

- ID_EX_rs1_data : rs1에서 읽은 데이터를 저장한다.
- ID_EX_rs2_data : rs2에서 읽은 데이터를 저장한다.
- ID_EX_imm : immediate generator로 생성된 immediate value를 저장한다.
- ID_EX_ALU_ctrl_unit_input : ALU_control_unit의 입력으로 사용된다.
- ID_EX_rd : rd(destination register)를 저장한다.
- ID_EX_rs1 : rs1을 저장한다.
- ID_EX_rs2 : rs2을 저장한다.
- ID_EX_HALT : HALT하는지 여부를 나타낸다.

<EX_MEM pipeline registers>

Control unit의 출력이 write되는 pipeline register 이다.

- EX_MEM_mem_write : ID_EX_mem_write의 값이 이동해온다. 메모리에 write하는지 여부를 나타낸다.
- EX_MEM_mem_read : ID_EX_mem_read의 값이 이동해온다. 메모리에서 read 하는지 여부를 나타낸다.
- EX_MEM_is_branch : ALU의 bcond의 값을 가져온다. 브랜치 해야 되는지 여부를 나타낸다. 본 pipeline cpu 구현에서는 사용하지 않았다.
- EX_MEM_mem_to_reg : . ID_EX_mem_to_reg의 값을 가져온다. Reg_write_data가 메모리에서 읽은 데이터를 register에 write해야 되는지 여부를 나타낸다.
- EX_MEM_reg_write : ID_EX_reg_write의 값을 가져온다. Destination Register에 write하는지 여부를 나타낸다.

Control unit과 관계없는 pipeline register이다.

- EX_MEM_alu_out : ALU의 결과를 가져와 저장한다.
- EX_MEM_dmem_data : alu_in_2를 가져온다. 이후에 설명 될 ForwardB MUX의 출력 값을 가져온다. 메모리에 저장 할 데이터를 의미한다.
- EX_MEM_rd : ID_EX_rd의 값을 가져와 저장한다. Destination register 번호를 나타낸다.
- EX_MEM_HALT : ID_EX_HALT의 값을 가져와 저장한다. HALT하는지 여부를 나타낸다.

<MEM_WB pipeline registers>

Control unit의 출력이 write되는 pipeline register 이다.

- MEM_WB_mem_to_reg : EX_MEM_mem_to_reg에서 값을 가져와 저장한다. Reg_write_data가 메모리에서 읽은 데이터를 register에 write해야 되는지 여부를 나타낸다.
- MEM_WB_reg_write : EX_MEM_reg_write에서 값을 가져와 저장한다. Destination Register에 write하는지 여부를 나타낸다.

Control unit과 관계없는 pipeline register이다.

- MEM_WB_mem_to_reg_src_1 : 메모리의 출력인 dmem_out값을 가져와 저장한다.
- MEM_WB_mem_to_reg_src_2 : EX_MEM_alu_out값을 가져와 저장한다.
- MEM_WB_rd : EX_MEM_rd에서 값을 가져와 저장한다. Destination register 번호를 나타낸다.
- MEM_WB_HALT : EX_MEM_HALT의 값을 가져와 저장한다. HALT하는지 여부를 나타낸다.

모든 Pipeline register들의 write는 clk synchronous하게 일어나도록 구현하였다. 이를 위해 임시로 파이프라인 레지스터에 write 될 값을 들고 있다가 clk rising edge일 때 write하도록 구현하였다.

(3) **Control signal implementation**

<Control Unit & pipeline registers>

ID stage의 ControlUnit module에서 control signal들을 생성하여 ID_EX파이프라인 레지스터에 update하도록 구현하였다. IF_ID_inst, is_stall을 입력으로 받고 IF_ID_inst값을 가져와 opcode부분을 통해 이 인스트럭션이 어떤 타입의 인스트럭션인지 구분하고 각 인스트럭션 별로 control signal 들을 생성하여 내보낸다. Is_stall이 만약 1이라면 mem_write, reg_write를 0으로 내보내 이 인스트럭션이 이후 스테이지들에서 bubble이 되도록 구현하였다.

