Mobile Processor

Simple Pipelined MIPS

Project #3

32171550 박다은

2020 Spring

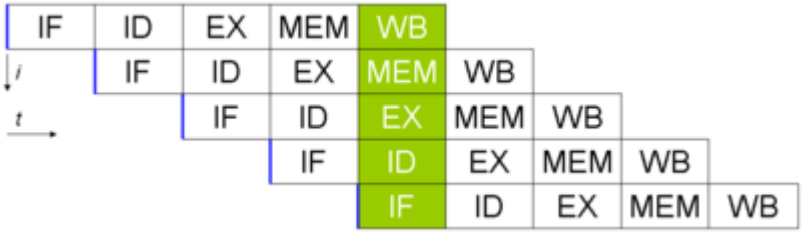
Freedays left: 0

Contents

1.Project Overview

(1) Project Introduction

Single-cycle 에서 향상된 pipeline MIPS emulator 을 만들어볼 것이다. 프로세서가 명령어 수행을 효율적으로 하기 위한 매커니즘인 pipeline 은 single-cycle 과 같이 IF, ID, EX, MEM, WB 의 다섯 단계로 이루어진다. Single-cycle 에서는 한 cycle 에서 한 명령어의 한 단계가 실행된다. 즉, 이론 적으로는 다섯개의 cycle 에 한 instruction 이 실행되는 것이다.



반면에 pipeline 에서는 위와 같이 각 단계에 instruction 이 하나씩 배치되어 실행되므로, 모든 단계가 쉬지 않아 전체 명령어가 처리되는 시간이 절약된다. 최대 다섯개의 명령을 동시에 실행시킬 수 있다. 이를 보면, 이론적으로 single-cycle 과 pipeline 의 cycle 수는 1/5 배 차이가 나야 한다. 하지만 pipeline 에서 발생하는 hazard 로 인해 실제로는 single-cycle cycle 수의 1/5 배 이상으로 나타난다.

(2) Project Goals

CPU 의 핵심기술 중 하나인 pipeline 을 직접 구현해보며, 그 개념을 이해한다. 또한 pipeline 의 문제점인 hazard 와 적합한 해결법을 알아보며 함께 구현한다. Cycle 의 수를 최대한 줄일 수 있는 방법을 고안하며 시도한다.

(3) Concepts in MIPS simulator

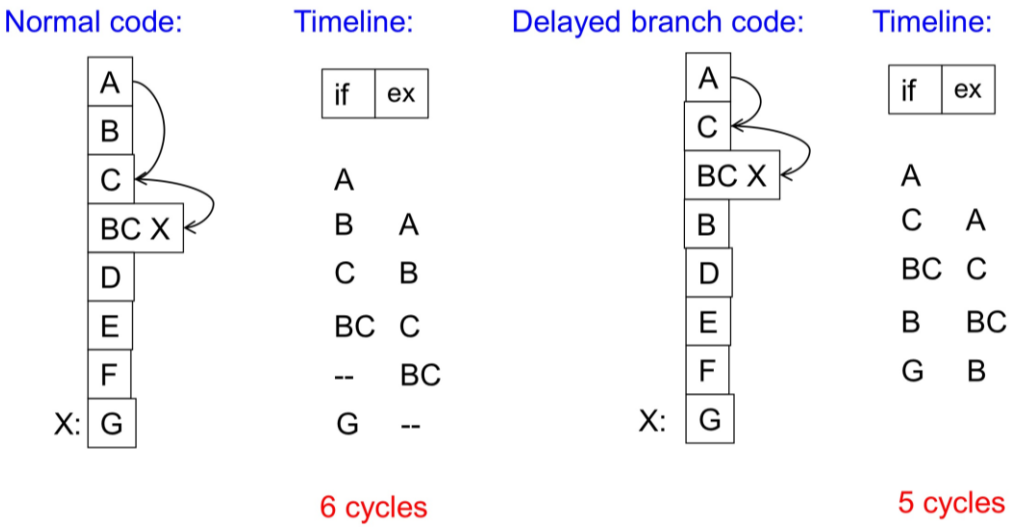
- ### Instruction Pipeline Hazard

- 해저드(Hazard)란 파이프라인에서 명령어 실행이 불가하여 지연, 중지가 발생해 속도가 저하되는 현상을 말한다. 크게 data dependency 와 control dependency 를 예로 들 수 있다.

- Data dependency: 이전 명령어에서 레지스터나 메모리의 값을 바꾸기 전에 명령어가 그 값을 읽거나 쓸 때 발생한다. 즉, 명령어의 값이 현재 파이프라인에서 수행중인 이전 명령의 값에 종속된 것이다. 해결방법은 stalling, forwarding, scoreboarding 등이 존재한다.

- Stalling: 명령어가 파이프라인단계를 진행하지 않고 레지스터나 메모리의 값이 바뀔 때까지 멈추는 것이다. 이는 결국 CPI = 1 을 포기하게 된다.
- Forwarding: 필요한 데이터가 레지스터 또는 메모리에 쓰여질 때까지 기다리지 않고, 사용하는 방법이다. 메모리나 레지스터에 데이터가 쓰여지는 단계는 각각 MEM 과 WB 이지만, 결과를 만들어 내는 것은 ALU 가 존재한 EXE 단계이다. 따라서 EXE 에서 forwarding 을 해주면 문제가 해결된다.
- Scoreboarding: 명령어 bit 이외로 새로운 1bit 를 추가해 레지스터 접근을 scoreboard 하는 방법이다. 가능하다면 1, 불가능하다면 0 으로 값을 매겨 체크한다.

- Control dependency: 일반적으로 PC 는 이전 PC 에 4 를 더한 값이지만, jump 나 branch 가 실행될 경우에는 PC 의 값이 명령어에 따라 달라진다. Jump, branch 명령어의 목표 PC 값은 각각 최소 decode, execute 단계에서 알 수 있다. 그 전까지는 목표 주소의 명령어가 아닌 다른 명령어가 파이프라인으로 들어오게 되어 발생하는 현상이다. 이를 해결하기 위해서는 stalling, branch delay slot, branch prediction 등의 방법이 필요하다.
  - Stalling: data dependency 의 stalling 과 비슷하게 다음 PC 의 값을 알 때까지 멈추는 것이다.
  - Branch delay slot: branch 의 실행을 지연시키는 방법이다. Branch 를 만나면 branch 와 전혀 관련성이 없는 명령어를 branch 명령어 다음에 위치하도록 변경하여 작업을 실행시키는 일이다.



  - Branch prediction: branch 를 실행하기 전에 목표 주소를 예측하는 방법이다. 예측이 맞을 경우, 파이프라인이 계속 실행되며, 만약 예측이 틀렸다면 실행됐던 명령어를 kill 하고 올바른 명령으로 다시 실행한다.

(4) Program Structure

Open_file

+----------------------------

| load_memory()

+----------------------------

| fetch()

| decode()

| execute()

| memory()

| write_back()

| Update Pipeline Latch

+----------------------------

각 단계에서 각자의 역할은 결과를 출력하는 것이다.

전체적으로 single-cycle 의 structure 와 유사하다. 먼저 메모리에 명령어들이 로드되는 것을 보여주고, 사이클 당 다섯단계의 결과를 출력한다. 마지막으로 각 단계의 임시저장소인 latch 변수를 업데이트한다.

