Project #1

<MIPS Single Cycle CPU Implementation>

컴퓨터구조실험

학번: 2021202045

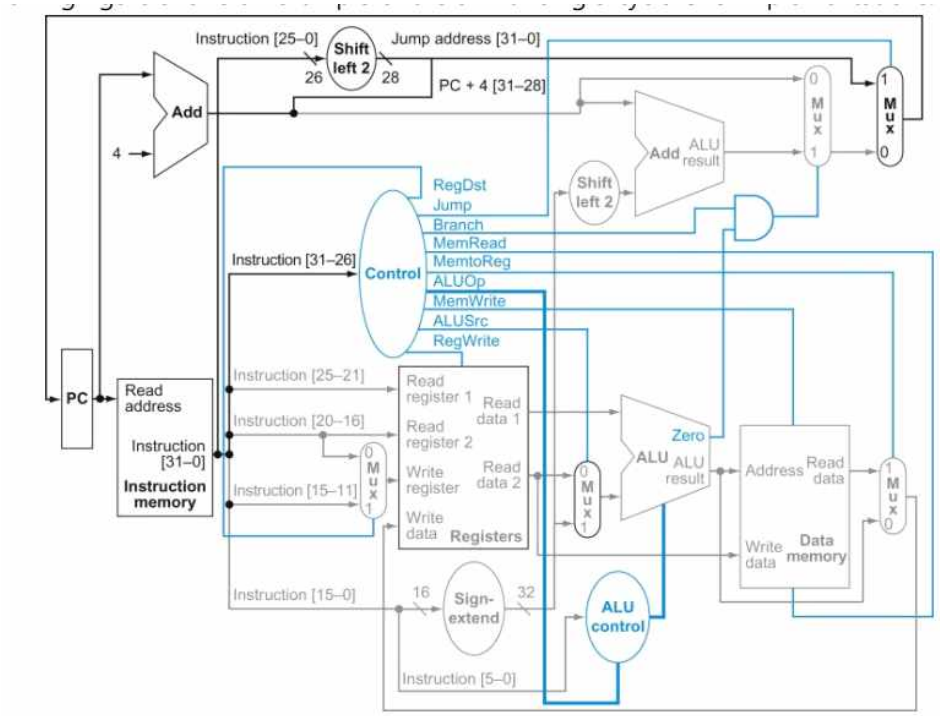
이름: 김예은

담당 교수님: 이성원 교수님

분반: 컴구실 (수) 분반

Ⅰ. 명령어 설명 및 명령어별 하드웨어 구현

다음은 이번에 구현해야할 single cycle cpu 블록도이다. 이 블록도를 바탕으로 본인이 구현한 명령어 control을 설명하고자한다.



1. R-type

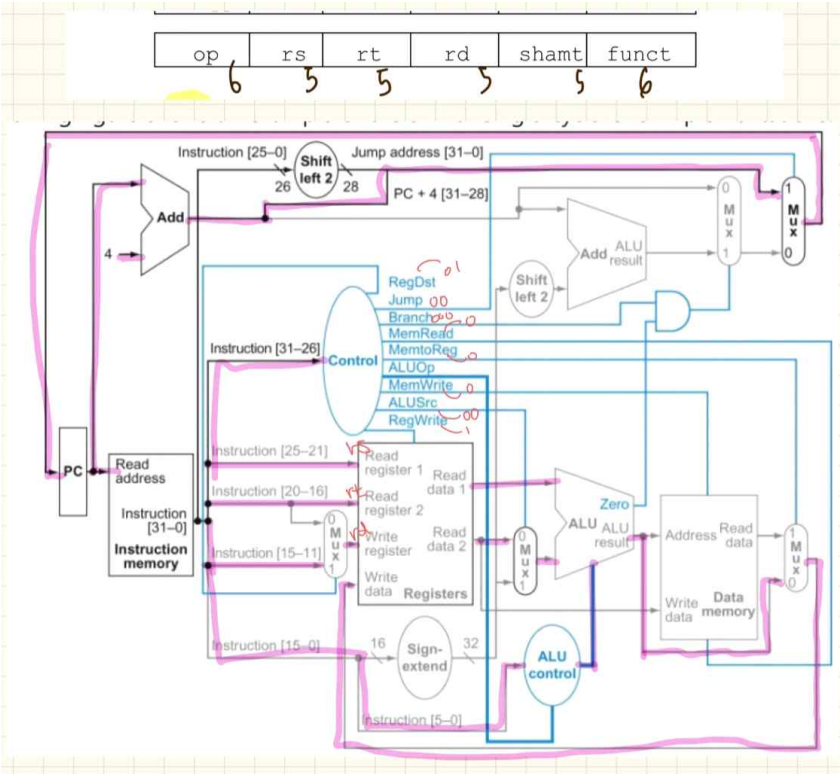
R-type은 opcode가 000000으로 통일되어 있고, function field에 각 operation에 맞는 function을 적어 control하는 방식이다.

