Computer Architecture and Mobile Process

Single Cycle

Project2

April 9, 2022

Left day 5

Mobile System Engineering

32184939

Heo ChanYong

싱글사이클이란?

싱글 사이클은 instruction 하나를 하나의 사이클에서 수행하는 것을 의미한다. 한 사이클이 끝나면 다음 instruction 을 수행을 한다. Instruction 의 format 에 따라 작동해야 하는 하드웨어가 달라진다. 그래서 여러가지 instruction 을 수행하려면 controller 로 하드웨어를 조절해줘야 한다. 물론 단일 type 의 instruction 을 다룬다고 가정을 하면 따로 controller 를 조절할 필요가 없는 경우가 생길 것이다. 물론 앞서 말한 가정은 instruction 이 복잡하지 않다는 가정이다. 하지만 type 에 따라 하드웨어가 필요하면 비용이 많이 들것이다. 이와 같은 이유로 하나의 하드웨어에 controller 를 두어 instruction 의 opcode 에 따라 controller 로 하드웨어의 작동을 조절해줄 수 있다.

싱글 사이클을 만들기 위해 필요한 생각들

싱글 사이클을 구성할 때 가장 먼저 생각했던 것들은 어떻게 instruction 을 읽어와 bit 값들을 하드웨어 어떻게 전달을 하는 것 하나와 control 을 어떻게 조절을 해서 instruction 의 opcode 나 func 에 맞게 연산을 할 수 있을까? 였습니다. 그래서 저는 instruction 을 메모리에서 가져와 어떤 instruction 인지와는 상관없이 opcode, rt, rs, rd, shamt, funct, address 를 나누어 주었습니다. 원래는 opcode 에 따라 변수들이 정해지지만 Datapath 를 보면 먼저 instruction 을 사용하여 모든 변수들을 저장하는 것을 보고 아이디어를 얻었다. 위와 같이 decode 를 하고 decode 에서 얻은 opcode 를 통해 control 을 조절할 수 있다. Opcode 를 통해 나온 control 로 각각의 하드웨어들을 선택적으로 동작 시킨다. 예를 들어 add 라고 가정을 하면 opcode 는 0 이 되어 controller 에서 RegWrite =1 RegDst=1 이 되고 나머지 controller 는 0 이 된다. 그래서 rd 레지스터에 reg[rt]+ reg[rs] 값을 저장을 한다. 물론 instruction 의 format 에 따라 크게 분류할 수 있겠지만 항상 예외들은 존재하기 마련이다. 그래서 DataPath 를 그려보고 어떻게 controller 를 조절하면 좋을까 생각을 많이 했다.

싱글 사이클 구현

싱글 사이클을 구현할 때 가장 먼저 하는 것은 bin 파일에서 binary 형식으로 instruction 을 읽어와 memory 에 저장을 하는 것이다. 여기서 조심해야하는 점은 mips-binary 파일은 거꾸로 저장이 되기 때문에 이것을 우리가 조절을 해야 한다. 예를 들어 instruction 이 원래는 0x27bd7ff0 인 경우 27 bd 7f f0 순서대로 받아야 하지만 mips-binary 의 특성상 f0 7f bd 27 로 저장이 되어있다는 것이다. 그래서 bin 파일을 받을 때 순서에 맞게 swap 을 해줘야 한다. 본인 같은 경우는 아래 사진과 같이 swap 을 해주었다.

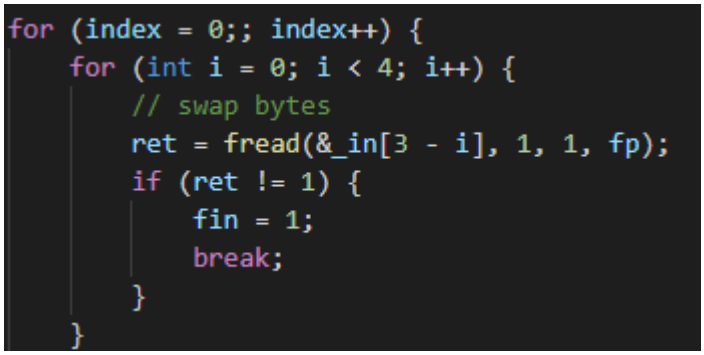


figure 1

위 사진과 같이 instruction 을 받으면 순서에 맡게 Memory 에 저장을 해준다. bin 파일을 다 읽고 우리가 만든 메모리 배열에 저장이 된다.

