Mobile Processor

Single-cycle MIPS

project #2

32171550 박다은

2020 Spring

Freedays left: 2

Contents

1.Project Overview

(1) Project Introduction

: Making single-cycle MIPS CPU emulator

Single Cycle machine에서는 한 명령어를 다음 단계를 거쳐 실행시킨다.

1) Instruction Fetch: 명령어가 메모리에서 CPU로 옮겨진다.

2) Instruction Decode: 명령어가 각 type별로 decode된다.

3) Execution: ALU가 작동되고, 결과값이 나온다.

4) Memory: Load/Store memory operation이 완료된다.

5) Write Back: 레지스터 값이 업데이트된다.

이 후 PC의 값이 업데이트되며 한 사이클은 끝이 난다. 이와 같은 사이클을 반복하다가 PC의 값이 0xffff:ffff가 되면 사이클을 종료한다.

(2) Project Goals

RISC 기반의 마이크로 프로세서 명령어 집합구조인 MIPS의 명령어가 Single-cycle machine에서 한 클록에 실행되는 과정을 이해한다. R-type, I-type, LW, SW 등 type별 instruction의 datapath와 control을 이해한다. Linux 환경에서 프로그래밍을 해보며 작동방법을 익히고 숙지한다.

(3) Concepts in MIPS simulator

31개의 MIPS 명령어에는 R_type, I_type, J_type이 있다. MIPS 실행 가능한 이진 파일(input.bin)을 input으로 사용한다. MIPS 이진 파일은 mips-cross-comiler에서 실행할 수 있다.

- R_type

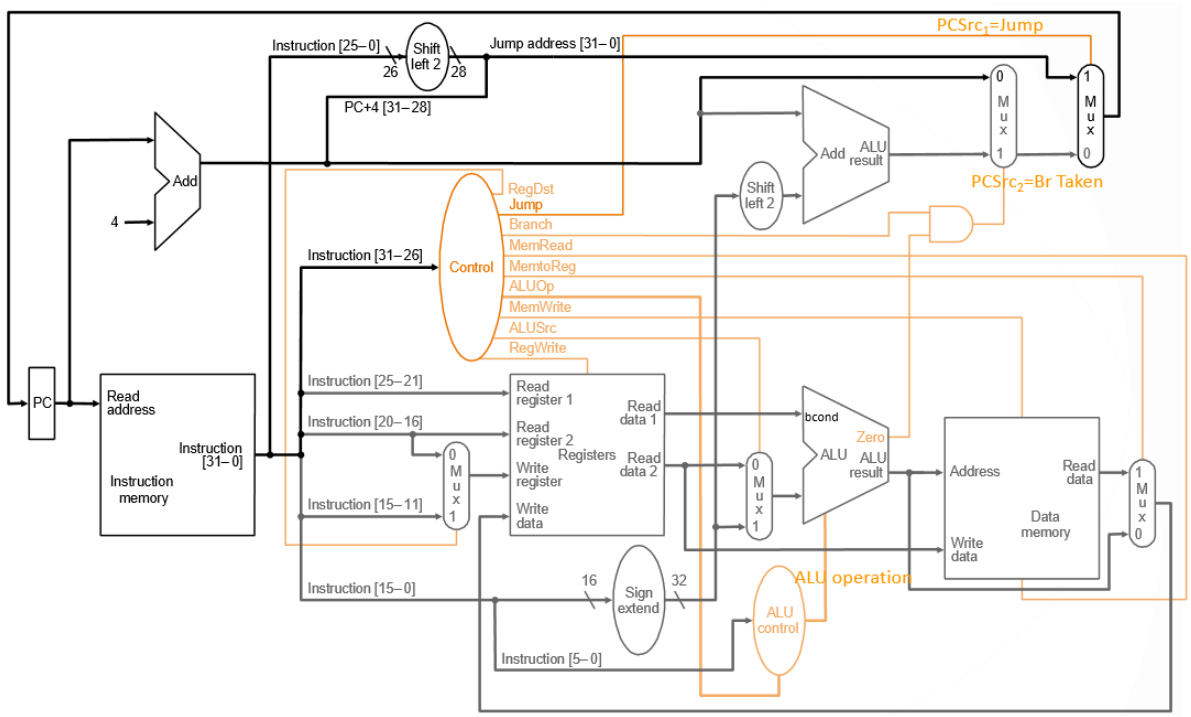
Opcode	Rs	Rt	Rd	shamt	Funct

- I_type

Opcode	Rs	Rt	Immediate

- J_type

Opcode	Address



다음은 single-cycle의 datapath와 control이다. Single Cycle은 한 클록에 한 명령어가 실행된다. 한 명령어가 완료되면 다음 명령어가 실행되는 구조이다.

File operations:

Fopen()을 이용해 파일을 열고, fclose()를 이용해 파일을 닫는다.

Error operations:

perror(): perror함수는 입력 인자로 전달한 문자열 뒤에 :과 함께 최근에 발생한 에러 메시지를 출력한다.

(4) Program structure

Open_file

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

| reverse()

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

| Print_Fetch

| Print_Decode

```
|        Print_Execute
|        Print_Memory
|        Print_WB
|        update PC
+------------------------
```

각 단계에서 각자의 역할은 결과를 출력하는 것이다.

먼저 뒤집힌 파일을 필요한 명령어로 바꿔주는 과정을 거친다. 그 후에 명령어 하나당 한 사이클을 돈다. 앞서 말했듯이, 한 명령어는 5단계를 거쳐 실행되고, 각각의 단계에 맞게 결과를 출력한다. 마지막으로 PC의 값을 업데이트한다.

(5) build environment

Compilation: Linux Assam, with GCC

To compile, please type:

gcc ./CA_32171550.c -o single_cycle_32171550

To run, please type:

./CA_32171550.c input.bin (input: simple, simple2, fib, gcd …)

2. Code & Review

(1) 주요코드

① Written function

A. Control

➢ Global

```
32 //control (instruction [31-26] -> opcode)
33 int RegDst, Jump, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWr
   ite;
34
35
36 void control(int op, int fun, int c_rs, int c_rt); //contol type별  구 분
```

- 먼저 control 관련 int 변수들을 main문 밖 전역변수로 선언해주었다. 그리고, control signal을 type별로 구분해 주는 함수를 따로 만들었다.

➢ function

```
278 //type 별  control 구 분
279 void control(int op, int fun, int c_rs, int c_rt){
280
281         RegDst = 0;
282         Jump = 0;
283         Branch = 0;
284         MemRead = 0;
285         MemtoReg = 0;
286         ALUOp = 0x100;
287         MemWrite = 0;
288         ALUSrc = 0;
289         RegWrite = 0;
```

- 첫 번째로 signal들을 초기화해주었다. 명령어의 type 별로 각각의 signal이 계속 변하기 때문에, 전역변수로 선언했을 때 초기화하지 않고, 함수 안에서 초기화해주었다. 이 함수는 main문 내의 while문에서 사용되므로, instruction에 따라 계속 signal들의 값이 바뀌게 된다. 0으로 초기화 한 다른 signal들과 달리 ALUOp는 0x100으로 초기화해주었다. 0을 제외한 임의의 수를 넣은 것이다. 그 이유는, '2-(1)-②-A-3.Execute(11페이지)'에서 더 자세히 설명하도록 하겠다.

```
291         if(op == 0){
292                 if(fun == 0x8 || fun == 0x9) //jr, jalr
293                         Jump = 1;
294                 else{ //R_type
295                         RegDst = 1;
296                         ALUOp = fun;
297                         RegWrite = 1;
298                 }
299         }
300         else if(op == 0x2 || op == 0x3) //jal, jr
301                 Jump = 1;
302         else if(op == 0x23){ //lw
303                 MemRead = 1;
304                 MemtoReg = 1;
305                 ALUOp = 0x20;
306                 ALUSrc = 1;
307                 RegWrite = 1;
308         }
309         else if(op == 0x2b){ //sw
310                 ALUOp = 0x20;
311                 MemWrite = 1;
312                 ALUSrc = 1;
313         }
314         else if(op == 0x4 || op == 0x5){ //beq, bne
315                 Branch = 1;
316                 ALUOp = op;
317         }
318         else { //I_type
319                 ALUOp = op;
320                 ALUSrc = 1;
321                 RegWrite = 1;
322         }
323         return;
324 }
```