Control unit에서 생성된 컨트롤 시그널들은 ID_EX, EX_MEM, MEM_WB 파이프라인 레지스터를 거쳐 이동한다. Reg_write는 EX_MEM, MEM_WB까지 계속 이동되고 mem_read, mem_write는 ID_EX 파이프라인 레지스터를 거쳐 EX_MEM 파이프라인 레지스터까지만 이동되며 mem_to_reg, reg_write는 ID_EX 파이프라인 레지스터, EX_MEM파이프라인 레지스터를 거쳐 MEM_WB 파이프라인 레지스터 까지 이동한다.

(4) **Hazard detection implementation**

1) Load instruction related data hazard control

Data forwarding을 통해 alu결과를 EX_MEM pipeline register, MEM_WB pipeline register에서 바로 받아와서 ALU모듈의 입력으로 사용 가능 하기 때문에 ALU, ALU_immi 인스트럭션 에 의한 데이터 해저드는 모두 해결된다. 그 외 Data hazard가 발생하는 경우는 바로 앞 인스트럭션에서 memory에서 읽어서 register에 write 하는 값을 read해야 하는 상황이다. 따라서 ID stage에 Hazard Control unit 모듈을 두고 이 모듈에서 RAW 데이터 해저드 발생 여부를 판단하고 데이터 해저드가 발생 할 시 stall 하도록 구현하였다. Hazard Control Unit은 IF_ID 파이프라인 레지스터인 IF_ID_inst, ID_EX 파이프라인 레지스터의 ID_EX_rd, ID_EX_mem_read 레지스터 값을 입력으로 가진다. 그리고 인스트럭션에서 rs1, rs2 각각이 사용되는지 여부를 판단하여 만약 rs1을 이 인스트럭션이 사용하고 rs1이 0이 아니라면 use_rs1 register값이 1을 가지게 한다. 그리고 만약 rs1 이 ID_EX 파이프라인 레지스터의 rd값과 같고 ID_EX_mem_read가 1이고 use_rs1이 1이라면 stall신호를 1로 내보낸다. 여기서 내보내진 stall 신호는 ControlUnit의 인풋으로 들어가 ControlUnit에서 control signal들을 0으로 내보내 stall이 이뤄지도록 구현하였다. 그리고 mux를 통해 IF_ID_inst도 stall이 1이라면 기존 값을 그대로 들고 있도록 구현하였고 current_pc도 stall이 1일 때는 그대로 유지되도록 구현하였다. Rs2에 대해서도 같은 방식으로 stall 신호를 내보내도록

구현하였다.

2) Ecall related data hazard control

또한 ecall instruction이 ID stage에서 수행 중이라면 id_ex_mem_read, id_ex_rd를 통해 ecall 바로 앞에서 수행되고 있는 인스트럭션이 load instruction인지, rd가 17인지 판단한다. 그리고 바로 앞 인스트럭션이 load이고 rd가 17번 레지스터라면 stall신호를 1로 내보내 control unit을 통해 stall하게 하였다. 또한 ex_mem_mem_read, ex_mem_rd를 통해 두 단계 앞(MEM stage)에서 실행 중인 인스트럭션이 load이고 rd가 17번 레지스터인지 판단하여 맞다면 stall신호를 1로 내보내 control unit을 통해 stall 하도록 구현하였다.

(5) Data forwarding implementation

데이터 포워딩의 구현은 다음과 같은 방식으로 하였다.