위 과정이 끝나면 진정한 single cycle 의 구현이 시작된다. Single cycle 의 구현을 위해서 필요한 모든 변수를 전역변수로 만들면 좋겠지만 예상치 못한 에러가 발생할 수도 있고 가시성에서 떨어질 수도 있다고 생각을 한다. 그래서 본인은 struct 형태로 필요한 변수들을 묶었다. 물론 모든 변수들을 struct 형식으로 만든 것은 아니다. 아래 사진과 같이 inst_, Control_, stat_ 3 개만 struct 로 만들어 주었다.

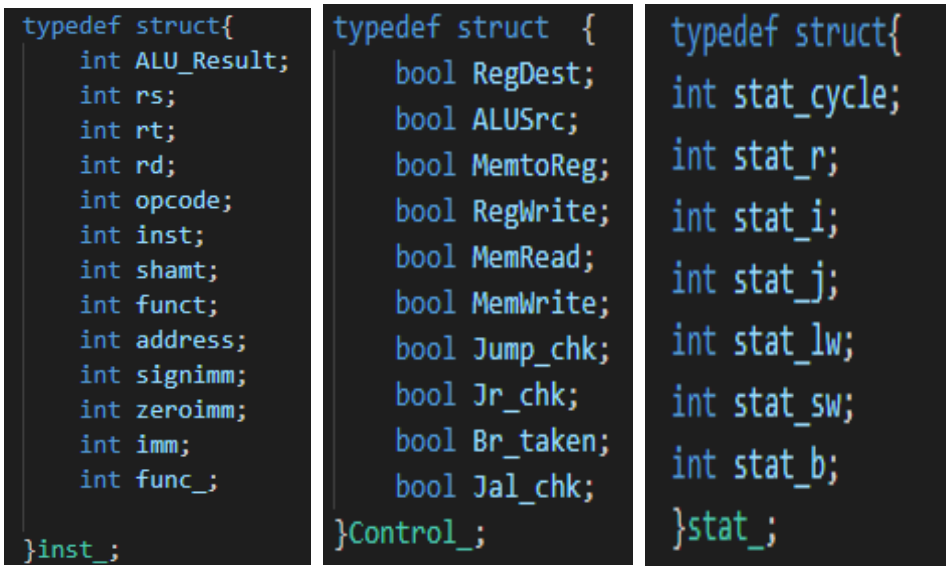


figure 2,3,4

본인이 생각하기에는 해당 변수들을 struct 로 나누어 준 이유는 관리가 편리하다고 생각을 하였기 때문이다. 해당 변수들을 초기화를 하고 싶으면 memset() 을 사용하여 원하는 값들로 코드 한 줄로 바꿀 수 있기 때문이다. 위 과정이 끝나면 instruction 을 저장한 메모리를 읽기 시작한다. 초기 pc 는 0 이고 fetch 를 통해 Memory 에서 instruction 을 읽어온다. 읽어온 instruction 을 decode 과정을 해준다. Decode 는 instruction 을 받아 inst_ 구조체의 변수들의 활용에 맞게 다 업데이트 해준다. 그렇게 decoding 을 하고 나서 업데이트 된 inst_ 구조체를 통해 control 을 조절을 해준다. 여기서 Control 은 매우 중요하다. Control 은 figure 5 와 같이 수행된다. 해당 Controle 함수는 opcode 를 통해 control 을 조절하는데 Control 구조체에 들어가 있는 controller 들을 조건에 맞게 bool 값으로 업데이트 해주었다. 교수님께서 업로드 하신 자료에는 jr_chk 나 Jal_chk 와 같은 controller 가 없었으나 해당 single cycle 을 구현할 때 필요하다고 생각이 되어 따로 controller 를 control 구조체에 추가를 하였다. 물론 jump 한다는 점은 같을 수 있으나 사용해야하는 기능들이 조금씩 다르기 때문에 이와 같은 결정을 하였다. 해당 로직에서 조심해야할 점들은 교수님의 자료에 있는 DataPath 나 Control 의 조건에는 구현이 안된 instruction 이 있기 때문에 구현하는 본인이 수정을 해야 하는 부분이 있다는 것을 인지를 해야 한다. 본인도 해당 부분을 인지를 하지 못하고 구현을 하다 무한루프에 빠져 하나씩 decoding 을 통해 문제점을 확인할 수 있었다.

```
void Control(inst_ *ins, Control_* c) {
    if (ins->opcode == 0) c->RegDest = true;
    else c->RegDest = false;
    if (ins->opcode != 0 && ins->opcode != beq && ins->opcode != bne) c->ALUSrc = true;
    else c->ALUSrc = false;
    if (ins->opcode == lw || ins->opcode == ll || ins->opcode == lbu || ins->opcode == lhu) c->MemtoReg = true;
    else c->MemtoReg = false;
    if (ins->opcode != sw && ins->opcode != bne && ins->opcode != beq && ins->opcode != j && ins->opcode != jr) c->RegWrite = true;
    else c->RegWrite = false;
    if (ins->opcode == lw) c->MemRead = true;
    else c->MemRead = false;
    if (ins->opcode == sw || ins->opcode == sh || ins->opcode == sb) c->MemWrite = true;
    else c->MemWrite = false;
    if (ins->opcode == j) c->Jump_chk = true;
    else c->Jump_chk = false;
    if (ins->opcode == beq ) c->Br_taken = true;
    else if (ins->opcode == bne ) c->Br_taken = true;
    else c->Br_taken = false;
    if(ins->opcode==0 && ins->func_==jr) c->Jr_chk=true;
    else c->Jr_chk=false;
    if (ins->opcode==jal) c->Jal_chk=true;
    else c->Jal_chk=false;
}
```