- 이후에는 type별로 나눠서 각각의 control signal의 값을 바꿔주었다. 개념적인 것을 코드

로 옮긴 간단한 부분이지만, ALUOp에 대해선 조금 특이한 부분이 있다. 개념적으로 ALUOp는 R-type과 I-type에서는 각각 function code와 opcode이고, lw와 sw에서는 add, branch에서는 bcond이다. 하지만, ALUOp를 정수형으로 선언했으므로, lw/sw, branch에서는 이를 코드에 맞게 바꿔주는 과정이 필요했다.

- 먼저, lw와 sw에서 ALUOp의 값을 0x20으로 설정해주었다. 앞서 말했듯이, lw와 sw의 ALUOp는 원래 'add'이다. 명령어들 중 add, addu, addi, addiu가 연산장치 ALU에서 add 성질을 띈다. 이들의 ALUOp는 각각 0x20, 0x21, 0x8, 0x9인데, 이들과 같은 행동을 취하도록 이 4개 중 하나의 값을 lw와 sw의 ALUOp로 정한 것이다. 물론, 0x20대신 다른 세 개의 값으로 대체해도 아무런 상관이 없다.
- 그리고, branch에서는 ALUOp의 값을 opcode로 설정해주었다. Datapath를 보면 'bcond'와 signal 중 하나인 'Branch'를 and로 받아 1이면, branch가 'taken'된다. 즉, 'bcond'와 'Branch'의 값이 모두 1이어야 실행된다. 따라서, opcode가 beq나, bne이면 일단 둘다 Branch의 값을 1로 받고, ALU 부분에서 조건을 충족함에 따라 bcond의 값이 바뀌게하였다. ALUOp가 0x4(beq)거나 0x5(bne)일 때, bcond의 값을 정하도록 main문 내의 ALU 부분에서 설정하였으므로, ALUOP의 값을 opcode로 설정한 것이다.

B. Reverse

```
326 //1byte씩 뒤 집 혀 있 는  instruction 원 래 대 로  뒤 집 기
327 void reverse(FILE *f){
328         int rev_inst = 0;
329         int index = 0;
330         int data = 0;
331         size_t ret = 0;
332         while(1){
333                 int h1 = 0;
334                 int h2 = 0;
335                 int h3 = 0;
336                 int h4 = 0;
337                 int rev_inst = 0;
338
339                 //reading file to eof
340                 ret = fread(&data, sizeof(data), 1, f);
341                 if(ret == 0) break;
342
343                 //printout
344                 printf("orig 0x%08x    ", data);
345
346                 h1 = ((data & 0xff) << 24);
347                 h2 = ((data & 0xff00) << 8);
348                 h3 = ((data & 0xff0000) >> 8) & 0xff00;
349                 h4 = ((data & 0xff000000) >> 24) & 0xff;
350
351                 rev_inst = h1 | h2 | h3 | h4;
352
353                 Memory[index/4] = rev_inst;
354                 printf("[0x%08x] 0x%08x\n",
355                                 index, Memory[index/4]);
356                 index = index + 4;
357         }
```

- 파일을 이진모드로 받으면 명령어들이 1byte씩 뒤집힌 채로 나타난다. 예를 들어, 기대한 명령어가 27bd8010이면, 파일에서는 1080bd27로 나오게 된다. 따라서, 이를 원하는 방향으로 뒤집어주는 과정이 필요하다. 따라서 and(&)를 사용해 1byte씩 나눠주고, 이를 shift를 사용해 원래 자리를 찾아가도록 하였다. 그리고 뒤집힌 명령어를 Memory에 저장해주었다.

C. Count

➢ Global

```
25 //count
26 int inst_count = 1;
27 int r_count = 0;
28 int i_count = 0;
29 int j_count = 0;
30 int m_count = 0;
31 int b_count = 0;
```

- count와 관련된 변수들을 global에 선언해주었다. 다른 변수들은 0으로 초기화했지만, inst_count는 1로 초기화했다. 그 이유는, 한 cycle을 출력할 때, instruction의 index도 같이 출력되게 하였는데(참고: 2-(2)출력 화면), main에서 instruction의 개수가 카운트되기 전에 출력이 먼저 나오도록 코드를 짰다. 따라서 0으로 초기화하면 맨 처음 명령어가 실행됐지만, index가 0으로 뜨기 때문에 1로 초기화한 것이다. 이렇게 하면 프로그램이 종료될 때, 명령어가 count된 개수는 1개가 증가되므로, 출력할 때 inst_count에서 1을 뺀 값을 출력하도록 설정하였다.

➢ Function

```
361 //type별  instruction 개 수  count
362 void count(int op){
363         if(op == 0) r_count++; //r_type
364         else if(op == 0x2 || op == 0x3) j_count++; //j_type
365         else if(op == 0x23 || op ==0x2b){ //lw. sw(i_type)
366                 i_count++;
367                 m_count++;
368         }
369         else i_count++; //i_type
370
371         return ;
372 }
```

- 프로그램이 종료되기 직전에 총 instruction의 개수와 각각의 type별 instruction개수, memory access instruction개수, taken branch의 개수를 출력해야 한다. 따라서 이들의 개수를 세는 함수를 따로 만들어주었다. taken branch의 개수를 셀 때는 조건을 충족했는지 확인해야 하므로 함수의 인자인 opcode만으로 개수를 셀 수 없어, main문 안의 관련된 부분에서 따로 개수를 셌다.

D. PrintfE

```
376 //execute 후  결과  출력함수
377 void PrintfE(int op, int fun, int p_rs, int p_rt, int p_v1, int p_v2, int p_
    v3){
378
379         if(op == 0){ //R_type
380                 if(fun == 0x20 || fun == 0x21) //add, addu
381                         printf("[E] result(0x%x) = R[%d](0x%x) + R[%d](0x%x)
    \n",
382                                 p_v3, p_rs, p_v1, p_rt, p_v2);
383
384                 else if(fun == 0x24) //and
385                         printf("[E] result(0x%x)(%d) = R[%d](0x%x) & R[%d](0
    x%x)\n",
386                                 p_v3, p_v3, p_rs, p_v1, p_rt, p_v2);
387
388
389                                 ...중간생략...
```

- 사이클의 세 번째 과정인 Execute 후 실행되는 함수이다. Main문 내의 Execute는 ALUOp를 switch문으로 받도록 구성되어 있는데, R-type, I-type관계없이 add, and, or 등 명령어의 연산에 맞게 실행되도록 하였다. 하지만, 출력될 때는 구분을 해서 출력이 되도록 만들고 싶었다. 예를 들어, R-type에서 add는 레지스터들의 합이고, I-type에서 add는 레지스터와 simm값의 합이다. 각각에 맞게 출력하도록 하고싶어서 이 함수를 만들었다. 단순히 출력만 해주는 함수이고, 따로 특별한 건 없어서 코드를 중간생략하였다.