(5) Build Environment

Compilation: Linux Assam, with GCC

To compile, please type:

gcc ./invalidation.c -o pipeline_32171550 or

gcc ./branch_prediction.c -o pipeline_32171550

To run, please type:

./pipeline_32171550 input.bin (input: simple, simple2, fib, gcd …)

2.Code & Consideration

Control dependency 를 해결하는 방법으로 'invalidation'과 'branch prediction (Last-Time Predictor)' 두가지를 사용해 총 두개의 코드를 작성했다. 전체적으로 모든 코드가 유사하나, branch 가 실행되는 execute 단계 와 branch 에 따라 PC 가 달라지는 fetch 부분은 두 방법에서 다르게 나타난다. 따라서 보고서에는 execute, fetch 단계만 구분해서 작성하고, 나머지 부분들은 구분없이 작성하도록 하겠다.

(1) 주요 코드

① Header File (INST.h)

A. 구조체

con_signal	RegDst, Jump, Branch, MemRead 등 control signal 에 해당하는 변수들이 들어있는 구조체이다.
instruction	opcode, rs, rt, rd 등 instruction 분해 후 나오는 변수들이 들어있는 구조체이다.
IFL	fetch 단계에서의 임시 저장소 역할을 하는 구조체이다.
IDL	decode 단계에서의 임시 저장소 역할을 하는 구조체이다.
EXL	execute 단계에서의 임시 저장소 역할을 하는 구조체이다.
MML	memory 단계에서의 임시 저장소 역할을 하는 구조체이다.
WBL	write_back 단계에서의 임시 저장소 역할을 하는 구조체이다.

B. 변수

공통	
Valid	각 단계가 실행이 될 지 정하는 변수이다.
pc4	PC+4 를 뜻하는(jump 나 branch 가 이루어지지 않음) 변수이다.
ir	현재 단계에서의 Instruction 을 뜻한다.
if_latch	
Inst	메모리에서 가져온 instruction 이다.
Id_latch, ex_latch, mm_latch, wb_latch	
WReg	연산 후 값이 쓰일 Register 의 index 이다.
C	구조체 con_signal 이 변수로 사용된다.
I	구조체 instruction 이 변수로 사용된다.
Id_latch, ex_latch	
v1, v2	ALU 에 들어갈 v1 과 v2 이다.
Npc	next PC 의 값으로, jump 와 branch 할 때만 사용된다.

ex_latch, mm_latch	
ALU_result	ALU 를 통해 연산이 된 결과값이다.
ex_latch	
Bcond	branch 의 taken, not taken 을 구별하는 변수이다.
mm_latch	
mm_output	다른 단계들에서의 forwarding 을 위해 사용되는 변수로, memory 단계의 아웃풋이다.
wb_latch	
wb_v	결과값이 저장된 레지스터의 값이다.

② Global

A. micro-architectural state

```
16 ///micro-architectural state
17 int Memory[0x1000000/4];
18 int PC;
19 int Regs[32];
```

Memory	명령어들이 저장 되어있는 장소이다.
PC	다음에 실행할 명령어의 주소를 기억하고 있는 배열이다.
Regs	레지스터

B. count

```
21 //count
22 int cycle = 0;
23 int inst_count = 0;
24 int m_count = 0;
25 int r_count = 0;
26 int b_count = 0;
27 int nb_count = 0;
28 int j_count = 0;
```

cycle	사이클의 개수를 세는 변수이다.
inst_count	사용된 명령어의 총 개수를 세는 변수이다.
m_count	메모리 명령어의 개수를 세는 변수이다.
r_count	레지스터 연산 명령어의 개수를 세는 변수이다.
b_count	branch 명령어의 개수를 세는 변수이다.
nb_count	taken 되지 않은 branch 명령어의 개수를 세는 변수이다.
j_count	jump 명령어의 개수를 세는 변수이다.

C. Control Signal in Decode Stage

```
30 //control signal in Decode stage
31 struct con_signal d_c;
```

d_c	decode 단계에서 사용되는 control signal 이다.

D. Five stage latches

```
33 //five stages latches
34 struct IFL if_latch[2];
35 struct IDL id_latch[2];
36 struct EXL ex_latch[2];
37 struct MML mm_latch[2];
38 struct WBL wb_latch[2];
```

- latch[0]은 현재 사이클 아웃풋의 임시저장소이고, latch[1]은 이전 사이클의 값의 임시저장소이다.

if_latch	IFL(fetch latch)의 임시저장소이다.
id_latch	IDL(decode latch)의 임시저장소이다.
ex_latch	EXL(execute latch)의 임시저장소이다.
mm_latch	MML(memory latch)의 임시저장소이다.
wb_latch	WBL(write back latch)의 임시저장소이다.

E. 기본 함수들

```
40 //stages 시작 전
41 void load_memory(FILE* fp);//file 읽고 메모리 load
42 void machine_initialization();//레지스터 값 초기화
43
44 void control(int op, int fun, int c_rs, int c_rt);//control type별 구분
```

load_memory	input file 을 읽고, 메모리에 저장하는 함수이다.
machine_intialization	R[31]과 R[29]를 초기화하는 함수이다.
Control	명령어 별 control signal 을 구분하는 함수이다.

F. Five stages (pipeline stages) 함수들

- Single cycle 과 마찬가지로 pipeline 도 F>D>E>M>W 의 다섯 단계로 이루어져 있다.

```
46 //five stages(pipeline stages)
47 int fetch(struct IFL *if_out, struct IFL *if_in, struct IDL *id_in, struct EXL *e
   x_in);
48 int decode(struct IFL *if_in, struct IDL *id_out, struct MML *mm_in);
49 int execute(struct IDL *id_in, struct EXL *ex_out, struct EXL *ex_in, struct MML
   *mm_in, struct WBL *wb_in);
50 int memory(struct EXL *ex_in, struct MML *mm_out, struct MML *mm_in);
51 int write_back(struct MML *mm_in, struct WBL *wb_out);
```

Fetch	명령어와 PC 의 값을 결정하는 단계이다.
Decode	명령어를 분해하는 단계이다.
execute	명령어에 맞는 연산을 하는 단계이다.

Memory	메모리에 저장을 하거나 로드하는 단계이다.
write_back	마지막으로 연산한 값을 레지스터에 저장하는 단계이다.

G. Stages 가 끝난 후 실행되는 함수들

```
53 //cycle 맨 마지막에서 print changed state, count instructions, update pipeline
54 void state_and_update_cnt(struct IFL if_latch[2], struct IDL id_latch[2], struct
   EXL ex_latch[2], struct MML mm_latch[2], struct WBL wb_latch[2]);
55
56 //모든 cycle이 종료되고 count한 값들 출력
57 void print_stastics();
```

stages_and_updage_cnt	사이클이 끝난 후, 바뀐 architectural state 를 출력하고, pipeline latch 를 업데이트하는 함수이다. 각각의 instruction 의 type 별 개수도 이 함수에서 센다.
print_stastics	모든 사이클이 종료된 후 프로그램이 끝나기 전, count 한 값들을 출력하는 함수이다.