figure 5

controller 들을 업데이트 하고 나서 excute 함수로 이동을 한다. excute 같은 경우에는 control 구조체를 통해 연산이 적용이 된다. 가장 먼저 ALUSrc 가 false 이며 opcode 가 0 인 경우 funct 를 통해 연산이 이루어진다. 위와 같은 조건으로 나눈 이유는 beq 나 bne ALUSrc 가 False 인데 I 타입이므로 연산과정이 달라지기 때문이다. 특히나 beq 나 bne 같은 경우는 조건이 맞냐 틀리냐 에 따라 Br_taken 이 업데이트 되어야하기 때문에 조건을 나누어 주었다. 이제 ALUSrc 가 true 인 경우는 I type 이기 때문에 opcode 에 맞는 연산을 실행해준다. J 같은 경우 excute 가 필요 없기 때문에 해당 excute 함수에는 포함되지 않는다. 실행이 되고 나서 memory 에 저장하거나 레지스터에 저장을 해야 한다. 이를 수행하기 위해서 본인은 Memory_and_store 라는 함수를 만들었다. Memory_and_store 함수에서 확인하는 controller 는 Memwrite, MemRead, RegWrite, MemtoReg, Jump_chk 이다. 여기서 jump_chk 를 왜 확인을 하지라고 생각을 할 수 있다. 해당 부분은 본인의 실력이 부족하여 그럴 수도 있고 아니면 프로그래밍이다 보니 생기는 문제일 수 있다. 본인이 생각한 jump_chk 의 활용은 I type 의 instruction 을 다룰 때 문제가 생긴다. I type 은 jump 와 controller 가 비슷하다는 문제점이 있다. RegWrite=1, ALUSrc=1, RegDest=0, MemtoReg=0, MemRead=0 이기 때문에 jump_chk 를 통해서

구분을 해줘야 한다. 만약 위와 같이 jump_chk 를 넣어주지 않으면 엉뚱한 register 에 엉뚱한 값이 저장이 될 수 있기 때문이다. 위의 과정이 끝나면 pc 를 업데이트 해주어야한다. Pc 와 register 는 opcode 가 j, jal, jr 에 따라서 업데이트가 다르게 작동이 된다. j 같은 경우 jump_chk 라는 controller 를 통해서 확인을 하고 jump address 로 pc 를 업데이트 시킨다. jal 같은 경우는 jal_chk 라는 controller 로 reg[31]를 pc 값으로 업데이트 시키고 pc 를 jump address 값으로 업데이트 시킨다. jr 같은 경우는 jr_chk 라는 controller 로 확인을 하고 excute 함수를 통해 얻은 ALU_Result 값으로 pc 를 업데이트 시킨다. 마지막으로 BNE 나 BEQ 같은 경우 Br_taken controller 와 excute 에서 얻은 ALU_Result 값이 동일한지를 통해 pc 를 branch_address 로 업데이트 시켜준다. 위의 과정이 끝나면 한 사이클을 돌았다고 한다.