② Main문

A. fopen

```
51          if(argc == 2) //2번째  자리에  파일명  입력하면  그  파일을  읽음
52                  fp = fopen(argv[1], "rb");// "rb" : 읽기모드  + 이진파일모드
53          if(fp == NULL){ //입력하지  않았거나  잘못입력했으면  오류표시
54                  perror("no such input file"); //오류메세지  출력함수
55                  return 0; //프로그램  종료
56          }
```

- 다른 파일을 불러올 때, 매번 코드를 고치는 게 번거로워서 첫 번째 프로젝트에서 사용했던 argument를 이용해 코드를 바꿔주었다. 입력창에서 파일명을 입력하면 그 파일을 불러오도록 바꾼 것이다.

B. While문

```
62          while(1){
63
64                  //PC가  0xffff:ffff가  되면  cycle 종료
65                  if(PC == 0xffffffff) {
66                          printf("\n PC : 0x%8x\n", PC);
67                          printf(" <Halt program>\n\n");
68                          break;
69                  }
70                  printf("\n\ninstruction %d-----------PC : 0x%x-------------\
    n",
```

```
71                                  inst_count, PC);
72
73                  //1.fetch -> 2.decode -> 3.execution / ALU -> 4.Mem -> 5.WB
```

- 다섯단계에 걸쳐 한 클록을 반복하는 while문이다. PC가 0xffff:ffff가 될 때까지 반복한다. 한 클록의 시작에서 instruction의 index와 PC의 값을 출력하도록 했다.

➢ 1. Fetch

```
76                  //1.fetch-------------------------------------------------------
77                  //getting an instruction from memory
78                  //divide by 4 to get the index into the array
79                  inst = Memory[PC/4];
80
81                  printf("[F] 0x%08x\n", inst);
82
83                  //instruntion이  0x00000000이 면  남 은  while문  실 행 X.
84                  if(inst == 0) {
85                          printf(" none\n");
86                          PC = PC + 4;
87                          inst_count++;
88                          continue;
89                  }
```

- reverse에서 메모리에 저장했던 명령어를 가져오는 과정이다.
- Instruction이 0x00000000이면 while문의 남은 부분을 실행하지 않도록 continue를 사용해 코드를 짰다. 0x00000000을 opcode가 0이고, func도 0인 명령어(sll)로 고려를 해야 할지 고민이 있었다. 하지만 명령어에 속한다 하더라도 어차피 아무 변화도 일어나지 않으므로 따로 예외처리를 했다. 하지만 instruction count는 세도록 했다.

➢ 2. Decode

```
91                  //2. decode----------------------------------------------------
92                  //identify the instruction to execute
93                  int opcode = (inst & 0xfc000000) >> 26;
94                  int rs     = (inst & 0x03e00000) >> 21;
95                  int rt     = (inst & 0x001f0000) >> 16;
96                  int rd     = (inst & 0x0000f800) >> 11;
97                  int shamt  = (inst & 0x000007c0) >> 6;
98                  int func   = (inst & 0x0000003f) >> 0;
99                  int imm    = (inst & 0x0000ffff) >> 0;
100                 int addr   = (inst & 0x03ffffff) >> 0;
101                 int simm   = (imm >> 15) ?
102                                 (0xffff0000 | imm) : imm;
103                 int zimm   = imm;
104                 int Br_Addr= simm << 2;/*(imm >> 15) ?
105                                 (0xfffc0000 | imm << 2) : (imm << 2);*/
106                 int J_Addr = ((PC+4 & 0xf0000000) | addr) << 2;
107
108                 printf("[D] opcode: 0x%x, rs: %d, rt: %d, rd: %d, shamt:%d, f
    unc: 0x%x\n"
109                         , opcode, rs, rt, rd, shamt, func);
110                 printf("        simm 0x%08x (%d)\n", simm, simm);
111
```

```
112                 count(opcode); //opcode를 구분해 instruction type별 개수 세기
113                 control(opcode, func, rs, rt); //control type별 구분
```

- 명령어를 각각의 opcode, rs, rt, rd 등으로 분해하는 과정이다.
- 예를 들어, 명령어에서 opcode를 분해해온다고 하면, opcode는 명령어에서 맨 앞 6bit이므로, 앞의 6bit는 1이고 나머지 26bit는 0으로 구성된 32bit(0xfc000000)를 원래의 명령어와 and(&)를 해 맨 앞 6bit를 제외한 나머지를 0으로 만든다. 그리고 0이된 26bit를 shift를 이용해 없애고 앞의 6bit만 가지고 오는 것이다.
- 위와 같은 방법으로 나머지들도 분해를 해주고, 분해된 값을 출력해주었다.
- count함수를 위 과정에 위치시켰다. opcode가 선언되고 초기화된 이후면 어디든 상관이 없지만, 초기화된 직후로 위치시켰다.
- control 함수는 execute전에 위치해야 하므로, 마찬가지로 decode 과정 직후로 위치시켰다.

➢ 3. Execute

```
115                 //3. execute-------------------------------------------------
116                 //run the ALU
117                 int v1, v2, ALU_result;
118                 int bcond = 0; //branch taken or not taken 구분
119
120                 //ALU에 사용할 v1, v2초기화
121                 v1 = Regs[rs];
122
123                 if(ALUSrc) v2 = simm; //ALUSrc = 1이면 v2 = simm
124                 else v2 = Regs[rt]; //0이면 v2 = R[rt]
125
126                 //zimm or imm or shamt 사용하는 것들 예외처리
127                 if(opcode == 0xc || opcode == 0xd)
128                         v2 = zimm;
129                 if(opcode == 0xf)
130                         v2 = imm;
131                 if(opcode == 0 && (func == 0x0 || func == 0x2))
132                         v1 = shamt;
```

- Datapath에서 v1은 항상 R[rs]이고, v2는 ALUSrc의 값에 따라 달라진다. ALUSrc가 1일 때(I-type), v2 = simm이고, 0이면(R-type) v2 = R[rt]이다.
- 예외도 존재한다. andi(0xc)와 ori(0xd)에서 simm대신 zimm이 쓰이고, lui(0xf)에서는 simm대신 imm이 쓰인다. 또, sll(0/0x0)과 srl(0/0x2)(R-type)에서 변수는 R[rt]와 shamt이므로 R[rs]대신 shamt를 쓰도록 v1을 shamt로 바꿔주었다.

```
134                 switch(ALUOp){
135                         case 0x100: //j, jal, jalr, jr
136                                 break;
137                         case 0x20: //add //+sw, lw
138                         case 0x21: //addu
139                         case 0x8: //addi
140                         case 0x9: //addiu
```

```
141                             ALU_result = v1 + v2;
142                             break;
143                     case 0x24: //and
144                     case 0xc: //andi
145                             ALU_result = v1 & v2;
146                             break;
147
148                             ...중 간 생 략 ...

169                     case 0x4: //beq
170                             if(v1 == v2)
171                                     bcond = 1;
172                             break;
173                     case 0x5: //bne
174                             if(v1 != v2)
175                                     bcond = 1;
176                             break;
177                     case 0x0: //sll
178                             ALU_result = v2 << v1;
179                             break;
180                     default: //잘 못 된  instruction 명 령  시  프 로 그 램  종 료
181                             printf("\twrong insrtuction\n");
182                             return 0;
183             }
184
185             //Excute 단 계  printf
186             PrintfE(opcode, func, rs, rt, v1, v2, ALU_result);
```

- ALUOp에 따라 각각의 연산을 실행하도록 switch문을 사용해주었다.
- 앞서 control함수에서 ALUOp를 0x100으로 초기화했었다. 다른 변수들과 같이 0으로 초기화할 수 없었던 이유는 switch문의 case 중 0x00이 존재하기 때문이다. 따라서, case들과 겹치지 않는 임의의 수로 0x100을 정한 것이다. 겹치지만 않다면 다른 수가 와도 상관없다.
- switch문이 끝나고 연산하는 과정을 출력하는 PrintfE함수를 호출했다.
- 135-136: control함수에서 jump(j, jal, jalr, jr)는 따로 ALUOp의 값을 변경하지 않도록 했다. switch문에서 어느 case에도 속하지 않으면 프로그램을 종료하도록 default를 설정했다. 따라서 원래는 ALU와 관련이 없는 jump이지만, 하나의 case를 따로 만들어주었다. ALUOp가 0x100이면 switch문을 빠져나오도록 하였다.
- 169-176: 조건이 충족됐을 때, bcond의 값을 1로 바꿔주었다.