H. Branch Prediction

- Branch prediction 에서만 쓰이는 구조체이다.

```
17 struct predict{
18         int target[10];
19         int BTB_PC[10];
20         int BTB_index;
21         int index;
22         int valid;
23 }p;
```

Target	Branch 의 목표주소가 적히는 배열이다.
BTB_PC	Branch 의 주소가 적히는 배열이다.
BTB_index	BTB 내에서 index 역할을 한다.
Index	새로운 branch 명령어가 생기면 새로운 배열에 저장할 수 있게 배열의 index 를 증가시켜주는 역할을 한다.
Valid	BTB 에 해당하는 명령어가 들어있는지 구분해주는 변수이다.

③ Written Function

A. load_memory

- 이 전의 single-cycle 에서 명령어들을 메모리에 로드할 때 사용된 reverse()함수를 이번 pipeline 에서는 이름을 바꿔 load_memory() 함수로 사용했다. 먼저 파일을 이진모드로 받아 뒤집힌 명령어들을 원하는 명령어로 바꿔주고, 그 명령어를 메모리에 저장하도록 했다.

B. state_and_update_cnt

- 한 사이클이 끝나기 직전에 실행되는 함수이다.

```
496 //cycle 맨 마지막에서 print changed state, count instructions, update pipeline
497 void state_and_update_cnt(struct IFL if_latch[2], struct IDL id_latch[2], struct
    EXL ex_latch[2], struct MML mm_latch[2], struct WBL wb_latch[2]){
498         //prints out the changed state
499         //count instructions
500         printf("\n");
501
502         //changed PC
503         if(id_latch[0].c.Jump){
504                 printf("changed PC      : PC = 0x%x\n",
505                         id_latch[0].npc);
506                 j_count++;
507         }
508         if(ex_latch[0].c.Branch){
509                 if(ex_latch[0].bcond){
510                         printf("changed PC      : PC = 0x%x\n",
511                                 ex_latch[0].npc);
512                 }
513                 else nb_count++;
514                 b_count++;
515         }
516
517         //changed memory
518         if(mm_latch[0].c.MemWrite || mm_latch[0].c.MemRead){
519                 if(mm_latch[0].c.MemWrite)
520                         printf("changed memory  : M[0x%x] = 0x%x\n",
521                                 mm_latch[0].ALU_result,Regs[mm_latch[0].i.rt]);
522                 m_count++;
523         }
524
525         //changed register
526         if(wb_latch[0].c.RegWrite){
527                 printf("changed register: R[%d] = 0x%x\n",
528                          wb_latch[0].WReg, wb_latch[0].wb_v);
529         }
530
531         r_count = inst_count - m_count - b_count - j_count;
```

- 먼저 PC, register, memory 의 값이 바뀌었다면 바뀐 값이 출력하도록 했다. 각각에 적합한 control signal 을 사용해서 바뀌었는지 여부를 확인하도록 했다.

- PC 가 바뀌었다는 것은 jump 혹은 taken branch 가 실행되었는지를 기준으로 하였다. Single-cycle 에서는 instruction 하나씩 기준으로 PC 값이 바뀌었는지 확인하면 됐는데, 이번에는 다섯 단계의 PC 가 모두 다르므로 기준이 모호했기 때문이다.

- 점프는 decode 단계에서, branch 는 execute 단계에서 행해지도록 설정하였기 때문에, 각각 decode 의 아웃풋과 execute 아웃풋을 if 문에 들어가는 변수로 사용했다. 마찬가지로 프로그램의 memory 는 memory 단계에서, register 은 write_back 단계에서 바뀌므로 memory 의 아웃풋과 write_back 의 아웃풋을 변수로 사용했다.

- 각각의 type 별 instruction 의 개수를 세는 것도 이 함수에서 하도록 해주었다. 파이프라인 단계에서 해도 되지만, 따로 함수를 만들어서 정리하는 것이 보기 편할 것 같았다. 마침, 바뀐 architectural state 를 출력하도록 만든 이 함수에 count 를 셀 때

필요한 조건이 모두 있어서 따로 함수를 만들지 않고, 이 함수에 합쳐주었다. 예외로, 총 instruction 의 개수를 세는 것만 fetch 단계에서 행해지도록 하였다.

- r_count(# of register operation instruction)를 레지스터에 값이 쓰이는 instruction 의 개수인지(R-type, I-type, LW), 아니면 레지스터 연산이 이루어진 instruction 의 개수인지(R-type, I-type)에 대해 고민이 있었다. 전자면 RegWrite 가 1 인지 여부를 통해, 즉 이 함수의 마지막 if 문에서 개수를 세면 되지만 후자면 조금 까다로웠다. R-type 과 I-type 두개만 겹치는 control signal 이 없기 때문이다. 고민 끝에 후자로 결정하였고, 총 명령어의 개수에서 memory instruction, jump instruction, branch instruction 을 빼는 방법으로 r_count 의 개수를 셌다.

```
533         //update
534         if_latch[1] = if_latch[0];
535         id_latch[1] = id_latch[0];
536         ex_latch[1] = ex_latch[0];
537         mm_latch[1] = mm_latch[0];
538         wb_latch[1] = wb_latch[0];
539 }
```

- 사이클이 끝난 후 latch 를 업데이트 하는 과정도 이 함수에 넣었다. latch[0]은 현재 사이클의 아웃풋을 저장한 임시공간이고, lathc[1]이 다음 사이클의 인풋으로 사용될 것이므로 한 사이클의 끝에서 이를 업데이트 해줘야 한다.

④ Pipeline Stages

A. fetch

➢ Delay Branch Slot

```
152         //decode단 계 에 서  Jump가  이 루 어 졌 을  때
153         if(id_in->valid == 1 && id_in->npc){
154                 //jump할  address로  PC값 을  바 꾼 다
155                 PC = id_in->npc;
156                 if(PC == 0xffffffff){
157                         if_out->pc4 = 0xffffffff;
158                         if_out->ir = 0x0;
159                         return 0;
160                 }
161         }
162         //execute단 계 에 서  Branch가  taken됐 을  때
163         else if(ex_in->valid == 1 && ex_in->npc){
164                 //jump할  address로  PC값 을  바 꾼 다
165                 PC = ex_in->npc;
166                 if(PC == 0xffffffff){
167                         if_out->pc4 = 0xffffffff;
168                         if_out->ir = 0x0;
169                         return 0;
170                 }
171         }else PC = if_in->pc4;//둘  다  아 니 면  PC = 이 전 PC+4
```

- jump 나 taken branch 가 행해지지 않은 명령어의 PC 값은 이전 PC 값의 4 를 더한 값이다. 하지만 이전 명령어가 jump 나 taken branch 라면, 목표 주소값으로 PC 값을 바꿔주어야

한다. jump 는 decode 단계, branch 는 execute 단계에서 실행되게 했으므로 이에 맞게 if 문을 나눠주었다.

- PC 의 값이 0xffffffff 이면 프로그램을 종료해야 한다. Single-cycle 에서는 PC 가 0xffffffff 가 되면, 바로 종료하면 되지만, pipeline 에서는 그러지 못한다. 아직 다른 명령어들이 실행 중이기 때문이다. 따라서 write_back 단계의 명령어의 주소가 0xffffffff 이면 종료하도록 하였다.

➢ Branch prediction

```
175         //branch 검색
176         for(int i=0; i<10; i++){
177                 if(PC == p.BTB_PC[i] && PC != 0){
178                         if_out->npc = p.target[i];
179                         if(p.BTB_PC[i] == p.target[i]){
180                                 if_out->npc = PC + 4;
181                         }
182                         break;
183                 }
184                 else{
185                         if_out->npc = PC+4;
186                 }
187         }
188         PC = if_out->npc;
```

- PC 가 BTB 에 저장돼 있는지 확인하고, 저장되어 있다면 PC 를 저장했던 타겟 주소로 변경하도록 했다. 만약 BTB 의 PC 값과 타겟 주소의 값이 같다면 not-taken branch 라 예측하는 것이기 때문에 PC 의 값을 변경하지 않고 이전 PC 의 4 를 더하도록 했다.