싱글 사이클 코드 실행 결과

Simple.bin 실행결과

```
Cycle : 7
[Fetch] pc[0x18] instruction 0x27BD0008
[Decode] opcode(09) rs: 1d rt: 1d imm: 0
[Excute] ADDIU ALU_Result= ( reg[29] + signimm(8) )   : 0x1000000
[PC Update] pc -> 0x1c

Cycle : 8
[Fetch] pc[0x1C] instruction 0x03E00008
[Decode] opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[PC Update] JR pc-> 0xffffffff

0

stat_cycle 8
stat_r 4
stat_i 4
stat_j 0
stat_lw 1
stat_sw 1
stat_b 0
```

figure 6

Simple2.bin 실행결과

```
Cycle : 9
[Fetch] pc[0x20] instruction 0x27BD0018
[Decode] opcode(09) rs: 1d rt: 1d imm: 0
[Excute] ADDIU ALU_Result= ( reg[29] + signimm(24) )   : 0x1000000
[PC Update] pc -> 0x24

Cycle : 10
[Fetch] pc[0x24] instruction 0x03E00008
[Decode] opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[PC Update] JR pc-> 0xffffffff

100

stat_cycle 10
stat_r 3
stat_i 7
stat_j 0
stat_lw 2
stat_sw 2
stat_b 0
```

figure 7

Simple3.bin 실행결과

```
Cycle : 1329
[Fetch] pc[0x64] instruction 0x27BD0018
[Decode] opcode(09) rs: 1d rt: 1d imm: 0
[Excute] ADDIU ALU_Result= ( reg[29] + signimm(24) )   : 0x1000000
[PC Update] pc -> 0x68

Cycle : 1330
[Fetch] pc[0x68] instruction 0x03E00008
[Decode] opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[PC Update] JR pc-> 0xffffffff

5050

stat_cycle 1330
stat_r 409
stat_i 920
stat_j 1
stat_lw 407
stat_sw 206
stat_b 102
```

figure 8

Simple4 실행결과

```
Cycle : 242
[Fetch] pc[0x28] instruction 0x27BD0020
[Decode] opcode(09) rs: 1d rt: 1d imm: 0
[Excute] ADDIU ALU_Result= ( reg[29] + signimm(32) )   : 0x1000000
[PC Update] pc -> 0x2c

Cycle : 243
[Fetch] pc[0x2C] instruction 0x03E00008
[Decode] opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[PC Update] JR pc-> 0xffffffff

55

stat_cycle 243
stat_r 79
stat_i 153
stat_j 11
stat_lw 59
stat_sw 41
stat_b 10
```

figure 9

gcd.bin 실행결과

```
Cycle : 1060
[Fetch] pc[0x40] instruction 0x27BD0030
[Decode] opcode(09) rs: 1d rt: 1d imm: 0
[Excute] ADDIU ALU_Result= ( reg[29] + signimm(48) )   : 0x1000000
[PC Update] pc -> 0x44

Cycle : 1061
[Fetch] pc[0x44] instruction 0x03E00008
[Decode] opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[PC Update] JR pc-> 0xffffffff

1

stat_cycle 1061
stat_r 359
stat_i 637
stat_j 65
stat_lw 333
stat_sw 153
stat_b 73
```