➢ 4, Memory

```
189             //4. memory-----------------------------------------------------
190             //access memory
191             //addr
192             //MemRead
193             //value = Memory[(addr>>2)];
194
195             //lw
196             int Address;
197             if(MemRead){ //메 모 리  가 져 오 기
```

```
198                     Address = Memory[ALU_result];
199                     printf("[M] Read Memory\n");
200                     printf("        0x%x <= M[0x%x]\n",
201                                     Address, ALU_result);
202             }
203
204             //sw
205             if(MemWrite){ //메 모 리  저 장 하 기
206                     Memory[ALU_result] = Regs[rt];
207                     printf("[M] Write Memory\n");
208                     printf("        M[0x%x] <= R[%d](0x%x)\n",
209                                     ALU_result, rt, Regs[rt]);
210             }
```

- 이 과정에는 lw와 sw일 때만 속한다.
- MemRead가 1이면(lw) ALU에서 ALU에서 연산한 값의 메모리를 address로 가져오도록 하였다.
- MemWrite가 1이면(sw) R[rt]의 값을 연산한 값의 메모리에 저장하도록 하였다.

➢ 5.Write Back

```
213             //5. write back-------------------------------------------------
214             //update register values
215             int WriteR;
216             if(RegDst) WriteR = rd; //RegDst = 1이 면  WriteR = instruction
    [15-11](rd)
217             else WriteR = rt; //0이 면  WriteR = instruction[20-16](rt)
218
219             if(RegWrite){
220                     if(MemtoReg){ //lw
221                             Regs[WriteR] = Address;
222                             printf("[W] R[%d] = 0x%x\n",
223                                     WriteR, Address);
224                     }
225                     else{
226                             Regs[WriteR] = ALU_result;
227                             printf("[W] R[%d] = 0x%x\n",
228                                     WriteR, ALU_result);
229                     }
230             }//-------------------------------------------------------------
```

- RegDst가 1이면(R-type) WriteR이 rd가 되도록 하였고, 0이면 WriteR이 rt가 되도록 하였다.
- RegWrite가 1일 때(R-type, lw, I-type)만 레지스터에 값을 쓰도록 하였다. MemtoReg이 1이면(lw) Address의 값을 레지스터에 저장하도록 하였고, 0이면(R-type, I-type) ALU의 결과값을 레지스터에 저장하도록 하였다.

➢ Update PC

```
234                 //Update PC
235                 if(Jump){
236                         if(opcode == 0 && func == 0x8) //jr
237                                 PC = Regs[rs];
238                         else if(opcode == 0 && func == 0x9){ //jalr
239                                 Regs[rd] = PC + 8; //rd : 31
240                                 PC = Regs[rs];
241                         }
242                         else if(opcode == 0x2) //j
243                                 PC = J_Addr;
244                         else if(opcode == 0x3){ //jal
245                                 Regs[31] = PC + 8;
246                                 PC = J_Addr;
247                         }
248                         printf("Jump to 0x%x\n", PC);
249                 }
250                 else if(Branch && bcond){ //branch taken
251                         PC = PC + 4 + Br_Addr;
252                         printf("Jump to 0x%x\n", PC);
253                         b_count++;
254                 }
255                 else PC = PC + 4;
```

- PC의 값을 update시키는 부분이다.
- Jump가 1이거나 Branch && bcond가 1이면, 각 명령어에 따라 PC의 값이 이동하도록 하고 이동하는 값을 출력하도록 하였다. 그렇지 않으면, PC = PC + 4를 수행하도록 하였다.
- Jalr에 대해서는 2-(3)에서 더 자세히 설명하도록 하겠다.

➢ Changed architectural state

```
257                 //changed architectural state
258                 //register, PC, memory가  변 했 을  시  출 력
259                 printf("\n");
260                 printf("changed PC      : PC     = 0x%x\n", PC);
261                 if(RegWrite)
262                         printf("changed register: R[%d]  = 0x%x\n", WriteR, Re
    gs[WriteR]);
263                 if(MemWrite)
264                         printf("changed memory  : M[0x%x]= 0x%x\n", ALU_result
    , Regs[rt]);
265         }
```

- Register, PC, Memory의 값이 변했다면 클록에 마지막에서 변한 값을 출력하는 부분이다. PC의 값은 항상 바뀌므로 따로 if문을 설정하지 않았지만, Register와 Memory는 값이 변했을 때만(RegWrite = 1, MemWrite = 1일 때) 출력하도록 하였다. 이 과정을 마지막으로 한 사이클이 종료된다.

C. After the completion of the program

```
267         printf("\nfinal return value(R[2])              : %d\n", Regs[2]);
268         printf("number of excuted instructions       : %d\n", inst_count-1);
269         printf("number of R-type instructions        : %d\n", r_count);
270         printf("number of I-type instructions        : %d\n", i_count);
271         printf("number of J-type instructions        : %d\n", j_count);
272         printf("number of memory access instructions : %d\n", m_count);
273         printf("number of taken branches             : %d\n\n\n", b_count);
274
275         return 0;
276 }
```

- 모든 사이클이 끝나고 프로그램을 완전히 끝내기 전 main문 내에 결과값(R[2])과 시행된 각 type별 명령어들을 count값을 출력하도록 한 부분이다.

(2) 출력화면

```
orig 0xd0ffbd27    [0x00000000] 0x27bdffd0
orig 0x2c00bfaf    [0x00000004] 0xafbf002c
orig 0x2800beaf    [0x00000008] 0xafbe0028
orig 0x21f0a003    [0x0000000c] 0x03a0f021
orig 0x98120224    [0x00000010] 0x24021298
            ...중 간 생 략 ...
```

- 먼저, reverse함수에서 시행된 뒤집힌 명령어를 원하는 값으로 바꾸는 과정이 출력된다.

```
instruction 1-----------PC : 0x0-------------
[F] 0x27bdffd0
[D] opcode: 0x9, rs: 29, rt: 29, rd: 31, shamt:31, func: 0x10
        simm 0xfffffffd0 (-48)
[E] result(0xfffd0) = R[29](0x100000) + simm(0xfffffffd0)
[W] R[29] = 0xfffd0

changed PC      : PC     = 0x4
changed register: R[29]  = 0xfffd0


instruction 2-----------PC : 0x4-------------
[F] 0xafbf002c
[D] opcode: 0x2b, rs: 29, rt: 31, rd: 0, shamt:0, func: 0x2c
        simm 0x0000002c (44)
[E] result(0xffffc) = R[29](0xfffd0) + simm(0x2c)
[M] Write Memory
        M[0xffffc] <= R[31](0xffffffff)

changed PC      : PC     = 0x8
changed memory  : M[0xffffc]= 0xffffffff

                ...중 간 생 략 ...

instruction 1061-----------PC : 0x44-------------
[F] 0x03e00008
[D] opcode: 0x0, rs: 31, rt: 0, rd: 0, shamt:0, func: 0x8
        simm 0x00000008 (8)
Jump to 0xffffffff

changed PC      : PC     = 0xffffffff
```

```
PC : 0xffffffff
<Halt program>


final return value(R[2])             : 1
number of excuted instructions       : 1061
number of R-type instructions        : 222
number of I-type instructions        : 637
number of J-type instructions        : 65
number of memory access instructions : 486
number of taken branches             : 45
```

- 한 사이클마다 명령어의 index와 현재 PC의 값이 출력되는 것을 시작으로 앞서 설명한 [F][D][E][M][W] 5단계의 출력이 각각 뜬다. 마지막으로는 PC, Register, Memory의 값이 바뀌었다면 바뀐 값을 출력한다.
- PC가 0xffffffff가되면 사이클이 종료되는 것을 보여주고, 종료된 후 R[2]값과 count한 수들이 출력된다.