-addu: rd = rs + rt로, unsigned add역할을 수행하는 명령어이다.

-or: rd = rs | rt, rt와 rs의 각 비트를 or연산하는 것이다. or란, 비교 연산하는 비트 중에 하나만 1이어도 1을 출력하는 것이다.

-sll: rd = rt < < a, rt의 값을 a(shamt field)값만큼 left shift하는 연산을 수행한다.

-srav: rd = rt > > > rs, rt의 값을 rs 값만큼 sign bit right shift하는 연산을 수행한다.



R-type의 경우,

rd 레지스터에 rt와 rs의 ALU연산 결과값을 쓰므로, regwrite=1, memwrite=0으로 해줘야한다. rs와 rt의 값을 읽어 ALU input으로 넣어주고, data memory를 거치지 않고 다시 register file로 와서 wirte register인 rd에 들어가 data를 write해주는 data path를 가진다. 또한, pc의 경우 다음 명령어를 가리키기 위해 pc+4를 해준 값이 들어간다.

다음은 각 명령어에 맞는 function값이다.

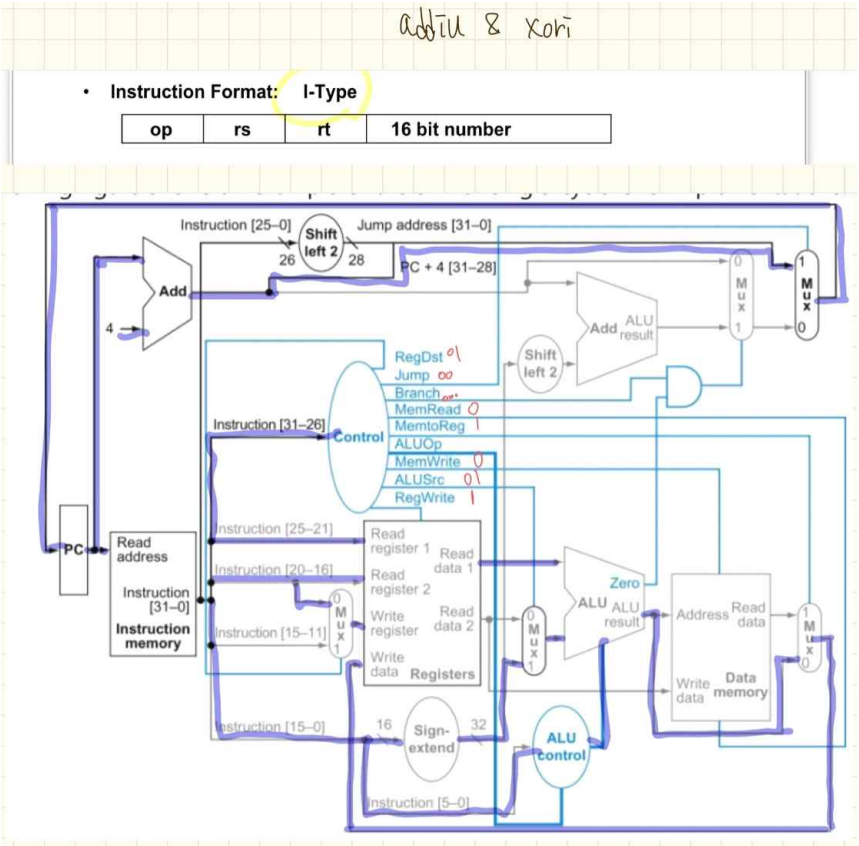
addu	100001
or	100101
sll	000000
srav	000011