B. decode

- 명령어를 각각에 맞게 분해하는 과정이다. decode 과정에서 사용되는 instruction 으로 구조체 instruction 변수인 D_inst 를 설정하였다.

- 분해하는 방법은 single-cycle 에서 사용했던 방법과 동일하다.

```
219         //control type별 구분
220         control(D_inst.opcode, D_inst.func, D_inst.rs, D_inst.rt);
221         id_out->c = d_c;
222
223         //execute의 ALU에서 사용할 v1,v2 정하기
224         //v1은 모두 rs
225         id_out->v1 = Regs[id_out->i.rs];
```

```
227         if(id_out->c.ALUSrc) //ALUSrc = 1이 면 (I-type) v2 = simm
228                 id_out->v2 = id_out->i.simm;
229         else //ALUSrc = 0이 면 (R-type) v2 = rt
230                 id_out->v2 = Regs[id_out->i.rt];
231
232         //zimm, imm, shamt 사 용 하 는  것 들  예 외 처 리
233         if(id_out->i.opcode == 0xc || id_out->i.opcode == 0xd)
234                 id_out->v2 = id_out->i.zimm;
235         if(id_out->i.opcode == 0xf)
236                 id_out->v2 = id_out->i.imm;
237         if(id_out->i.opcode == 0 && (id_out->i.func == 0x0 || id_out->i.func == 0
    x2))
238                 id_out->v1 = id_out->i.shamt;
239
240         //WB단 계 에 서  Write할  Register 정 하 기
241         if(id_out->c.RegDst) //RegDst = 1이 면 (R-type) WReg = rd
242                 id_out->WReg = id_out->i.rd;
243         else //RegDst = 0이 면 (I-type) WReg = rt
244                 id_out->WReg = id_out->i.rt;
```

- control 함수를 통해 명령어의 control signal 을 설정하였다.
- decode 단계에서 execute 의 ALU 에 들어 갈 v1 과 v2, write_back 에서 쓰일 WReg 의 값을 구하도록 했다. Single-cycle 에서는 execute 에서 v1, v2 를 구하고 write_back 에서 WReg 을 구했지만, 이번 pipeline 에서는 jump 를 decode 에 넣었으므로 decode 에서 이들을 미리 구하는 것으로 하였다.
- Single-cycle 과 마찬가지로 v1 은 항상 R[rs]이고, v2 는 ALUSrc 에 따라 달라진다. I-type 일 때 v2 = simm 이고, R-type 일 때 v2 = R[rt]이다. andi 와 ori 에서 simm 대신 zimm 이 쓰이고, lui 에서는 simm 대신 imm 이 쓰인다. 또 sll 과 srl 에서 변수는 R[rt]와 shamt 이므로 각각에 맞게 예외처리도 해주었다.
- 뿐만 아니라, R-type 일 때 결과 레지스터는 R[rd]이고, I-type 일 때는 R[rt]이므로 그에 맞게 설정해주었다.

```
246         //PC JUMP-----------------------------------------------------
247         int ToJump = Regs[id_out->i.rs];//R[rs]로  점 프 (jr,jalr에 서 만 )
248
249         //forwarding
250         if(mm_in->c.RegWrite) //R-type, I-type, LW
251                 //rs와  이 전  cycle mm단 계  WReg가  같 을 때
252                 if(id_out->i.rs == mm_in->WReg)
253                         ToJump = mm_in->Address;
254
255         if(id_out->c.Jump){
256                 if(id_out->i.opcode == 0 && id_out->i.func == 0x8) //jr
257                         id_out->npc = ToJump;
258                 else if(id_out->i.opcode == 0 && id_out->i.func == 0x9){ //jalr
259                         Regs[id_out->i.rd] = id_out->pc4+4;
260                         id_out->npc = ToJump;
261                 }
262                 else if(id_out->i.opcode == 0x2) //j
263                         id_out->npc = id_out->i.J_Addr;
264                 else if(id_out->i.opcode == 0x3){ //jal
265                         Regs[31] = id_out->pc4+4;
266                         id_out->npc = id_out->i.J_Addr;
267                 }
268                 printf("\t\t\tJump! ~> 0x%x\n",id_out->npc);
269         }
270         else
271                 id_out->npc = 0;
```

- rs 와 이전 cycle MEM 단계의 WReg 가 같을 때 data dependency 가 존재한다. 따라서 forwarding 해주는 과정이 필요하다.
- Decode 를 해 opcode 를 알아냄으로써 jump 명령어를 구분할 수 있다. 따라서 decode 에 jump 하는 과정을 넣었다. 최대한 빨리 점프를 해야 낭비가 줄기 때문이다. j 와 jal 은 명령어의 Address 로 점프하므로 상관이 없지만, jr 과 jalr 은 레지스터 값으로 점프하므로 data dependency 가 존재한다.

C. execute

```
287         //forwarding
288         if(ex_in->c.RegWrite){ //R-type, I-type, LW
289                 if(id_in->i.rs == ex_in->WReg)
290                         v1 = ex_in->ALU_result;
291                 if(id_in->i.rt == ex_in->WReg)
292                         if(!(id_in->c.ALUSrc)) //I-type, LW
293                                 v2 = ex_in->ALU_result;
294         }
295         else if(mm_in->c.RegWrite){ //R-type, I-type, LW
296                 if(id_in->i.rs == mm_in->WReg)
297                         v1 = mm_in->mm_output;
298                 if(id_in->i.rt == mm_in->WReg)
299                         if(!(id_in->c.ALUSrc)) //I-type, LW
300                                 v2 = mm_in->mm_output;
301         }
302         else if(wb_in->c.RegWrite){ //R-type, I-type, LW
303                 if(id_in->i.rs == wb_in->WReg)
304                         v1 = wb_in->wb_v;
305                 if(id_in->i.rt == wb_in->WReg)
306                         if(!(id_in->c.ALUSrc)) //I-type, LW
307                                 v2 = wb_in->wb_v;
308         }
```

- Data dependency 를 해결하기 위해 forwarding 방법을 사용했다.
- execute 단계에서 연산이 수행되므로, forwarding 을 해줘야 한다. 사용되는 동시에 값이 업데이트 되는 레지스터가 있을 수 있기 때문이다. If-else 문의 순서를 execute 단계에서 가까운 순으로 해주었다. (ex->mm->wb)
- main 문에서 WB 단계를 제일 먼저 즉, WB>IF>ID>EX>MEM 순으로 작성해주게 되면 마지막 if 문인 wb 단계 forwarding 을 따로 해주지 않아도 된다. 하지만 전체적으로 코드 작성을 저번 single-cycle 에서 보완하는 식으로 작성해서 이번 simple pipeline 에서도 IF>ID>EX>MEM>WB 순으로 구성했다. 낭비가 생길 수도 있을 것 같아 WB 단계를 먼저 작성하는 방법으로 고치려 했으나, 관련 변수들과 전체적인 틀을 다 바꿔줘야 돼서 현재의 방법을 유지하게 되었다.
-

```
315         //forwarding에서 v1값 업데이트됨 (v1이 shamt라 업데이트되면 안됨)
316         if(id_in->i.opcode == 0 && (id_in->i.func == 0x0 || id_in->i.func == 0x2))
317                 v1 = id_in->i.shamt;
```

- sll과 srl은 다른 R-type 명령어들과 달리 R[rs]를 아예 사용하지 않는다. 일반적으로 R-type 명령어의 v1 은 R[rs]이고, v2 는 R[rt]이다. 반면에 sll 과 srl 은 R[rt]와 shamt 를 사용하므로 R[rt]는 그대로 v2 로 두고, shamt 를 v1 로 설정하도록 했다. Decode 단계에도 이 부분이 있지만, 위의 forwarding 과정 때문에 v1 이 다시 R[rs]로 업데이트가 될 수 있다. 따라서 이를 예방하기 위해, 다시 v1 을 shamt 로 설정하는 과정을 넣어주었다.