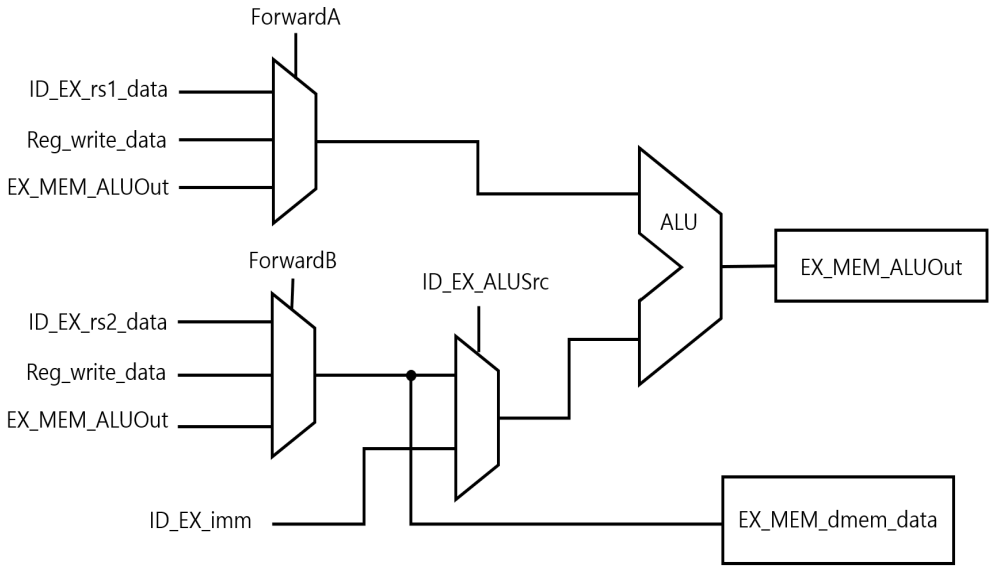

그림 1 Data Forwarding implementation

1) Forwarding signal implementation>

데이터 포워딩을 위해 데이터 포워딩 여부를 판단하는 Forwarding Unit을 구현하였다. ForwardingUnit 모듈의 output인 forwardA, forwardB가 mux의 출력을 결정함으로써 alu의 입력값을 결정한다. ForwardingUnit의 입력은 ID_EX 파이프라인 레지스터 ID_EX_rs1, ID_EX_rs2, EX_MEM 파이프라인 레지스터의 EX_MEM_rd, EX_MEM_reg_write, MEM_WB_rd, MEM_WB_reg_write를 가진다. 먼저 rs1의 alu input으로 데이터 포워딩 값을 사용할 지 결정하기 위해 rs1이 0이 아니고 rs1이 EX_MEM_rd와 같고 EX_MEM_reg_write가 1이라면 forwardA를 2'b 10으로 주어 mux에서 EX_MEM_alu_out이 출력되어 alu의 첫번째 인풋으로 사용되도록 구현하였다. 위 상황이 아닐때 만약 rs1이 0이 아니고 rs1이 MEM_WB_rd 와 같고 MEM_WB_reg_write값이 1이라면 forwardA를 2'b 01로 주어 mux의 출력이 WB stage에서 destination register에 Write back 하는 값인 reg_write_data로 되게 하여 alu의 첫번째 인풋으로 사용되게 한다. 만약 이 두가지 상황 모두 충족되지 않은 상황에는 forwardA를 2'b 00 으로 주어 mux의 출력이 ID_EX_rs1_data 가 되고 이 값이 alu 의 첫번째 인풋으로 사용되도록 구현하였다.

위와 같은 방식으로 rs2에 대해서 ForwardB 값으로 alu_in_2에 어떤 값이 사용될지 결정하도록 구현하였다.

2) 사용된 MUX 구현

a) ForwardA MUX

ForwardA MUX에서 ALU의 첫번째 입력을 결정하는 과정은 구체적으로는 다음과 같다. ForwardA 값이 0이면 ID_EX_rs1_data를 alu의 첫 번째 인풋으로 사용한다. ForwardA 값이 1이면 WB stage에서 destination register에 쓰고 있는 값인 reg_write_data를 alu의 첫번째 인풋으로 사용된다. ForwardA값이 2이면 MEM stage에서 사용되고 있는 값인 EX_MEM_alu_out이 alu의 첫 번째 인풋으로 사용된다.

b) ForwardB MUX

ForwardA MUX와 같은 방식으로 출력이 결정되도록 구현하였다.

c) ID_EX_alu_src MUX

ForwardB MUX의 결과는 바로 ALU의 두번째 입력으로 사용되지 않고 ID_EX_alu_src MUX를 통해 출력된 값이 ALU의 두번째 입력으로 사용되도록 구현하였다.

ID_EX_alu_src 가 0일때 ForwardB MUX의 아웃풋 ALU_in_2가 alu의 두번째 인풋으로 들어가고 1인 경우에는 ID_EX_imm 가 alu의 두번째 인풋으로 들어가도록 구현하였다.