2. I-type

I type의 경우 lw, sw, branch가 있는데, rd 레지스터 대신 rt 레지스터가 쓰이고, 16-bits immediate값이 쓰인다는 것이 특징이다.

-addiu : addu의 rt값 대신 rs와 sign extended하여 32bit가 된 immediate값을 unsigned add연산을 해준 후, 결과 값을 rt 레지스터 값에 저장해준다.

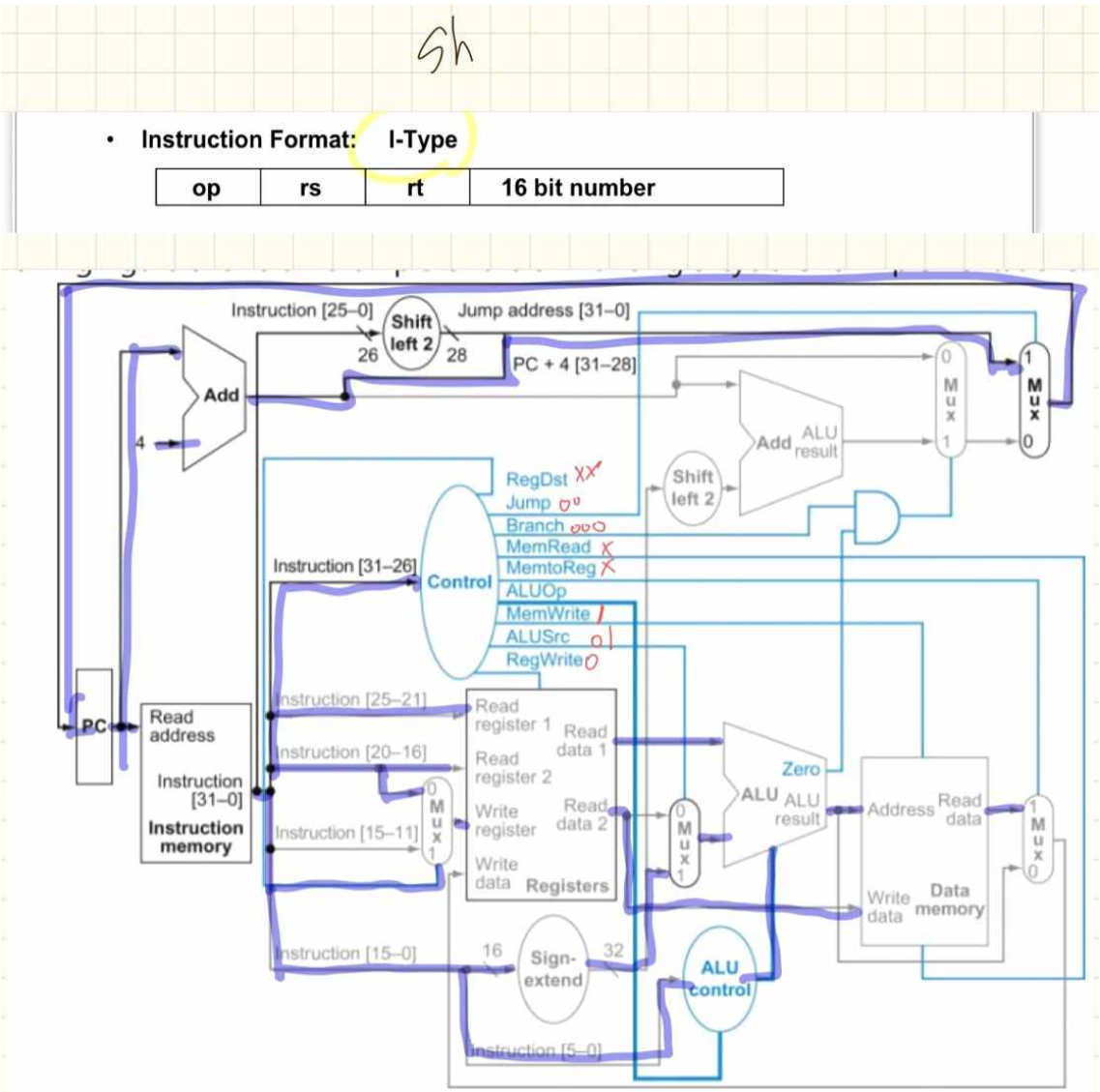
-xori : rs값과 zero extended한 imm값을 xori해준 후, rt값에 넣어준다. xor연산의 경우 1이 홀수 개 있을 때, 1을 반환하는 연산이다.