(3) jalr(jump and link using register)

```
238                         else if(opcode == 0 && func == 0x9){ //jalr
239                                 Regs[rd] = PC + 8; //rd : 31
240                                 PC = Regs[rs];
241                         }
```

- jalr에서, 개념 상 rd는 31이고 R[rd]에는 return address가 저장된다. 처음에 이 'return address'가 무엇인지 이해가 되지 않아 구글링을 해보았다. 명확한 해답이 나오진 않았지만, jal도 R[31]에 'return address'를 저장한다고 하였다. Green sheet를 보면 jal에서 R[31]은 PC+8을 저장하도록 되어있다. 따라서 jalr에서도 같을 것이라 생각돼 R[rd]에 PC+8을 저장하도록 하였다.
- rs가 jump target address이므로 PC가 R[rs]로 이동하도록 하였다.

```
00000000: 27bd ffd8 afbf 0024 afbe 0020 03a0 f021  '......$... ...!
00000010: 2402 000a afc2 0018 8fc4 0018 0c00 0010  $...............
00000020: 0000 0000 afc2 001c 03c0 e821 8fbf 0024  ...........!...$
00000030: 8fbe 0020 27bd 0028 03e0 0008 0000 0000  ... '..(........
00000040: 27bd ffd0 afbf 002c afbe 0028 afb0 0024  '......,...(...$
00000050: 03a0 f021 afc4 0030 8fc2 0030 0000 0000  ...!...0...0....
00000060: 2842 0003 1040 0004 0000 0000 2402 0001  (B...@......$...
00000070: 0800 002e 0000 0000 8fc2 0030 0000 0000  ...........0....
00000080: 2442 ffff 0040 2021 0c00 0010 0000 0000  $B...@ !........
00000090: 0040 8021 8fc2 0030 0000 0000 2442 fffe  .@.!...0....$B..
000000a0: 0040 2021 0c00 0010 0000 0000 0202 1021  .@ !...........!
000000b0: afc2 0018 8fc2 0018 03c0 e821 8fbf 002c  ...........!...,
000000c0: 8fbe 0028 8fb0 0024 27bd 0030 03e0 0008  ...(...$'..0....
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000e0: 0a                                       .
```

- 코드를 짠 후 fib.bin의 jal을 jalr로 바꾸는 것을 시도해보았다. fib.bin 파일에서 :%!xxd를

입력해 jal인 명령어가 무엇이 있는 지 확인해보았다.

- jal의 opcode는 0x3이므로 이를 이진수로 바꾸면 000011이다. 앞에서부터 4bit로 나눠 다시 16진수로 바꾸면 0c이므로 0c로 시작하는 명령어들을 찾았고, 위에서 표시한 것과 같이 세 개의 명령어가 있었다. 이 3개는 모두 0c00 0010으로 동일했다. 이를 해석하면, R[31]에는 PC+8이 저장되고, 0x40으로 점프하는 명령어이다.
- 이 jal 명령어를 jalr로 바꾸려면 R[rs]의 값이 0x40이어야 했다. R[rs]의 값을 어떻게 설정할 지 고민하다가 이 예제파일에서는 사용되지 않는 하나의 레지스터를 정해 그 레지스터에 addi를 이용해 값을 넣는 것을 생각했다.
- 사용되지 않는 레지스터는 임의로 R[15]로 정했고, R[15]에 0x40을 넣는 addi 명령어를 또 따로 만들어야 했다. 이를 위해선 또 쓰지 않는 레지스터가 필요했고, 이를 R[16]으로 정했다. 원하는 과정을 표현하면 다음과 같다.

  R[15] = R[16](0x0) + 0x40;

  이 식을 컴퓨터가 알아볼 수 있는 언어로 바꿔야했다. addi를 이용하는 것이므로 opcode는 0x8이고, rs는 16, rt는 15, imm은 0x40인 I-type의 명령어이다. 이를 이진수로 바꾸면 opcode:001000, rs:10000, rt:01111, imm:0000 0000 0100 0000이다.

  즉, 명령어는 0010 0010 0000 1111 0000 0000 0100 0000이고, 이를 다시 16진수로 바꾸면 0x220f0040이된다.

- 궁극적 목표인 jalr도 명령어로 바꿔줘야 한다. 인간친화적 식으로 표현하면 다음과 같다.

  R[rd(31)] = PC+8; PC = R[rs(15)];

  이를 컴퓨터친화적으로 바꾸기 위해, 앞서 했던 것처럼 분해하면 opcode = 0, rs = 15, rt = 0, rd = 31, func = 0x9이다(개념 상 rd, rs, funct을 제외한 모든 bit는 0이다.). 이를 이진수로 표현하면 opcode:000000, rs:01111, rt:00000, rd:11111, func:000 0000 1001이고, 합쳐진 명령어는 0000 0001 1110 0000 1111 1000 0000 1001이다. 이를 다시 16진수로 표현하면, 0x01e0f809이다.

```
00000000: 27bd ffd8 afbf 0024 afbe 0020 03a0 f021  '......$... ...!
00000010: 2402 000a afc2 0018 8fc4 0018 220f 0040  $..............
00000020: 01e0 f809 afc2 001c 03c0 e821 8fbf 0024  ...........!...$
00000030: 8fbe 0020 27bd 0028 03e0 0008 0000 0000  ... '..(........
00000040: 27bd ffd0 afbf 002c afbe 0028 afb0 0024  '......,...(...$
00000050: 03a0 f021 afc4 0030 8fc2 0030 0000 0000  ...!...0...0....
00000060: 2842 0003 1040 0004 0000 0000 2402 0001  (B...@......$...
00000070: 0800 002e 0000 0000 8fc2 0030 0000 0000  ...........0....
00000080: 2442 ffff 0040 2021 01e0 f809 0000 0000  $B...@ !........
00000090: 0040 8021 8fc2 0030 0000 0000 2442 fffe  .@.!...0....$B..
000000a0: 0040 2021 01e0 f809 0000 0000 0202 1021  .@ !...........!
000000b0: afc2 0018 8fc2 0018 03c0 e821 8fbf 002c  ...........!...,
000000c0: 8fbe 0028 8fb0 0024 27bd 0030 03e0 0008  ...(...$'..0....
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000e0: 0a                                       .
```

- 위 과정이 끝나면, 이제 jal 대신, jalr을 넣어주고 jalr 전에 R[15]에 0x40을 저장하는 addi만 넣어주면 된다. Vi 명령어 'i'를 통해 다음과 같이 바꿔주었다.