- ALU 는 Single-cycle 때와 마찬가지로 ALUop 를 통해 switch 문을 구분하게 했다. 또한 전과 마찬가지로 sw, lw 는 add 연산을 실행하도록 하였고, 점프 명령어는 아무 연산도 수행하지 않고 switch 문을 빠져나오도록 하였다.

➢ Invaliation

```
377         //Update PC(branch)
378         if(id_in->c.Branch){
379                 if(bcond){ //branch taken
380                         if_latch->valid = 0; //필요없는 값이 fetch되었으므로
381                         ex_out->npc = id_in->pc4 + id_in->i.Br_Addr;
382                         printf("\t\t\tJump! ~> 0x%x\n",ex_out->npc);
383                 }
384                 else{
385                         ex_out->npc = 0;
386                 }
387         }
388         else ex_out->npc = 0;
```

- 일단 branch 가 taken 인지 not-taken 인지 상관없이 다음 명령어는 계속 fetch 되도록 했다. Branch 명령어가 not-taken 이면 상관없지만, 만약 taken 이라면 fetch 된 명령어들이 이후 단계들을 실행하지 못하도록 없애야 한다. 매 단계마다 실행될지는 변수 valid 를 통해 구별하도록 했다. valid=1 이면 단계가 실행되고, valid=0 이라면 그 단계는 실행되지 않는 것이다. 그러므로 taken branch 라면, fetch 단계의 valid 의 값을 0 으로 바꾸도록 했다.

➢ Branch Prediction

```
400         if(id_in->c.Branch){
401                 if(bcond){
402                         ex_out->npc = id_in->pc4 + 4 + id_in->i.Br_Addr;
403                         printf("%x\n",ex_out->npc);
404                 }
405                 else
406                         ex_out->npc = id_in->pc4;
```

- bcond가 1일 때는(taken), Invalidation에서 valid를 뺀 것과 동일하게 PC의 값을 정했다.

- 하지만 bcond 가 0 이면, npc 를 0 으로 값을 바꿔주었던 invalidation 과 달리 branch prediction 에서는 npc 를 pc4 값으로 바꾸도록 했다. 위 방법에서는 npc 는 branch 가 taken 됐을 때만 사용되지만 이번 방법에서는 branch 명령어이기만 하면 npc 를 PC 의 값으로 바꾸도록 했기 때문이다.

```
398                 //branch 검색
399                 for(int i = 0 ; i<10 ; i++){
400                         if(p.BTB_PC[i] == id_in->pc4){
401                                 p.BTB_index = i;
402                                 p.valid = 1;
403                                 break;
404                         }
405                         else
406                                 p.valid = 0;
407                 }
```

- BTB 에 현재 PC 에 해당하는 값이 있는지 검색한다.

```
409                 if(p.valid == 1){
410                         if(p.target[p.BTB_index] != ex_out->npc){
411                                 PC = ex_out->npc;
412                                 memset(&if_latch[0], 0 ,sizeof(if_latch));
413                                 memset(&id_latch[0], 0 ,sizeof(id_latch));
414                                 p.target[p.BTB_index] = ex_out->npc;
415                         }
416                 }
417                 else{
418                         if(bcond)
419                                 if_latch->valid = 0;
420                         PC = ex_out->npc;
421                         p.target[p.index] = PC;
422                         p.BTB_PC[p.index] = id_in->pc4;
423                         //memset(&if_latch[0], 0 , sizeof(if_latch));
424                         p.index++;
425                 }
426         }
```

- BTB 에 해당하는 값이 있지만 target 과 현재 PC 의 값이 다르다면(잘못 예측했다면), PC 의 값을 현재 execute 단계의 PC 값으로 바꿔준다. 예측했던 시점으로 다시 돌아가 예측했던 것과 반대로(taken -> not taken, not taken -> taken) 실행시켜야 하기 때문이다. 현재 fetch 단계와 decode 단계에는 잘못된 명령어들이 들어와 있는 것이므로, 그 명령어들을 kill 하기 위하여 memset 이라는 함수를 사용하였다. 마지막으로 takget 의 값을 새로운 주소로 다시 바꿔주었다.

- BTB 에 일치하는 PC 가 없다면, 새로운 branch 명령어 이므로 BTB 의 target 과 PC 의 값을 넣어주는 과정을 거쳐야 한다. 한번 값이 적힌 BTB 의 index 에는 또다른 값이 적히면 안되므로, index 의 값을 1 만큼 증가시켜주었다.

D. memory

- LW 는 따로 hazard 가 존재하지 않으므로 single-cycle 과 동일하게 코드를 짰다.

```
422         //SW
423         else if(ex_in->c.MemWrite){ //메모리 저장하기
424                 v2 = Regs[ex_in->i.rt];
425
426                 //forwarding
427                 if(mm_in->c.RegWrite) //R-type, I-type, LW
428                         if(ex_in->i.rt == mm_in->WReg)//rs와 이전 cycle mm단계 WReg가
    같을 때
429                                 v2 = mm_in->mm_output;
430
431                 Memory[ex_in->ALU_result] = v2;
432                 printf("        Write Memory\n");
433                 printf("                M[0x%x] <= R[%d](0x%x)\n",
434                                 ex_in->ALU_result, ex_in->i.rt, v2);
435         }
```

- 반면에 SW 에서는 data dependency 가 존재한다.
- rt 와 이전 사이클 mm 단계의 Write Register 가 동일하면 hazard 가 존재한다. 따라서 forwarding 을 해서 문제를 해결해주었다.

```
437         //jal, jalr (decode단계 Jump에서 사용)
438         else if(ex_in->i.opcode == 3 || (ex_in->i.opcode == 0 && ex_in->i.func == 9))
439                 mm_out->mm_output = ex_in->pc4;
440         else //ex단계 forwarding에서 사용
441                 mm_out->mm_output = ex_in->ALU_result;
```

- 원래 memory 단계는 LW, SW 만 고려하면 끝이지만, memory 단계 때문에 생기는 forwarding 으로 인해 mm_output 이라는 변수를 만들어 forwarding 을 해결하도록 했다.