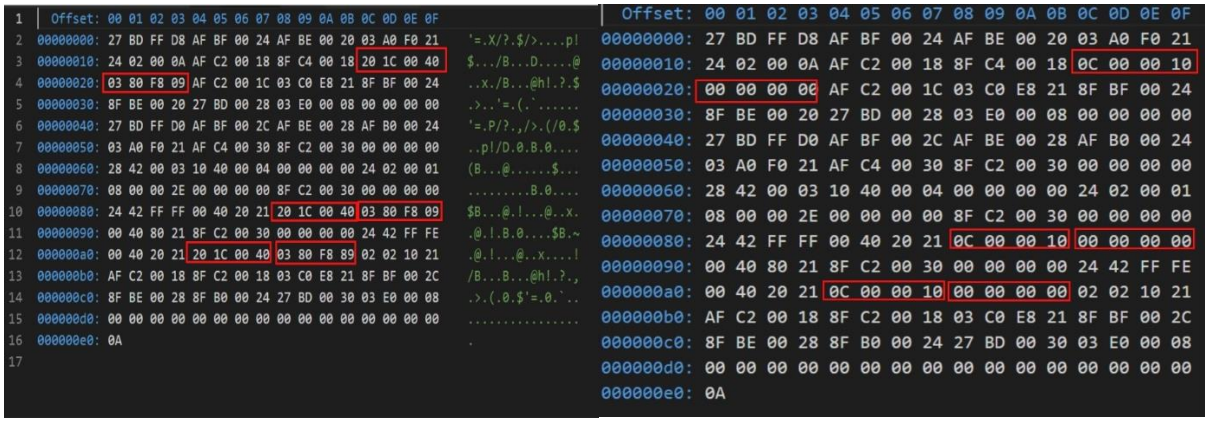
figure 10

fib.bin 실행결과

```
Cycle : 2678
[Fetch] pc[0x34] instruction 0x27BD0028
[Decode] opcode(09) rs: 1d rt: 1d imm: 0
[Excute] ADDIU ALU_Result= ( reg[29] + signimm(40) )   : 0x1000000
[PC Update] pc -> 0x38

Cycle : 2679
[Fetch] pc[0x38] instruction 0x03E00008
[Decode] opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[PC Update] JR pc-> 0xffffffff

55

stat_cycle 2679
stat_r 818
stat_i 1697
stat_j 164
stat_lw 601
stat_sw 494
stat_b 109
```

figure 11

Fib2.bin



```
Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 27 BD FF D8 AF BF 00 24 AF BE 00 20 03 A0 F0 21   '=.X/?.$/>....p!
00000010: 24 02 00 0A AF C2 00 18 8F C4 00 18 20 1C 00 40   $.../B...D.... .@
00000020: 03 80 F8 09 AF C2 00 1C 03 C0 E8 21 8F BF 00 24   ..x./B...@h!.?.$
00000030: 8F BE 00 20 27 BD 00 28 03 E0 00 08 00 00 00 00   .>..'=.(.`......
00000040: 27 BD FF D0 AF BF 00 2C AF BE 00 28 AF B0 00 24   '=.P/?.,/>.(/0.$
00000050: 03 A0 F0 21 AF C4 00 30 8F C2 00 30 00 00 00 00   ..p!/D.0.B.0....
00000060: 28 42 00 03 10 40 00 04 00 00 00 00 24 02 00 01   (B...@......$...
00000070: 08 00 00 2E 00 00 00 00 8F C2 00 30 00 00 00 00   .........B.0....
00000080: 24 42 FF FF 00 40 20 21 20 1C 00 40 03 80 F8 09   $B...@.!...@..x.
00000090: 00 40 80 21 8F C2 00 30 00 00 00 00 24 42 FF FE   .@.!.B.0....$B.~
000000a0: 00 40 20 21 20 1C 00 40 03 80 F8 89 02 02 10 21   .@.!...@..x....!
000000b0: AF C2 00 18 8F C2 00 18 03 C0 E8 21 8F BF 00 2C   /B...B...@h!.?.,
000000c0: 8F BE 00 28 8F B0 00 24 27 BD 00 30 03 E0 00 08   .>.(.0.$'=.0.`..
000000d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000000e0: 0A                                                .
```

```
Offset: 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00000000: 27 BD FF D8 AF BF 00 24 AF BE 00 20 03 A0 F0 21
00000010: 24 02 00 0A AF C2 00 18 8F C4 00 18 0C 00 00 10
00000020: 00 00 00 00 AF C2 00 1C 03 C0 E8 21 8F BF 00 24
00000030: 8F BE 00 20 27 BD 00 28 03 E0 00 08 00 00 00 00
00000040: 27 BD FF D0 AF BF 00 2C AF BE 00 28 AF B0 00 24
00000050: 03 A0 F0 21 AF C4 00 30 8F C2 00 30 00 00 00 00
00000060: 28 42 00 03 10 40 00 04 00 00 00 00 24 02 00 01
00000070: 08 00 00 2E 00 00 00 00 8F C2 00 30 00 00 00 00
00000080: 24 42 FF FF 00 40 20 21 0C 00 00 10 00 00 00 00
00000090: 00 40 80 21 8F C2 00 30 00 00 00 00 24 42 FF FE
000000a0: 00 40 20 21 0C 00 00 10 00 00 00 00 02 02 10 21
000000b0: AF C2 00 18 8F C2 00 18 03 C0 E8 21 8F BF 00 2C
000000c0: 8F BE 00 28 8F B0 00 24 27 BD 00 30 03 E0 00 08
000000d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000e0: 0A
```

```
Cycle : 2787
[Fetch] pc[0x34] instruction 0x27BD0028
[Decode] opcode(09) rs: 1d rt: 1d imm: 0
[Excute] ADDIU ALU_Result= ( reg[29] + signimm(40) )   : 0x1000000
[PC Update] pc -> 0x38

Cycle : 2788
[Fetch] pc[0x38] instruction 0x03E00008
[Decode] opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[PC Update] JR pc-> 0xffffffff

55

stat_cycle 2788
stat_r 927
stat_i 1806
stat_j 55
stat_lw 601
stat_sw 494
stat_b 109
```