- 전체 명령어의 개수가 바뀌면 안될 것 같아 이를 유지하도록 했다. 마침, 순서상 첫 번째 jal 명령어의 바로 뒤인 '00000020: 줄'의 첫번째 명령어가 0000 0000이라 이를 01e0 f809(jalr)로 바꿔주었고, 원래 jal자리에는 220f 0040(addi)을 넣어주었다. 다른 jal자리들은 새로운 jalr명령어로 대체해주었다.

```
'½ÿØ¯¿^@$¯¾^@ ^C ð!$^B^@
¯Â^@^X<8f>Ä^@^X"^O^@@^Aàø        ¯Â^@^\^CÀè!<8f>¿^@$<8f>¾^@ '½^@(^Cà^@^H^@^@^@
^@'½ÿÐ¯¿^@,¯¾^@(¯°^@$^C ð!¯Ä^@0<8f>Â^@0^@^@^@^@(B^@^C^P@^@^D^@^@^@^@$^B^@^A^H
^@^@.^@^@^@^@<8f>Â^@0^@^@^@^@$Bÿÿ^@@ !^Aàø     ^@^@^@^@^@@<80>!<8f>Â^@0^@^@^@^
@$Bÿþ^@@ !^Aàø    ^@^@^@^@^B^B^P!¯Â^@^X<8f>Â^@^X^CÀè!<8f>¿^@,<8f>¾^@(<8f>°^@$'
½^@0^Cà^@^H^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@^@
```

- :%!xxd -r을 입력해 위와 같이 binary모드를 풀어주고 저장을 해 실행을 시켜보았다. 결과는 무한루프였다. 많은 허무감이 들었다. 인터넷에 검색을 해봐도 해결책이 나오지 않았고, 혼자 실패의 이유를 생각해본 것은 이렇다. 출력된 것을 분석해보니 명령어가 0000 0000이어도 자리 수를 채우는 것인지, 나름의 이유가 있어 보였고 함부로 바꾸면 안되는 것 같았다. 따라서 딱 jal만 바꾸도록 해보았다.

```
00000000: 27bd ffd8 afbf 0024 afbe 0020 03a0 f021  '......$... ...!
00000010: 2402 000a afc2 0018 8fc4 0018 01e0 f809  $............"..@
00000020: 0000 0000 afc2 001c 03c0 e821 8fbf 0024  ...........!...$
00000030: 8fbe 0020 27bd 0028 03e0 0008 0000 0000  ... '..(........
00000040: 27bd ffd0 afbf 002c afbe 0028 afb0 0024  '......,...(...$
00000050: 03a0 f021 afc4 0030 8fc2 0030 0000 0000  ...!...0...0....
00000060: 2842 0003 1040 0004 0000 0000 2402 0001  (B...@......$...
00000070: 0800 002e 0000 0000 8fc2 0030 0000 0000  ...........0"..@
00000080: 2442 ffff 0040 2021 01e0 f809 0000 0000  $B...@ !........
00000090: 0040 8021 8fc2 0030 0000 0000 2442 fffe  .@.!...0"..@$B..
000000a0: 0040 2021 01e0 f809 0000 0000 0202 1021  .@ !...........!
000000b0: afc2 0018 8fc2 0018 03c0 e821 8fbf 002c  ...........!...,
000000c0: 8fbe 0028 8fb0 0024 27bd 0030 03e0 0008  ...(...$'..0....
000000d0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
000000e0: 0a                                       .
```

- 위와 같이 말이다. 하지만, 이렇게 하면 R[15]에 0x40을 넣는 과정이 생략되게 된다.

```
Regs[15] = 0x40;//fib.bin에 서  jal->jalr과 정  때 문 에
```

- 따라서 다른 방법은 생각나지 않아, main문 내에서 강제로 R[15]를 0x40으로 초기화해주었다. 이렇게 해주니, 잘 돌아갔다. 코드에서 jalr 부분이 알맞게 설정된 것이고, fib.bin에서 jal을 jalr로 잘 바꾸었다는 증거이긴 했지만, 원하던 바가 아니라 약간의 찝찝함은 남았다.

(4) Problems and Solutions

A. Control signal

```
                if(opcode == 0) RegDest = 1;
                if((opcode != 0) && (opcode != 0x4) && (opcode != 0x5)) ALUSrc = 1
;
                if(opcode == 0x23) MemtoReg = 1;
                if((opcode != 0x2b) && (opcode != 0x4) && (opcode != 0x5) && (opco
de != 0x2) && (opcode != 0x3)) RegWrite = 1;
                if(opcode == 0x23) MemRead = 1;
                if(opcode == 0x2b) MemWrite = 1;
```

- 현재 함수로 만들어진 control부분은 여러 과정을 거친 뒤 함수로 만들어졌다. 제일 처음에는 위와 같이 각각의 signal이 1이되는 조건들을 찾아 signal 별로 if문을 만들었다. 하지만 control함수에서 언급했듯이 예외적인 부분도 많고, 위의 signal들은 이대로도 구현이 되지만, ALUOp 등의 signal은 이 방법으로는 구현하기 어려운 부분이 있기도 해서 지금과 같이 type별로 나누어 구현했다.

- 바꾼 코드는 각 type별로 signal의 값들을 넣어주는 간단한 과정이지만, 바뀌기 전보다 main 문내에서 차지하는 비율이 꽤 커졌다.

```
struct control{
        int RegDst;
        int Jump;
        int Branch;
        int MemRead;
        int MemtoReg;
        int ALUOp;
        int MemWrite;
        int ALUSrc;
        int RegWrite;
}c;
```

- 따라서 처음엔 control의 signal들을 배열에 넣을까 생각했다. control[8]을 만들어 control[0]이 RegDst이고, control[1]이 Jump라는 등 그냥 변수의 구실만 하도록 할까 생각했다. 배열로 만들면, 각각의 signal 변수들에 값을 넣기도 편하고 main문 길이도 훨씬 줄어들 것 같았다. 하지만, 각 변수의 이름을 만들어줄 수가 없어, 사용할 때도 헷갈리고 나를 뺀 다른 사람은 코드를 이해하기 힘들 것 같았다. 그 다음으로는 위와 같은 구조체를 생각했다.

```
        if(opcode == 0){ //R_type
                if(func == 0x8 || func == 0x9){
                        struct control c = {0,1,0,0,0,0,0,0,0};
                }
                else{
                         struct control c = {1,0,0,0,0,func,0,0,1};
                }
        }
        else if(opcode == 0x2 || opcode == 0x3){ //jal, jr
                struct control c = {0,1,0,0,0,0,0,0,0};
        }
        else if(opcode == 0x23){ //lw
                struct control c = {0,0,0,1,1,0x20,0,1,1};
        }
        else if(opcode == 0x2b){ //sw
                struct control c = {0,0,0,0,0,0x20,1,1,0};
        }
```

```
        else if(opcode == 0x4 || opcode == 0x5){ //beq, bne
                struct control c = {0,0,1,0,0,opcode,0,0,0};
        }
        else{ //I_type
                struct control c = {0,0,0,0,0,opcode,0,1,1};
        }
```

- 구조체를 만들고 다음과 같은 방법으로 signal들을 초기화해줘서 type별로 if문마다 각각의 signal변수들을 하나씩 초기화 해줬을 때보다 코드가 훨씬 줄어들었다.

```
 none


instruction 13174-----------PC : 0xcdd4-------------
[F] 0x00000000
 none


instruction 13175-----------PC : 0xcdd8-------------
[F] 0x00000000
 none


instruction 13176-----------PC : 0^C
```

- 하지만 출력화면에서 이처럼 무한루프가 돌게 되었다.

```
                else{ //I_type
                        struct control c = {0,0,0,0,0,opcode,0,1,1};
                        printf("%d,%d,%d,%d,%d,%d,%d,%d,%d",c.RegDst,c.Jump,c.Bran
ch,c.MemRead,c.MemtoReg,c.ALUOp,c.MemWrite,c.ALUSrc,c.RegWrite);
                }
                printf("%d,%d,%d,%d,%d,%d,%d,%d,%d",c.RegDst,c.Jump,c.Branch,c.Mem
Read,c.MemtoReg,c.ALUOp,c.MemWrite,c.ALUSrc,c.RegWrite);
```

- 여러 방법을 시도하며 해결책을 모색하던 중 명령어를 100개까지만 나오게 설정하고, 위처럼 if문의 안과 밖에서 구조체 내의 변수들 값을 각각 출력하게 해보았다.

```
instruction 1-----------PC : 0x0-------------
[F] 0x27bdfff8
[D] opcode: 0x9, rs: 29, rt: 29, rd: 31, shamt:31, func: 0x38
        simm 0xfffffff8 (-8)
0,0,0,0,0,9,0,1,1
0,0,0,0,0,0,0,0,0
```

- 출력 결과, If문 내에서는 원하는 대로 변수들의 값이 잘 설정되었지만 if문을 빠져나오면 변수들의 값이 다시 0으로 초기화되었다. 인터넷을 열심히 찾아보고 책도 찾아봤지만 특별한 이유와 해결책은 찾지 못하였다.