(6) Halt implementation

Halt는 ecall instruction이 수행될 때 (앞 모든 인스트럭션이 모두 실행이 끝난 상황에) 17번 레지스터 값이 10이면 HALT하도록 구현하였다. 실행중인 인스트럭션이 ecall 인스트럭션임을 확인하는 과정은 ID stage Control Unit에서 opcode를 통해 판단하고 ecall 인스트럭션이라면 결과값으로 is_ecall =1을 내보내도록 구현하였다. 그리고 is_ecall이 1이라면 EX stage에서 실행중인 인스트럭션의 destination register를 확인해 만약 destination register(ID_EX_rd)가 17번 레지스터 이고 reg_write(ID_EX_reg_write)가 1이라면 alu결과 값을 Forwarding 해와서 이 값을 10과 비교한다. Alu결과값을 가져오는 로직은 blocking assignment 를 사용한 asynchronous logic을 사용하였다. Alu결과 값이 10과 같다면 wire_ID_EX_HALT(clk synchronous하게 업데이트 하기 위해 잠시 저장) = 1로 두어 clk rising edge때 ID_EX_HALT<=wire_ID_EX_HALT로 write 될 수 있도록 구현하였다. 이 HALT signal은 매 사이클 마다 다음 파이프라인 레지스터로 이동하여 MEM_WB 파이프라인 까지 이동하면 is_halted를 1로 주어 프로그램을 종료 시키도록 구현하였다. 즉, 미리 ID stage에서 Ecall 인스트럭션의 결과로 프로그램이 종료되는 것을 미리 알고 있더라도 ecall instruction의 앞 인스트럭션들이 모두 retired 되고 나서 프로그램이 종료 된다. 그리고 예외적으로 ecall 보다 먼저 실행되고 있는 EX stage, MEM stage의 인스트럭션이 load 인스트럭션이고 rd가 17인 경우에는 ControlUnit에서 stall signal을 1로 내보내도록 구현하였다. 따라서 ecall 바로 앞 두 인스트럭션이 17번 레지스터에 값을 불러오는 load instruction인 경우에만 instruction이 stall되고 나머지 경우에는 data forwarding을 통해 halt를 판단한다. 만약 ecall이 ID stage에서 실행중인데 EX stage에서 실행중인 인스트럭션의 rd(ID_EX_rd)가 17이고 reg_write가 1인데 mem_read가 0이고 alu결과가 10이 아닌 상황에서는 MEM stage에서 실행 중인 인스트럭션의 destination register가 17이고 reg_write가 1인지 확인하여 EX_MEM_alu_out 값을 포워딩 해서 10과 비교한다. 만약 EX_MEM_alu_out이 10이라면 마찬가지로 wire_ID_EX_HALT = 1로 두어 clk rising edge때 ID_EX_HALT가 1로 write 될 수 있도록 구현하였다. 또한 EX stage에서 17번 레지스터에 10을 write하지 않고, MEM stage에서 실행 중인 레지스터가 17번 레지스터에 10을 write하지 않는다면 지금 현재 레지스터 파일에서 17번 레지스터 값을 확인하여 만약 10이면 wire_ID_EX_HALT =1로 두어 마찬가지로 clk rising edge때 ID_EX_HALT가 1로 write되고 그 다음 Rising edge때 EX_MEM_HALT가 1로 write되고 그 다음 risng edge때 MEM_WB_HALT가 1이 되어 MEM_WB_HALT가 1임이 확인되면 is_halted를 1로 내보내 프로그램을 종료하도록 구현하였다.

기존 모듈들의 구현은 이전 single cycle cpu 에서의 구현과 같게 하였다.

1) **Memory**

Memory 모듈은 Instruction memory, data memory로 나누지 않고 하나의 memory를 resource reuse하도록 구현하였다. IF stage에서는 pc를 주소로 받아서 Instruction memory처럼 쓰이고 MEM stage에서는 ALU 결과 혹은 레지스터에 저장된 주소를 메모리 주소로 받아서 이뤄지도록 구현하였다. Reset, CLK, 메모리 주소인 addr, 저장할 데이터, 메모리에서 데이터를 읽을지 여부를 나타내는 MemRead, 메모리에서 데이터를 쓸지 여부를 나타내는 MemWrite를 입력으로 받도록 구현하였다. MemRead, MemWrite는 Control Unit 모듈에서 온 값이다. MemRead가 1일 때만 Memory값을 읽을 수 있고 MemWrite가 1일 때만 Memory에 저장할 데이터 값을 저장할 수 있도록 구현하였다.