E. write_back

- single-cycle 은 PC 가 0xffffffff 이면 사이클이 종료되지만 pipeline 은 마지막 단계의 수행이 끝날 때까지 종료가 되면 안된다. 따라서 write_back 단계에서 현재 단계에서의 pc 가 0xffffffff 인지 확인하고 맞으면 종료하도록 했다.

```
466         int wb_v;
467         if(mm_in->c.RegWrite){ //R-type, I-type, LW
468                 if(mm_in->c.MemtoReg) //LW
469                         wb_v = mm_in->mm_output;
470                 else //R-type, I-type
471                         wb_v = mm_in->ALU_result;
472                 Regs[mm_in->WReg] = wb_v;
473                 printf("        R[%d] = 0x%x\n",
474                         mm_in->WReg, wb_v);
475         }
```

- RegWrite 가 1 인지 확인하고 그 안에서 또 LW 명령어와 R-type, I-type 명령어를 나눠 레지스터에 저장할 값을 정했다. decode 단계에서 이미 저장레지스터를 정했으므로 또 구할 필요 없이 값을 가져다 썼다.

(2) Screenshots

① 출력 화면

```
                    :
                    :
                    :

-------------cycle 295----------------
        R[29] = 0x100000

changed register: R[29] = 0x100000

-------------cycle 296----------------


-------------cycle 297----------------
        R[0] = 0x0

changed register: R[0] = 0x0

-------------cycle 298----------------

final return value(R[2])             : 55
# of cycles                          : 298
# of instuctions                     : 293
# of memory operation instructions   : 100
# of register operation instructions : 161
# of branch instructions             : 10
# of not-taken branches              : 1
# of jump instructions               : 22
```

- 출력은 이와 같이 cycle 당 바뀐 architectural state 을 출력하고, 전체 사이클이 끝나면 결과값과 count 한 값들을 출력하도록 했다.

② Cycle 비교

- Single cycle 의 cycle 수는 총 inst 수에 5 를 곱한 값으로 계산하였다.
- 대표로 simple.bin, simple4.bin, input4.bin 파일들에 대해서 single-cycle 과 invalidation, branch prediction 의 cycle 수를 비교해보겠다.

simple.bin

```
final return value(R[2])              : 0
number of cycles                      : 40
number of excuted instructions        : 8
number of R-type instructions         : 3
number of I-type instructions         : 4
number of J-type instructions         : 0
number of memory access instructions  : 2
number of taken branches              : 0
```

<single-cycle>

```
final return value(R[2])              : 0
# of cycles                           : 14
# of instuctions                      : 9
# of memory operation instructions    : 2
# of register operation instructions  : 6
# of branch instructions              : 0
# of not-taken branches               : 0
# of jump instructions                : 1
```

<Invalidation>

```
final return value(R[2])              : 0
# of cycles                           : 14
# of instuctions                      : 9
# of memory operation instructions    : 2
# of register operation instructions  : 6
# of branch instructions              : 0
# of not-taken branches               : 0
# of jump instructions                : 1
```

<Branch Prediction>

- single-cycle 과 pipeline 을 비교했을 때, cycle 수는 1/4 정도 차이가 난다.
- Branch 명령어가 없는 코드이기 때문에 invalidation 과 branch prediction 의 차이가 없다.

simple4.bin

```
final return value(R[2])              : 55
number of cycles                      : 1215
number of excuted instructions        : 243
number of R-type instructions         : 60
number of I-type instructions         : 153
number of J-type instructions         : 11
number of memory access instructions  : 100
number of taken branches              : 9
```

<single-cycle>

```
final return value(R[2])              : 55
# of cycles                           : 298
# of instuctions                      : 293
# of memory operation instructions    : 100
# of register operation instructions  : 161
# of branch instructions              : 10
# of not-taken branches               : 1
# of jump instructions                : 22
```

<Invalidation>

```
final return value(R[2])              : 55
# of cycles                           : 275
# of instuctions                      : 270
# of memory operation instructions    : 100
# of register operation instructions  : 138
# of branch instructions              : 10
# of not-taken branches               : 1
# of jump instructions                : 22
```

<Branch Prediction>

- single-cycle 과 pipeline 을 비교했을 때, cycle 수는 simple.bin 과 비슷하게 1/4 정도 차이가 난다.
- Branch 명령어가 존재하는 코드이므로, invalidation 과 branch prediction 의 총 instruction 수와 cycle 수에 차이가 생긴다. Invalidation 에서 cycle 수는 298 이고, branch prediction 의 cycle 수는 275 로 branch prediction 이 더 효율이 좋음을 확인할 수 있다. 둘 다 control dependency 를 제어하는 코드이지만, 보기와 같이 효율에서 차이를 보인다.

input4.bin

```
final return value(R[2])              : 85
number of cycles                      : 116863530
number of excuted instructions        : 23372706
number of R-type instructions         : 5076368
number of I-type instructions         : 13219741
number of J-type instructions         : 103
number of memory access instructions  : 7116606
number of taken branches              : 2028830
```

<single-cycle>

```
final return value(R[2])              : 85
# of cycles                           : 27430475
# of instuctions                      : 27430470
# of memory operation instructions    : 7116606
# of register operation instructions  : 18284061
# of branch instructions              : 2029699
# of not-taken branches               : 869
# of jump instructions                : 104
```

<Invalidation>

```
final return value(R[2])              : 85
# of cycles                           : 23376940
# of instuctions                      : 23376935
# of memory operation instructions    : 7116607
# of register operation instructions  : 14231347
# of branch instructions              : 2028877
# of not-taken branches               : 870
# of jump instructions                : 104
```

<Branch Prediction>

- 위의 input 파일들과 마찬가지로, single-cycle 과 pipeline 의 cycle 수는 1/4 정도의 차이가 난다.
- 코드가 복잡해지고, cycle 의 수가 늘어남으로써 pipeline 의 두 코드도 확연한 차이를 보인다. Invalidation 의 cycle 수는 27,430,475 이고, branch prediction 의 cycle 수는 23,376,940 으로 약 4,000,000 만큼 차이가 난다.

(3) Making Input File

저번 프로젝트에서 factorial 파일을 input 파일로 만들어보았기 때문에, 이번에는 1 부터 10 까지 더하는 'sum'과 500 원을 셀 수 있는 경우의 수를 구하는 'coin'을 만들어보았다.

```
int main(){
        int i = 0;
        int sum = 0;

        for(i=1; i<=10; i++){
                sum += i;
        }

        return sum;
}
```

- 먼저 sum.c 파일은 다음과 같다. 'mips-mti-linux-gnu-gcc -c sum.c'를 입력해 compile 을 시켜주었고, 'mips-mti-linux-gnu-gcc -c sum.c -o sum.o'를 입력해 sum.o 파일을 만들어주었다. 그리고, 'mips-mti-linux-gnu-objcopy -O binary -j .text sum.o sum.bin'을 입력해 sum.bin 파일을 만들어 주었다. 마지막으로, 'mips-mti-linux-gnu-objdump -d sum.o'를 입력해 어셈블리까지 만들어주었다.

```
00000000: 27bd ffe8 afbe 0014 03a0 f025 afc0 0008  '..........%....
00000010: afc0 000c 2402 0001 afc2 0008 1000 0008  ....$...........
00000020: 0000 0000 8fc3 000c 8fc2 0008 0062 1021  .............b.!
00000030: afc2 000c 8fc2 0008 2442 0001 afc2 0008  ........$B......
00000040: 8fc2 0008 2842 000b 1440 fff6 0000 0000  ....(B...@......
00000050: 8fc2 000c 03c0 e825 8fbe 0014 27bd 0018  .......%....'...
00000060: 03e0 0008 0000 0000 0000 0000 0000 0000  ................
00000070: 0a                                       .
```

- sum.bin 파일에서 ':%!xxd'를 입력하면 나오는 화면은 다음과 같다.