- 원래의 코드보다 훨씬 정돈되어졌고 보기 편해졌지만 제대로 작동이 되지 않아, 원래 코드로 되돌아올 수밖에 없었다. 하지만 여전히 불필요하게 긴 코드가 마음에 들지 않았고 결국 이 부분을 함수로 만드는 방법을 택했다. 현재로서는 최선의 방법이었지만, 원하던

코드가 아니었던 지라 이에 대해선 아직도 아쉬운 마음이 있다.

B. Minor Error 1

- 코드를 원래는 각각의 datapath에 맞게 짰었다. 모든 코드를 나름의 방법대로 구현한 후 실행을 시켰을 때 simple3까지는 올바르게 실행이 됐지만, simple4는 결과값이 틀리고 fib과 gcd는 무한루프가 되는 오류가 있었다. 아예 안되는 게 아니라, 몇 예제 파일만 안되는 것이어서 각각 파일의 mips.asm을 비교해보았다.

<simple 3의 mips.asm>

```
00000000 <foo>:
   0:   27bdffe8        addiu
   4:   afbe0014        sw
   8:   03a0f021        move
   c:   afc00008        sw
  10:   afc0000c        sw
  14:   afc00008        sw
  18:   08000011        j
  1c:   00000000        nop
  20:   8fc3000c        lw
  24:   8fc20008        lw
  28:   00000000        nop
  2c:   00621021        addu
  30:   afc2000c        sw
  34:   8fc20008        lw
  38:   00000000        nop
  3c:   24420001        addiu
  40:   afc20008        sw
  44:   8fc20008        lw
  48:   00000000        nop
  4c:   28420065        slti
  50:   1440fff3        bnez
  54:   00000000        nop
  58:   8fc2000c        lw
  5c:   03c0e821        move
  60:   8fbe0014        lw
  64:   27bd0018        addiu
  68:   03e00008        jr
  6c:   00000000        nop
```

<simple 4의 mips.asm>

```
00000034 <foo>:
  34:   27bdffd8        addiu
  38:   afbf0024        sw
  3c:   afbe0020        sw
  40:   03a0f021        move
  44:   afc40028        sw
  48:   8fc30028        lw
  4c:   24020001        li
  50:   14620004        bne
  54:   00000000        nop
  58:   24020001        li
  5c:   08000025        j
  60:   00000000        nop
  64:   8fc20028        lw
  68:   00000000        nop
  6c:   2442ffff        addiu
  70:   00402021        move
  74:   0c000000        jal
  78:   00000000        nop
  7c:   00401821        move
  80:   8fc20028        lw
  84:   00000000        nop
  88:   00621021        addu
  8c:   afc20018        sw
  90:   8fc20018        lw
  94:   03c0e821        move
  98:   8fbf0024        lw
  9c:   8fbe0020        lw
  a0:   27bd0028        addiu
```

- 가장 큰 차이점은 branch의 유무였다. 따라서 branch 부분만 계속 집중해서 보고 고쳤었다. 여러 시도들로 무한루프가 더 이상 돌지 않는다는 등의 진전은 있었지만, 여전히 이상은 있었다.

```
//write back
//update register values
if(opcode == 0){
        Regs[rd] = v3;
        printf("[W] R[%d] = 0x%x\n", rd, v3);
}
else if(opcode == 0x23){ //lw
        Regs[rt] = Read;
        printf("[W] R[%d] = 0x%x\n", rt, Read);
}
else if(opcode != 0x2b) {
        Regs[rt] = v3;
        printf("[W] R[%d] = 0x%x\n", rt, v3);
}
```

- 그 이유는 이렇다. Branch의 오류가 맞긴 했지만, 오류는 생각한 부분이 아닌 Write Back에 있었다. WB에서 branch는 WB의 과정을 거치지 않으므로 따로 예외처리를 해주었어야 했는데, sw일 때만 예외처리를 했었다.

- 현재 코드에서는 jump와 branch의 위치가 WB 후에 위치해 상관이 없지만, 기존 코드에서는 jump와 branch가 WB 전에 위치해 실행 후 continue를 해 남은 while문의 코드에 의해 변동이 일어나지 않도록 했다. 하지만 branch의 조건이 충족되지 않았을 때는 continue를 따로 설정하지 않았었다. 그래서 branch가 'not taken'될 때마다 명령어에 따른 R[rt]의 값이 바뀌어서 오류가 생겼던 것이다.

- continue를 통해 branch부분 이후에는 이미 오류가 날 걸 막았다고 생각했다. 따라서 branch부분만 계속 확인하고 고쳤던 것이 오랫동안 오류를 발견하지 못한 원인이 되었다.

C. Minor Error 2

```
instruction 130-----------PC : 0x204-------------
[F] 0xafc20110
[D] opcode: 0x2b, rs: 30, rt: 2, rd: 0, shamt:4, func: 0x10
        simm 0x00000110 (272)
[E] result(0xf64a8) = R[30](0xf6398) + simm(0x110)
[M] Write Memory
        M[0xf64a8] <= R[2](0x39d)

instruction 131-----------PC : 0x208-------------
[F] 0x000005fa
[D] opcode: 0x0, rs: 0, rt: 0, rd: 0, shamt:23, func: 0x3a
        simm 0x000005fa (1530)
        wrong insrtuction
```

- 기존 코드를 control logic으로 바꾸는 과정에서 다른 예제파일은 다 실행이 됐는데, input4.bin파일만 제대로 실행이 되지 않았다. 131번째 instruction에서의 명령어가 잘못돼 (wrong instruction 출력) 프로그램이 끝나는 것이다. 다행히 원래 제대로 돌아가던 출력값을 파일로 저장해 둬서 두 출력의 차이점을 찾을 수 있었다.

- 130번째 명령어까지는 PC값과 명령어 모두 같았지만, 131번째에는 명령어가 0x24021bac -> 0x0000005fa로 아예 달라졌다. Memory[0x208/4]에 0x5fa가 들어있어서

그럴 것이라고 판단을 하고 파일 내에서 0x5fa를 검색해보았다.

```
instruction 58-----------PC : 0xe4-------------
[F] 0xafc20080
[D] opcode: 0x2b, rs: 30, rt: 2, rd: 0, shamt:2, func: 0x0
        simm 0x00000080 (128)
[E] result(0x82) = R[30](0x2) + simm(0x80)
[M] Write Memory
        M[0x82] <= R[2](0x5fa)
```

- 예상대로 M[0x208/4(=0x82)]에 0x5fa 값이 들어있었다. 원래였으면 Execution의 결과가 0xf64180이 돼 M[0xf6418]에 R[2]의 값인 0x5fa가 들어있어야 했다. 어느 부분에서 오류가 난 것인지 찾으니 고치기도 쉬웠다.

```
if(func == 0x0 || func == 0x2)
        v1 = shamt;
```

- 명령어가 sll과 srl일 때 예외처리를 해줬던 게 화근이었다. 다른 요소를 고려하지 않고 func만 생각해서 R-type을 제외한 다른 type들의 명령어의 function code도 0x00이나 0x2가 될 수 있다는 걸 예상하지 못했다.

```
if(opcode == 0 && (func == 0x0 || func == 0x2))
        v1 = shamt;
```

- 따라서 R-type의 경우에만 if문을 실행하도록 바꿔주었다. 이후에는 문제없이 실행되었다.