figure 12 13 14

Optional JAL 을 JALR 로 고쳐서 하는 추가 문제가 있었다. JAL 을 어떻게 JALR 로 고칠 수 있을까 생각을 하다가 가장 먼저 생각했던 것은 어떤 레지스터에 다음 주소값을 집어넣어야 할까였다. 만약에 사용하는 레지스터에 값을 집어넣게 되면 연산이 문제가 생길 수 있다고 판단을 하였기 때문이다. 그래서 mips 코드를 보았고 28 번 레지스터는 사용을 하지 않는다고 판단을 하였다. 그러면 레지스터에 주소 값을 어떻게 집어넣어야 할까? 를 다음으로 생각을 하였다. Addiu 를 사용해서 0 번 zero 레지스터를 이용해서 주소 값을 넣어야 겠다고 판단을 하였다. 그렇게 28 번 레지스터에 0x40 이라는 값을 집어넣었다. 0x40 이라는 값은 괜히 나온 값이 아닌 fib 함수의 첫 주소를 의미하였고

다음 jalr 이 가야하는 target address 를 의미하기 때문이다. Addiu 를 통해 r28 에 주소값을 삽입을 하고 다음 0x0380f889 를 통해서 다음 값으로 넘겨주었다. Opcode 는 0 이고 rs=28, rt= 0, rd =31 func 는 9 이다. 여기서 jalr 은 28 번 레지스터 값을 31 번 레지스터에 저장을 해준다. 그렇게 JAL 역할을 2 가지 instruction 으로 나누어 수행을 하도록 하였다. 실제 JALR 은 점프이기 때문에 pc 를 업데이트 하는 부분에서 조절을 해주었다. 해당 코드에서 조심해야하는 부분은 reg[ins->rd]=pc 라고 하였는데 해당 pc 는 현재 pc 에 4 를 더한 값이다. 원래라면 jump 를 하거나 JAL 을 할 때는 원래 pc 에 +8 을 하는 것이 정상적이나 다음 수행해야하는 pc 를 오직 +4 만 한 것은 원래 jal 뒤에는 nop 이라는 0x0000 0000 이라는 명령어가 있어 뛰어넘어야 했지만 본인은 nop 자리에 jalr 을 넣었기 때문에 현재 pc 에 +4 만 하였다. Jalr 은 jal 명령어를 두개로 나누다 보니 cycle 을 더 많이 돈 것을 확인할 수 있다.

```
}else if(ins->opcode==0 && ins->func_==jalr){
    reg[ins->rd]=pc;
    pc=reg[ins->rs];
    printf("JALR");
    stat->stat_r++;
}
```

figure 15

input4.bin 실행결과

```
Cycle : 23372705
[Fetch] pc[0x18EEC] instruction 0x27BD7FF0
[Decode] opcode(09) rs: 1d rt: 1d imm: 0
[Excute] ADDIU ALU_Result= ( reg[29] + signimm(32752) )   : 0x1000000
[PC Update] pc -> 0x18ef0

Cycle : 23372706
[Fetch] pc[0x18EF0] instruction 0x03E00008
[Decode] opcode(00) rs: 1f rt: 0 rd: 0 shamt: 0 funct(8)
[PC Update] JR pc-> 0xffffffff

85

stat_cycle 23372706
stat_r 10152862
stat_i 13219741
stat_j 103
stat_lw 6090476
stat_sw 1026130
stat_b 2029699
```

figure 16

해당 파일들을 전부 실행을 하였을 때 R, I, J instruction 을 수행했을 때 나오는 횟수의 합과 총 cycle 의 횟수는 같아야한다. 현재 코드는 nop 이라는 instruction 을 R 타입 instruction 이라고 가정을 하였다. 그래서 위와 같은 결과들이 나왔고 합도 cycle 의 횟수와 같게 나오고 결과값도 예상값과 똑같게 나와서 single cycle 을 알맞게 구현했다고 생각을 한다. 여기서 주목해야하는 점은 만약에 single cycle 을 최적화 시킨다면 어떨까라는 생각을 하였다. 물론 본인이 잘못 생각을 하고 있을수도 있지만 한 사이클을 돌 때 강의 자료에서 처럼 600ps 가 걸린다고 가정을 해보자. Gcd.bin 을 실행 결과를 참고하여 GCD.bin 의 실행시간은 636600 ps 가 걸리고 만약 아래 그림처럼 필요한 하드웨어만 들어갔다고 가정을 해보자.