메모리를 초기화하는 과정은 reset이 1인 경우에 메모리 값을 전부 0으로 초기화하고 주어진 경로에서 메모리를 가져오도록 구현하였다. 이 과정은 clock synchronous 하게 일어나도록 구현하였다. .

메모리 읽기 과정은 CLK asynchronous 하게 이뤄지도록 구현하였다. 이 모듈의 출력인 dout은 MemRead가 1이면 해당 addr에 저장된 데이터 값을 읽어서 dout으로 내보내고 MemRead가 0이면 0을 내보내도록 구현하였다.

메모리 쓰기 과정은 CLK synchronous 하게 이뤄지도록 구현하였다. MemWrite가 1일 때 저장할 데이터를 메모리에 write 한다.

2) **RegisterFile module**

RegisterFile 모듈은 레지스터 파일을 읽고 쓰는 모듈이다. Input으로 clk, rs1, rs2, rd, rd_din(input data for rd), write_enable을 받도록 구현하였다. 그리고 output은 rs1의 데이터를 읽은 값인 rs1_dout, rs2의 데이터를 읽은 값인 rs2_dout, 17번 레지스터 값을 읽은 값인 call을 가지도록 구현하였다. 레지스터파일에서 레지스터 값을 읽는 과정은 clock asynchronous 하게 이뤄지도록 구현하였다. 그리고 레지스터 파일에 write 하는 과정은 clock이 rising edge일 때 write 하도록(clock synchronous 하게) 구현하였다.

3) **ALU Control Unit**

어떤 연산을 해야 할지 결정하는 모듈이다. instruction, microPC을 입력받아 어떤 연산을 해야 하는지 구분한 뒤 alu_op_out으로 내보낸다. 먼저 opcode를 통해 어떤

타입의 인스트럭션인지 구분하고 그 뒤 funct3를 통해 어떤 instruction인지 세부적으로 구분한다. 예를 들어 opcode를 통해 B-type임을 판단 한 뒤에 funct3를 이용하여 BEQ, BNE, BLT 등을 구분하는 방식으로 구현하였다. ADD, SUB의 구분은 funct7을 통해 구분하도록 구현하였다. 여기서 출력한 alu_op_out은 ALU 모듈로 input으로 들어가게 구현하였다. 모든 과정은 CLK asynchronous 하게 이루어지게 구현하였다. 또한 ALU가 EX state 가 아닌 다른 state에서도 재사용 되므로 그 state에 적합한 alu_op_out을 가지도록 구현하였다. 예를 들면 Branch type 인스트럭션이 branch condition이 taken 됐을 때 WB stage에서는 ALU_ADD를 통해 PC+immediate value를 계산해야 하므로 instruction의 opcode를 통해 얻은 alu_op_out이 아닌 state를 통해 얻은 ALU_ADD를 alu_op_out으로 내보내서 ADD 연산을 이뤄질 수 있도록 구현하였다.

4) ALU

ALUControlUnit에서 어떤 연산을 해야 하는지 구분해 alu_op_in을 입력으로 받고 이를 통해 무슨 연산을 해야 하는지 판단 한 후 입력받은 두 입력값 input 1, input 2로 연산을 수행한다. 만약 Arithmetic 연산을 수행해야 한다면 input 1, input 2로 연산을 수행하고 연산 결과를 output reg [31:0] result_out으로 내보내고 b_con(branch condition이 True인지 False인지)을 0으로 내보낸다. 만약 연산이 branch condition을 확인하기 위한 연산(BEQ, BNE 등 을을 위한 연산) 이라면 branch conditon이 True일 때 b_cond를 1로 내보내고 아니면 b_cond를 0으로 내보내도록 구현하였다. 모든 과정은 CLK asynchronous 하게 이루어지게 구현하였다.