D. Factorial

```
int main()
{
        int fact_n = 4;
        int result;

        result = fact(fact_n);
}

int fact(int n)
{
        int result;
        if(n <= 1) return 1;
        else return n * fact(n-1);
}
```

- 나만의 예제파일을 만들어 잘 실행되는지 확인해보기 위해, factorial 4를 실행시키는 c파일을 만들어보았다.

- 입력 창에서 'mips-mti-linux-gnu-gcc -c fact.c'를 입력해 compile을 시켜주었고,

'mips-mti-linux-gnu-gcc -c fact.c -o fact.o'를 입력해 fact.o 파일을 만들어주었다. 그리고, 'mips-mti-linux-gnu-objcopy -O binary -j .text fact.c fact.bin'을 입력해 fact.bin 파일을 만들어 주었다.

- 이 후 fact.bin파일을 이용해 내 코드를 돌려보았다. 올바르게 작동되지 않았다.

```
daeun17@assam:~/test_prog$ mips-mti-linux-gnu-objdump -d fact.o

fact.o:     file format elf32-tradbigmips


Disassembly of section .text:

00000000 <main>:
   0:   27bdffd8        addiu   sp,sp,-40
   4:   afbf0024        sw      ra,36(sp)
   8:   afbe0020        sw      s8,32(sp)
   c:   03a0f025        move    s8,sp
  10:   24020004        li      v0,4
  14:   afc20018        sw      v0,24(s8)
  18:   8fc40018        lw      a0,24(s8)
  1c:   0c000000        jal     0 <main>
```

- 그 후에 'mips-mti-linux-gnu-objdump -d fact.o'를 입력해 파일의 어셈블리를 살펴보았다.

```
 8c:   70621002        mul     v0,v1,v0
```

- c파일에서 곱셈하는 과정이 있었기 때문에 mul이라는 명령어가 사용되었고, 프로그래밍을 하며 mul은 구현하지 않았었기에 제대로 작동을 하지 않았던 것이었다. bin파일을 만들어보았다는 것에 의의를 두었다.

(5) Personal Feeling

이번 single-cycle 프로젝트는 첫 프로젝트였던 simple calculator보다 난이도도 훨씬 높아졌고, 코드도 두 배 이상 길어졌다. 추가구현도 역시 매우 까다로워졌다. 강의를 듣고 green sheet의 operation을 코드로 옮길 때 까지만 해도 쉽게 모든 구현을 해낼 줄 알았다. 하지만 쉬운 부분은 딱 그만큼이었다. 코드를 구체화시킬수록 쉽게 해결되지 않겠다는 생각이 들었다. 코드를 깔끔히 구현하기 위해서는 각 type 별 datapath에 대해 완벽한 이해가 필요했다. 따라서 안 보고 그릴 수 있을 때까지 datapath를 계속 그려보았고, 이를 코드로 어떻게 구현해야 할지 적어보는 과정도 거쳤다. 사실 이 프로젝트 전 까지만 해도, 이런 과정없이 바로 코드를 짰었다. 굳이 이런 과정이 없어도, 코드를 짜는 데 큰 어려움을 느끼지 못 했기 때문이다. 바로 전 프로젝트인 simple calculator에서도 역시 코드를 짜기 전 어떻게 짜야 할지 상상해보기만 했지 따로 적거나 그림을 그려보거나 하지는 않았다. 하지만, 이번 프로젝트에서는 그냥 코드를 짜려니 막막했고, 진전이 없어 처음으로 어떤 식으로 짤지 틀을 그려보았다. 처음 해보는 과정이라 미숙했고, 정말 도움이

되는 게 맞을지 의심도 들었다. 하지만 그냥 코드를 짜는 것과 큰 틀을 생각해두고 그에 맞게 코드를 완성해 나가는 것은 매우 달랐다. 관련 개념의 이해도도 확실히 높아졌다.

물론 오류도 꽤 많았다. 400줄이 넘는 코드에서 잘못된 부분을 찾는 것은 결코 쉽지 않았다. 정말 사소한 오류들이지만 찾는 데는 최소 두시간이 걸렸고, 위에 따로 게시한 오류들은 minor error라고 쓰긴 했지만, 하룻동안 코드만 들여다보며 찾아내 고친 오류들이다. 친구들의 도움도 있었다. 나의 코드로 올바르게 돌아가는 예제파일이 있고, 친구의 코드로 올바르게 돌아가는 예제파일이 있어 서로 올바른 출력 값을 공유하며 어디가 잘못된 부분이었는지 찾아낼 수도 있었다. 또 자기가 미리 겪어본 오류들도 공유하며 같은 실수를 만들어내지 않도록 하기도 했다. 혼자 했다면 훨씬 오래 걸렸을 시간을 단축한 것이다. 오류들은 거의 사소한 실수들로 인한 것이었다. 스스로 책망하기도 했고, 이후엔 같은 오류를 범하지 않으려고 코드를 추가할 때마다 몇 번이고 확인하는 과정을 거쳤다. 오류가 발생한 후 그 오류를 찾아내는 것보다 꼼꼼히 확인하는 게 더 적은 시간이 들고 효율적일 것이라 생각했기 때문이다. 이후에는 정말 오류가 발생하는 횟수가 줄어들었고 같은 실수를 더이상 반복하지 않았다. 이런 깨달음은 앞으로 다른 코드를 작성할 때도 큰 도움이 될 것 같다.

앞서 말했듯이, 초반에는 코드를 구현하는데 큰 어려움없이 무난히 끝낼 수 있을 것이라 생각했고, 많은 시간을 투자하지는 않았었다. 이 결과는 프로젝트 기간인 3주 중 마지막 일주일을 강행군으로 만들게 되었다. 일주일동안 하루에 밥 먹는 시간을 제외하고는 계속 코딩만 한 것 같다. 여러 시행착오 끝에 결국 400줄이 넘는 코드를 완성시켰다. 앞으로 더 긴 코드도 작성하게 될 테이지만, 코드를 짜기 전 큰 틀을 짜며 코드의 처음부터 끝까지 스스로 완성시킨 건 이게 처음이라 더 기억에 남을 것 같다.

어쩌다 보니 아쉬운 점으로만 가득한 보고서가 된 것 같은데, 사실은 그렇지 않다. 이 아쉬운 점들을 다 뛰어넘을 수 있을 정도로 뿌듯함은 이루어 말할 수가 없다. 정말 많은 시간과 노력을 들인 코드이고, 정성을 담은 코드이다. 무엇보다 '스스로' 해냈다는 것이 가장 큰 뿌듯함으로 다가왔다. 저번 프로젝트까지만 하더라도 linux환경이 어색했고 불편했는데, 지금은 따로 putty 명령어들을 찾아보지 않고도 코드를 완성시켜 나갈 정도로 많이 친숙해졌다. 앞으로 두 개의 프로젝트가 더 남아있다. 여태까지 해낸 프로젝트들을 교훈삼아 다음 프로젝트에서는 같은 실수를 반복하지 않도록 할 것이다.