write register에 rd값 대신 rt값이 들어가고, rs값과 imm값이 ALU의 input으로 들어가서 알맞은 연산을 한다. ALU의 연산 결과 값은 Data Memory를 거치지 않고 RF로 들어가서 rt 값에 쓰인다. pc의 경우 pc+4를 해준다. data memory에 접근하지 않으므로, memwirte=0, regwirte=1이 된다.

명령어	opcode
addiu	001001
xori	001110

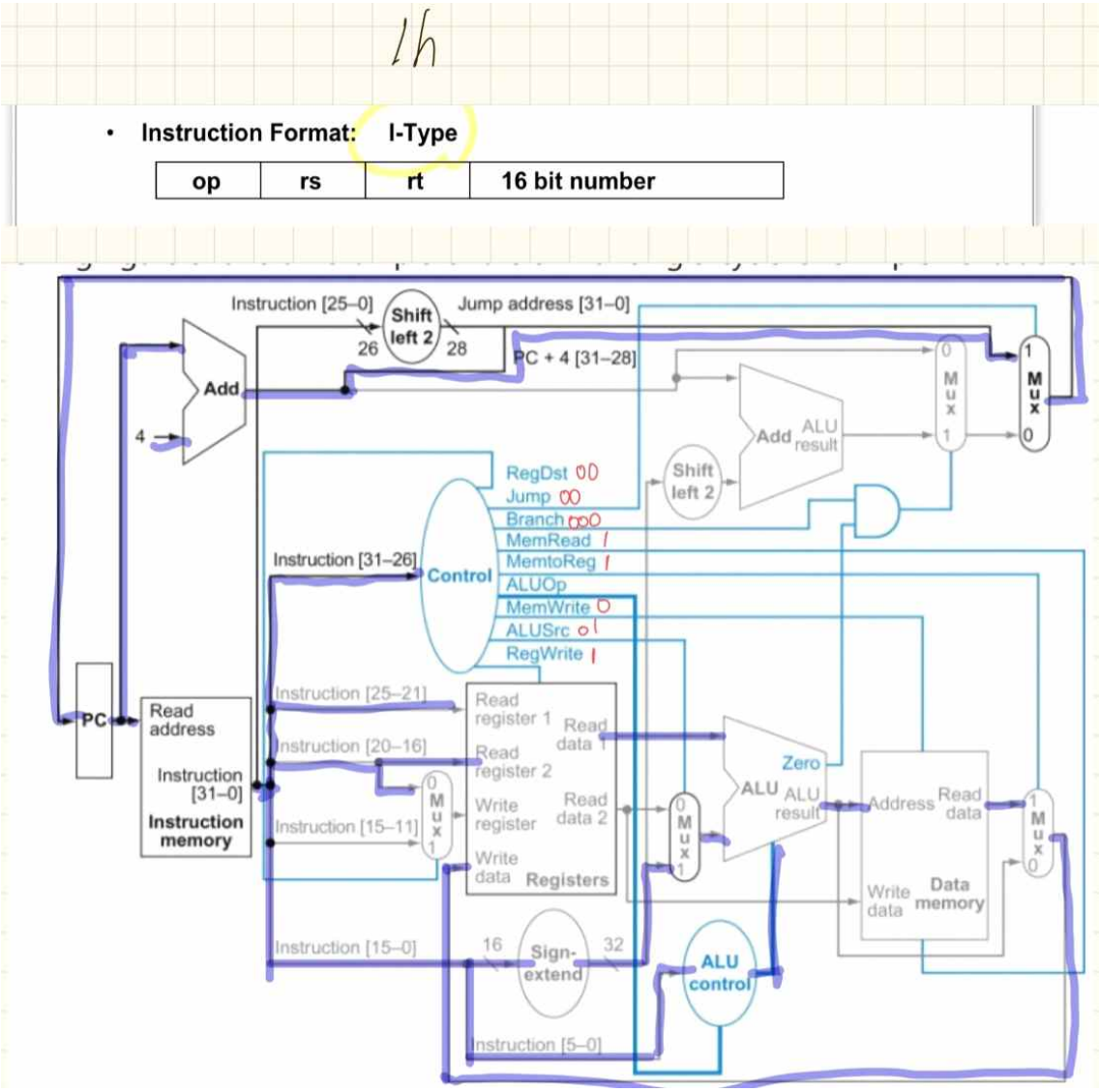
-sh: MEM [$s + i]:2 = LH ($t), rs와 imm값을 더한 값을 메모리 주소로 지정하고 그 주소의 2바이트(16비트)부분에 rt레지스터의 2바이트 데이터를 저장한다는 뜻이다.