- sum 파일로 출력을 해보니, 이상한 값이 나오게 됐다. 그래서 binary 파일의 문제인 줄 알고 계속 살펴보았다. 그런데 한번, single-cycle 로 sum 파일을 돌려보니 원하는 값이 나왔다. 현재 pipeline 코드에 잘못이 있다는 뜻이었다. 모든 예제 input 파일이 잘 돌아가 구현을 완성한 줄 알았는데 아니었다. Single-cycle 의 출력과 비교해보니, forwarding 이 제대로 처리되지 않았다.

```
if(ex_in->c.RegWrite){ //R-type, I-type, LW
        if(id_in->i.rs == ex_in->WReg){
                v1 = ex_in->ALU_result;
                if(ex_in->i.opcode == 0x23)
                        v1 = Memory[ex_in->ALU_result];
        }
        if(id_in->i.rt == ex_in->WReg)
                if(!(id_in->c.ALUSrc)){ //I-type, LW
                        v2 = ex_in->ALU_result;
                        if(ex_in->i.opcode == 0x23)
                        v2 = Memory[ex_in->ALU_result];
                }
}
```

- execute 의 forwarding 부분을 다음과 같이 바꿔주었다. 명령어가 LW 일 때, execute 단계에서 더하는 과정만 거쳤기 때문에, forwarding 을 할 때 그 결과값을 Memory 에 넣어 값을 받아야 했다. 따라서 이와 같이 알맞게 고쳤더니 잘 돌아갔다.

```
final return value(R[2])             : 55
# of cycles                          : 156
# of instuctions                     : 151
# of memory operation instructions   : 73
# of register operation instructions : 64
# of branch instructions             : 13
# of not-taken branches              : 1
# of jump instructions               : 1
```

<Invalidation>

```
final return value(R[2])             : 55
# of cycles                          : 139
# of instuctions                     : 134
# of memory operation instructions   : 73
# of register operation instructions : 47
# of branch instructions             : 13
# of not-taken branches              : 1
# of jump instructions               : 1
```

<Branch Prediction>

- 출력은 다음과 같다.

```
int main(){
        int coin = 500;
        int cnt = 0;
        int i,j,k,l;
        for(i=0; i<= coin/500; i++){
                for(j=0; j<= coin/100; j++){
                        for(k=0; k<= coin/50; k++){
                                for(l=0; l<= coin/10; l++){
                                        if(500*i + 100*j + 50*k + 10*l == coi
n){
                                                cnt++;
                                        }
                                }
                        }
                }
        }
        return cnt;
}
```

- coin.c 파일은 다음과 같다. 이를 sum.c 와 같은 방법으로 binary 파일과 assembly 파일을 만들었다.

```
19c:   00430018        mult    v0,v1
1a0:   00001810        mfhi    v1
```

```
-------------cycle 16-----------------
        wrong insrtuction
```

- 이와 같이 따로 구현하지 않은 명령어가 있었기 때문에 'wrong instruction'이라는 문구가 뜨면서 출력이 종료됐다.

- 저번 프로젝트에서 시도를 했던 fact 도 'mul'명령어 때문에 옳은 출력이 나오지 않았었다. 그 때는 무한루프가 돌았었는데, mul 명령어 때문에 그렇다고 생각했었다. 하지만 jal 이 무조건 0 으로 떠서 고쳐줘야 한다는 수업을 듣고 'jal 을 올바르게 고치지 않아서

무한루프가 돈 건 아닐까?'라고 생각했다. 왜냐면 'coin'에도 잘못된 명령어가 나왔지만 무한루프가 돌진 않았기 때문이다.

```
00000000: 27bd ffd8 afbf 0024 afbe 0020 03a0 f025  '......$... ...%
00000010: 2402 0004 afc2 0018 8fc4 0018 0c00 0010  $...............
00000020: 0000 0000 afc2 001c 03c0 e825 8fbf 0024  ...........%...$
00000030: 8fbe 0020 27bd 0028 03e0 0008 0000 0000  ... '..(........
00000040: 27bd ffe0 afbf 001c afbe 0018 03a0 f025  '..............%
00000050: afc4 0020 8fc2 0020 2842 0002 1040 0004  ... ... (B...@..
00000060: 0000 0000 2402 0001 1000 0009 0000 0000  ....$...........
00000070: 8fc2 0020 2442 ffff 0040 2025 0c00 0010  ... $B...@ %....
00000080: 0000 0000 0040 1825 8fc2 0020 7062 1002  .....@.%... pb..
00000090: 03c0 e825 8fbf 001c 8fbe 0018 27bd 0020  ...%........'..
000000a0: 03e0 0008 0000 0000 0000 0000 0000 0000  ................
000000b0: 0a                                       .
```

- 어차피 구현하지 않은 명령어가 있어 제대로 돌아가진 않을테지만 그래도 한번 jal 을 올바르게 바꿔보았다. 원래 '0c000000'인 명령어를 '0c000010'으로 고쳐주었다.

```
-------------cycle 79----------------
        wrong insrtuction
```

- 역시나 올바르게 작동되진 않았지만, 무한루프가 도는 것이 아닌 'wrong instruction'이 뜨고 프로그램이 종료되는 것을 확인할 수 있었다.

(4) Problems & Solutions

① Single-Cycle 의지

- 처음 코드를 작성할 때, single-cycle 을 수정 보완하며 나아가는 식으로 작성하였다. Pipeline 이 single-cycle 의 업그레이드 버전이기도 하고, single-cycle 이라는 좋은 비교대상이 있으니 비슷하게 시작하면 될 것이라 생각했다. 하지만 변수 이름부터 프로그램 구성방식 하나하나 다 동일하게 적용시키니 오류도 많았고 오히려 멀리 돌아가는 느낌이 들었다. Single-cycle 을 벗어나 백지 상태로 처음부터 코드를 작성했더니 훨씬 수월했다. Single-cycle 에서는 필요한 함수나 변수만 가져다 쓰는 식으로 참고만 했다.

② Data Dependency

- ALU 가 있는 execute 부분에서만 forwarding 을 해주면 될 것이라 생각했었다. 그래서 execute 부분 외의 다른 단계에서 data dependency 는 신경을 쓰지 못했다. Single-cycle 의 출력값과 비교하며 data dependency 가 생길 수 있는 예시들을 찾아나갔고, jump 와 memory 부분에도 forwarding 을 하도록 추가했다.

```
266         //forwarding
267         if(wb_in->c.RegWrite){
268                 if(ex_out->i.rs == wb_in->WReg)
269                         v1 = wb_in->wb_v;
270                 if(ex_out->i.rt == wb_in->WReg)
271                         if(!(ex_out->c.ALUSrc))
272                                 v2 = wb_in->wb_v;
273         }
274         else if(mm_in->c.RegWrite){
275                 if(ex_out->i.rs == mm_in->WReg)
276                         v1 = mm_in->Address;
277                 if(ex_out->i.rt == mm_in->WReg)
278                         if(!(ex_out->c.ALUSrc))
279                                 v2 = mm_in->Address;
280         }
281         else if(ex_in->c.RegWrite){
282                 if(ex_out->i.rs == ex_in->WReg)
283                         v1 = ex_in->ALU_result;
284                 if(ex_out->i.rt == ex_in->WReg)
285                         if(!(ex_out->c.ALUSrc))
286                                 v2 = ex_in->ALU_result;
287         }
```

- 처음 execute 에서의 forwarding 을 위와 같이 순서 상관없이 if-else 문을 썼었는데 오류가 나서 바꾸게 되었다. 필요하지 않은 if 문에서 값을 먼저 채갔기 때문이다. 가장 최신에 업데이트된 레지스터의 값으로 forwarding 을 해줘야 하기 때문에 거리가 가까운 식으로(ex->mm->wb) if-else 문을 구성하는 현재의 방식으로 바꾸게 되었다.