5) Immediate Generator

인스트럭션을 통해 immediate value를 계산하는 모듈이다. 인스트럭션을 input으로 받고 이 인스트럭션을 통해 계산된 immediate value인 imm_gen_out을 output으로 가진다. 먼저 Opcode 부분에 해당하는 part_of_inst[6:0]로 타입을 구분한다. 그리고 각 타입에 맞는 immediate value 위치를 인스트럭션에서 찾아 immediate value를 계산한 뒤 sign extension 하여 imm_gen_out으로 내보낸다. 모든 과정은 CLK asynchronous 하게 이루어지게 구현하였다.

6) PC

Program Counter 모듈이다. next_pc, reset, CLK를 input으로 가진다. CLK가 rising

edge일 때 reset이 1이면 current_pc를 0으로 초기화하고 reset이 0이 아니라면 input으로 받은 계산된 next PC value를 current_pc를 다음 pc값으로 업데이트해 준다. 이 과정은 clock synchronous 하게 이뤄지도록 구현하였다.

4. Discussion

(1) 우리가 구현한 5-stage pipelined cpu와 Ripe의 5-stage processor와의 비교.

Ripe의 5-stage processor는 non_control_flow_mem 테스트 벤치를 실행하는데 총 62cycle이 소요되지만 우리가 구현한 5-stage pipelined cpu에서는 총 56cycle이 소요된다. Ripes 에서는 ecall instruction이 EX stage에 오면 앞선 인스트럭션들(MEM, WB stage에 위치한)이 모두 Write Back 할 때까지 기다린다. 따라서 ecall 한 번 당 2 stall이 발생 한다. 우리 pipeline cpu 구현에서는 ecall instruction이 수행 될 때 ecall 바로 전 두 인스트럭션에서 17번 레지스터에 write한다면 17번 레지스터에 저장 될 값을 데이터 포워딩을 이용해서 가져오고 load해서 write하는 경우에만 stall을 만들기 때문에 본 테스트 케이스에서는 stall이 발생하지 않았다. 따라서 테스트 케이스에서 ecall instruction이 3번 호출되므로 Ripe의 5-stage processor와 우리 파이프라인 시피유는 6 사이클 차이가 나게 된다.

(2) Single cycle cpu와 pipelined cpu의 비교.

Single cycle cpu는 기본적으로 1cycle에 1instruction이 수행되므로 전체 실행되는 인스트럭션의 개수(n=51)가 total cycle이 된다. 5-stage Pipelined cpu는 stall이 발생하지 않는다면 전체 사이클 수는 5+(n-1) = 55 이 되므로 기본적으로 5-stage pipelined cpu는 single cycle cpu보다 많은 사이클 수를 필요로 한다. data forwarding을 사용한 경우에도 load instruction의 결과를 바로 사용하려고 할 때 data hazard가 발생하고 이로 인해 stall 하게 되므로 1cycle이 더 추가된다. 따라서 total cycle의 개수는 single cycle cpu가 pipelined cpu보다 작다. 하지만 pipelined cpu의 cycle 주기는 가장 긴 stage의 소요시간이고 single cycle cpu의 cycle의 주기는 가장 긴 Instruction의 소요시간이다. 따라서 pipelined cpu의 clk 주기가 single cycle cpu의 clk 주기보다 훨씬 작다. 따라서 pipelined cpu를 사용하면 throughput을 크게 증가시킬 수 있다.

5. Conclusion

이번 Lab에서는 Verilog를 이용해 Pipelined CPU를 구현하였다. 프로젝트를 진행하면서 Pipeline register를 clock synchronous하게 write하는 것의 중요함을 알게 되었다. Pipeline register를 만약 clock asynchronous하게 write한다면 데이터 무결성을 보장 할 수 없을 것이다. 파이프 라인 레지

스터들은 여러 stage에서 동시에 접근할 수 있으므로, 동시에 여러 군데에서 write할 가능성이 있다. 따라서 pipeline register들을 반드시 clk synchronous하게 write해야 된다.

5-stage Pipelined CPU는 Singe cycle CPU 보다 사이클 수는 많아지지만 throughput을 더 높일 수 있음을 알게 되었다.