sh의 경우 data memory에 쓰는 것이므로, memwrite=1, regwrite=0이다. alu에서 계산된 주소값으로 memory에 접근하여, rt(read data2)부분을 해당 메모리주소에 저장해준다. pc = pc+4를 해준다.

명령어	opcode
sh	101001

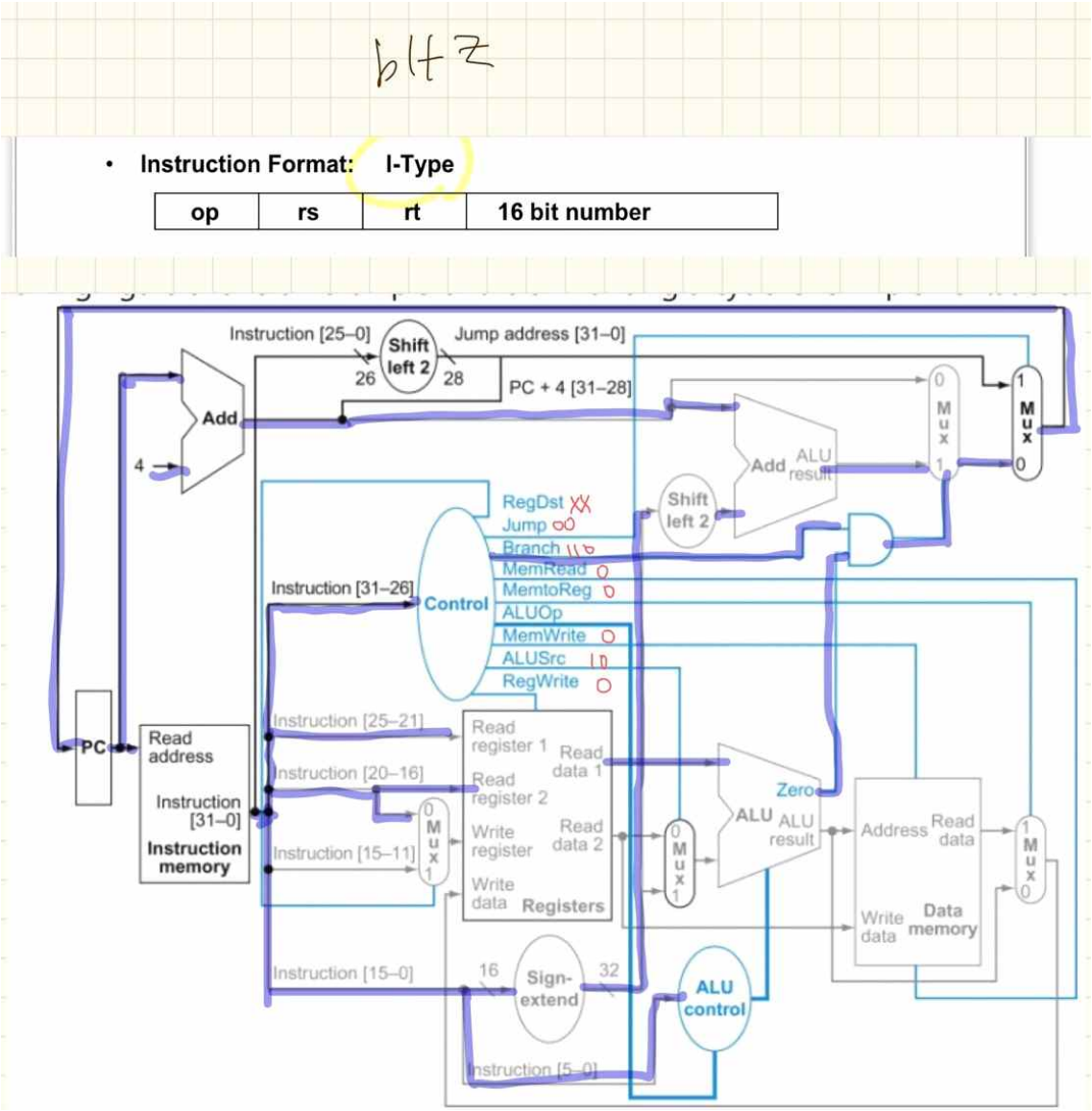
-lh: $t = SE (MEM [$s + i]:2), rs와 imm값을 더한 값의 메모리주소에서 값을 읽어와 2바이트만 읽고 그 값을 sign extended하여 rt 레지스터에 적는다.