R-type	200	50	100		50	400
I-type	200	50	100		50	400
LW	200	50	100	200	50	600
SW	200	50	100	200		550
Branch	200	50	100			350
Jump	200					200

figure 17

R 은 359*400-> 143600, I 는 400*78-> 31200, Lw 는 333*600 -> 199800, Sw 550*153 -> 84150, br 73*350 ->25550 총 484300ps 가 걸린다. 이렇게 최적화를 하면 약 30% 정도 cost 를 덜 사용할 수 있지 않을까를 생각했다. 물론 아직 배움이 짧다보니 해결책은 더 생각을 해보아야겠다.

해당 프로젝트를 하고 느낀점

프로젝트를 진행하기 전에는 겁이 많이 났다. 내가 올바르게 single cycle 이라는 개념을 이해하고 Data Path 에 대하여 올바르게 이해를 하였는지 확신도 안 섰기 때문이다. 해당 프로젝트를 Data Path 도 안 그려보고 시작을 하였으나 바로 장애물에 부딪히게 되었다. Data Path 를 보면 우리가 어떻게 하드웨어를 조절할지 하드웨어가 어떤 기능을 하는지에 대해서 알 수가 있다. 이러한 부분들을 하나씩 이해하고 공부를 하니 어떤 방식으로 접근을 해야 하고 어떤 함수들을 만들어 싱글사이클을 구현해야 할지 이해하였다.

해당 프로젝트를 하면서 막혀서 힘들었던 부분들을 몇 가지 서술하겠다.

첫번째는 구조체에 대해서 잘못 이해하고 있었던 점이다. 구조체를 선언을 하면 포인터로 관리를 하겠지 했으나 값들이 제대로 업데이트가 되지 않았고 그에 따라 pc 나 다른 구조체나 구조체의 변수들이 제대로 업데이트가 되지 않았다. 디버깅하면서 문제점들을 찾아 잘못 알고 있던 부분들을 수정하였다.

두번째는 Memory_and_Store() 함수의 조건이었다. Instruction 에 맞게 Control 이 업데이가 되고 그에 맞게 Memory_and_Store()이라는 함수가 작성되었어야 했다. 그래서 Memory_and_store 함수에 들어가는 조건들을 조절을 했어야 했다. 하지만 수업 자료에 있던 Data Path 를 맹신하여 코드를 짜서 문제가 발생하였다. 분명 교수님이 수정하셔야 합니다. 라고 말씀을 하셨으나 해당 내용들을 잊고 코드를 짰었다. 문제가 생겼던 부분은 I type 부분이었다. Jump 를 할 때도 해당 부분이 실행이 되어 건드릴 필요 없는 register 에 이상한 값들이 저장이 되어 무한루프가 도는 경우가 생겼었다. 이를 찾기 위해서 디버깅을 끊임없이 했어야 했다. 하지만 시작 전에 교수님의 말씀을 기억했다고 해도 아마 디버깅을 하면서 열심히 찾지 않았을까 생각이 든다.

해당 프로젝트를 하면서 많은 것들을 느꼈다. 그래도 코드는 어느정도 짠다고 생각했던 내 자신이 정말 부끄러웠다. 물론 실력이 많이 부족하다고 생각은 하였으나

평균정도는 될 것이라고 생각을 하였으나 누가 보기에는 간단한 코드도 며칠을 투자하여 끝낼 정도로 실력이 형편없다고 느꼈다. 뿐만 아니라 코드를 짜기 전에 어떻게 짤 것인지 구상을 하는 것도 무척 중요하다고 느꼈다. 무턱대고 생각나는 대로 작성하다 보면 많은 예상치도 못한 문제들이 발생하여 디버깅을 하더라도 찾지 못하는 경우가 생길 수 있다는 것을 느꼈다.