- 또한 271, 278, 285 번째 줄에 있는 rt 가 dependency 가 생기는 단계의 Write Register 와 같은 지 확인하며 또 I-type 이 아닌지 확인하도록 하는 if 문을 처음에는 넣지 않았다. I-type 에서 rt 는 write register 이므로, rt 가 forwarding 이 될 경우는 없기 때문에 예외로 치는 if 문을 만들어 줘야 했다. 이도 역시 오류가 나서 알게 되었다.

③ Control Dependency

- Branch delay slot 을 하게 되면 branch 가 execute 에서 실행될 때까지 fetch, decode 에 잘못 들어온 명령어들을 무효화를 시켜주는 과정이 필요하다. 이를 구현하면서 여러 시행착오들이 있었다. 여러 방법을 시도하며 해결책을 찾아가던 중 if latch 를 무효화시키는 방법이 생각났고, 그에 맞게 코드를 작성했더니 해결되었다.

- Branch prediction 을 위한 변수들을 처음엔 구조체로 작성하지 않았다. 하지만 valid 라는 겹치는 변수명도 존재하고, 사용할 때도 헷갈려서 따로 구조체로 만들어 사용하기 편하게 바꿨다.

- Branch prediction 에서 처음엔 temp 라는 변수를 만들어서 예측했던 값이 맞았는지 틀렸는지 확인하도록 했다. 맞으면, temp=1 로 틀리면 0 으로 값을 업데이트 하도록 했다. 그 후, fetch 단계에서 temp 가 1 이면 target 에 저장된 주소로 PC 의 값을 업데이트하고, temp 가 0 이면 PC+4 를 PC 의 값을 업데이트 하도록 했다. 이렇게 하니 simple4 와 fib 까지는 문제없이 돌아갔지만, gcd 에 문제가 생겼다. invalidation 에서는 알맞게 돌아갔으므로 invalidation 의 출력값과 비교를 하는 등 여러 시도를 하던 중, gcd 의 assembly 를 확인해보니 다른 input 파일들과 달리 gcd 에는 branch 가 두개가 있었다.

temp 의 값이 branch 의 구별없이 업데이트 돼 오류를 일으켰던 것이다. 따라서 temp 도 BTB_PC 와 target 처럼 배열로 만들어 index 별로 구분을 해줬다.

- 이로 인해 문제는 해결이 됐지만, 다른 코드를 구현하던 중 더 간단한 방법이 생각나 지금처럼 BTB_PC 의 값과 target 이 같으면 PC+4 를 하는 방법으로 바꾸게 되었다.

(5) Personal Feeling

저번 프로젝트에서 얻은 교훈으로 이번 Simple Pipeline 에서는 모든 관련 개념을 완벽히 습득한 후 코드를 짜기 시작했다. 전에는 어떻게 짜야 할 지 생각해보지 않은 채 뭐든 써보는 것으로 시작했지만 시간도 오래 걸렸고, 코드를 짜면서도 맞게 하고 있는건지 또, 오류가 나면 어디가 틀린 건지 찾는 과정이 힘들었다. 이번엔 백지상태에서 시작하지 않고, 전체적인 큰 틀을 짠 후에 시작하였다. 또한 시간여유도 넉넉히 두고 시작했다.

처음에는 single-cycle 의 단계를 명령어 하나씩 처리하도록 하는 과정부터 코드를 짰는데, 이는 꽤 수월하게 진행되었다. 이전 single-cycle 에서 이미 구현했던 것들을 함수로 만들어 각각의 명령어가 들어올 수 있도록 해주기만 하면 됐기 때문이다. 하지만 hazard 를 고려하면서부터 급격히 난이도가 높아졌다. Data dependency 를 해결하지 않으면 코드가 아예 올바르게 돌아가지 않았기 때문에, 어느 부분이 잘못됐는지 찾기 쉽지 않았다. 또 data dependency 를 해결했다고 하더라도, control dependency 를 해결하지 않으면 branch 명령어가 있는 input 파일이 실행되지 않았기 때문에 이도 구현하는 데 어려움이 있었다. 하지만 그랬기에 사소한 실수 없이 코드를 작성하려고 더 노력했던 것 같다. Simple Calculator 와 Single-Cycle 프로젝트에서는, 정말 사소하고 황당한 실수들을 많이 했었다. 두번의 프로젝트로 사소한 실수지만 이를 찾아내는 데에는 몇시간, 며칠이 걸리는 것을 깨달았고 따라서 이번엔 더 꼼꼼히 확인하는 과정을 거쳤다. 그 결과, 이번에는 모든 오류들을 저번 프로젝트에 비해서 단기간 내에 찾을 수 있게 되었다. 물론 올바른 출력을 몰랐던 single-cycle 과는 달리 이번에는 비교대상이 있었으므로 더 찾는 게 수월했던 것도 있었다.

앞서 시간여유가 있게 프로젝트를 시작했다고 언급했고 물론 저번 프로젝트들에 비해 일찍 시작한 것은 맞다. 하지만 세번째 프로젝트는 시험기간과 겹친다는 것을 인지하지 못했었다. 또한 일찍 시작해 시간이 남을 줄 알고 초반에는 설렁설렁한 것도 있다. 결국 저번 프로젝트들처럼 시간에 쫓기게 되었다. 프로젝트를 시작하고 지금까지 정말 쉬는 날없이 달려온 것 같다. 사실 너무 힘들어 invalidation 의 기본 구현이 끝나고 branch prediction 이라는 추가구현은 하지 않으려 했다. 하지만 며칠동안 찝찝한 기분은 사라지지 않았고, 잠을 줄여서라도 해내야겠다고 생각했다. 이도 쉬운 방법인 1bit last-time predictor 으로 밖에 구현을 하지 못해 아쉬움이 남는다. 하지만, 지난 몇 주는 나에게 해야 할 일도, 고려할 일도 정말 많았던 기간이었고, 이 상황에서 branch prediction 까지 해낸 건 스스로 정말 수고했다고 생각이 든다. 뿌듯함도 말할 수 없이 크다.

프로젝트를 마무리하며 보고서를 쓰고 있는 지금 많은 생각이 종합적으로 든다. 무엇보다 제일 큰 건 성취감이지만, 아쉬움, 후련함 등 복합적인 기분이다. 항상 프로젝트를 끝낼 때마다 얻어가는 게 많은 것 같다. 저번 프로젝트들로 인해 생긴 경험은 일상생활을 살아가는 데도

영향을 주었다. 아마 이번 프로젝트도 나에게 큰 양분이 될 것 같다. 아직 하나의 프로젝트가 남아있으니 이 기분을 잊지 않고 완벽한 마무리를 할 수 있도록 노력해야 겠다.