data memory에서 값을 읽어 rt 레지스터에 적어야하므로 regwrite=1, memwrite=0이다. rs와 imm값을 더하는 것을 ALU에서 해준 다음 그 결과값이 Data memory의 address부분으로 들어간다. pc=pc+4이다.

명령어	opcode
lh	100001

-bltz: rs레지스터값이 0보다 작다면 branch를 해주는 명령어로, 작다면 1을 반환하는데 이 flag를 and gate를 통해 mux의 select로 간다. branch할 때의 pc 주소값은 pc = pc+4+(imm<<2)값으로 계산한다.



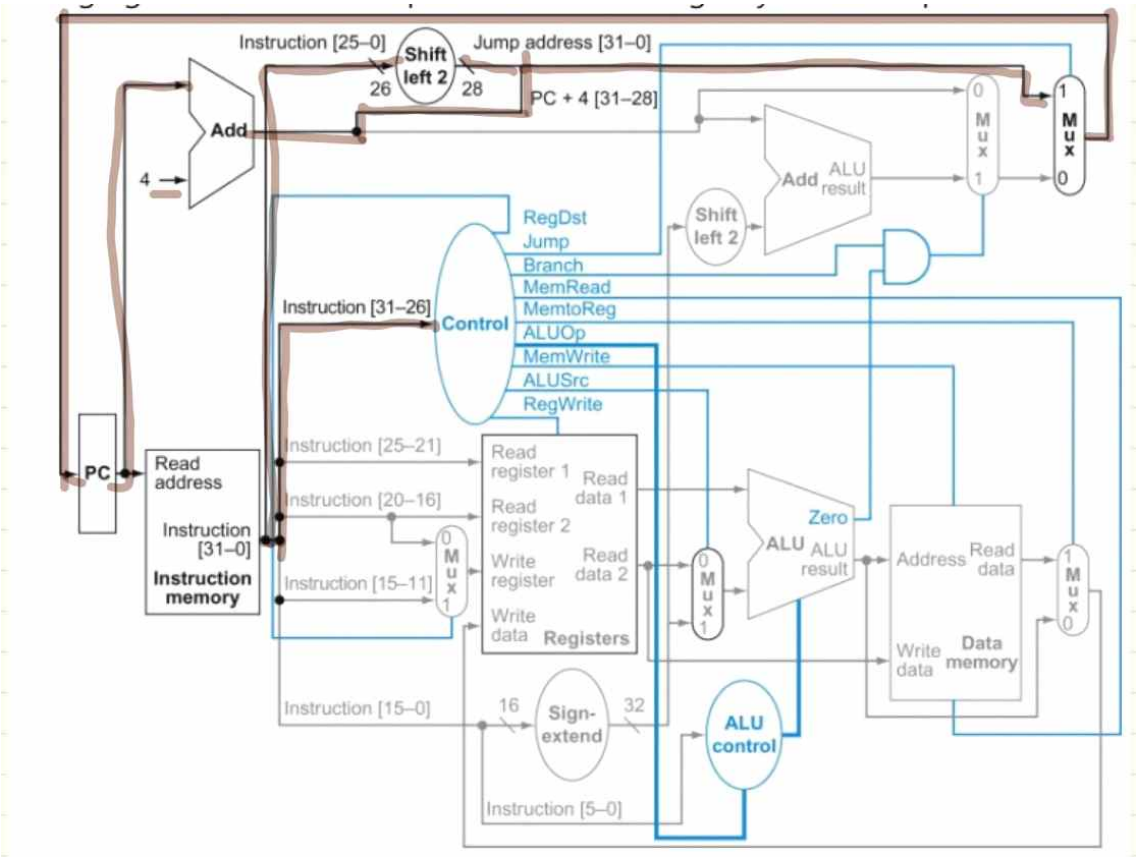
RF나 Data memory에 쓸 일이 없기 때문에 write 신호를 둘다 0으로 만들어준다. rs의 값이 zero보다 작을 때, branch를 해주는데, 그 때 pc값은 pc = pc+4+imm<<2값이므로, 위와 같은 datapath를 가진다.

명령어	opcode
bltz	(00000)

3. J-type

j type은 opcode 6 bits와 imm값 26 bits를 가진다. 주로 jump할 때 쓰이는 type이다. imm값이 26bits이므로, branch보다 큰 범위로 jump가 가능하다.

-jal: jump는 무조건으로, pc의 주소값 계산을 다음과 같은 data path를 가지면서 한다.



imm값 26bits를 shift left2를 하여 맨 뒤 부분이 00으로 되게 한 다음 이 값을 pc+4[31:28] 값을 띄어서 붙여 새로운 pc값을 만들어 저장한다.

명령어	opcode
jal	000011

Ⅱ. testbench

(1)

001111_00000_00010_0001_0010_0011_0100

001101_00010_00011_0101_0110_0111_1000 //$3에 0x12345678저장

001111_00000_00100_0001_0001_0010_0010

001101_00100_00101_0011_0011_0100_0100 //$5에 0x11223344

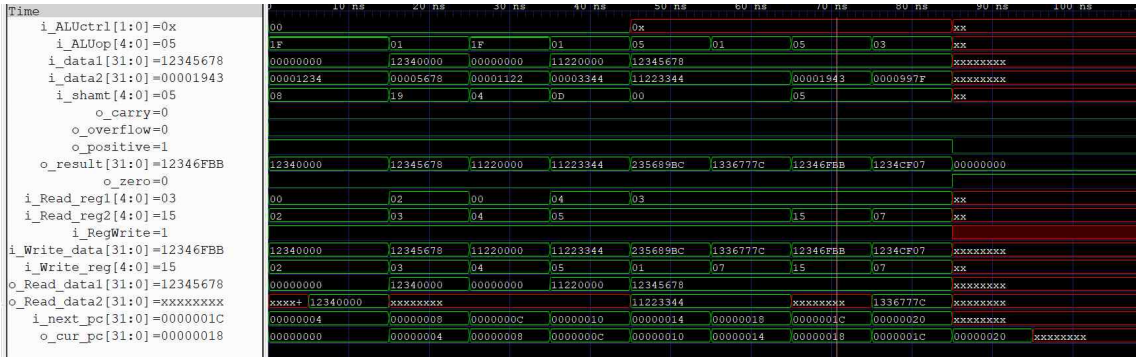
000000_00011_00101_00001_00000_100001//addu

000000_00011_00101_00111_00000_100101 //or

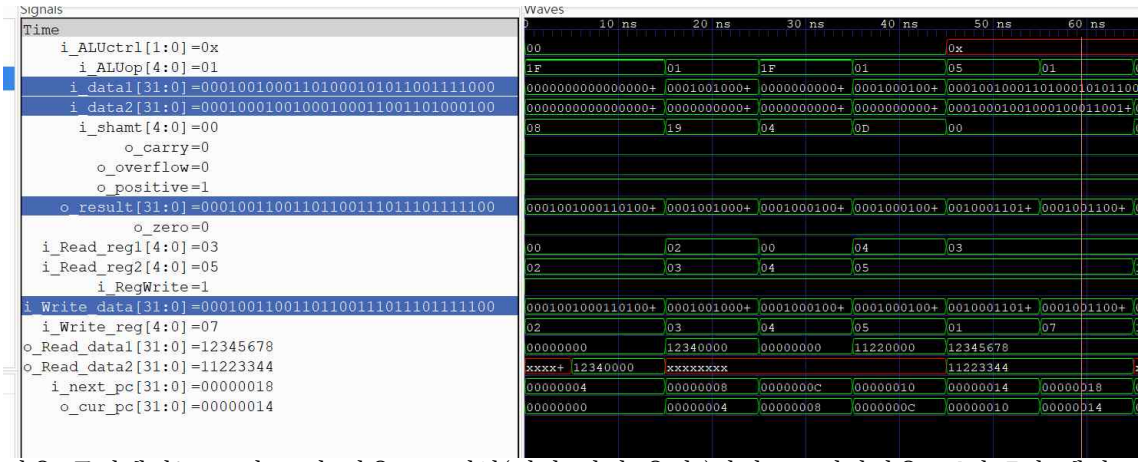
001001_00011_10101_00011_00101_000011//addiu

001110_00011_00111_10011_00101_111111//xori

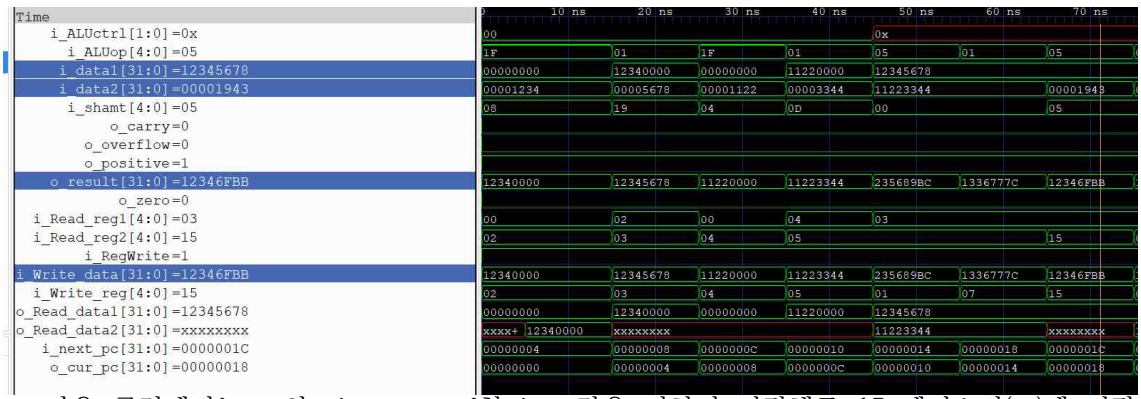
에 대한 testbench 결과 사진은 다음과 같다.



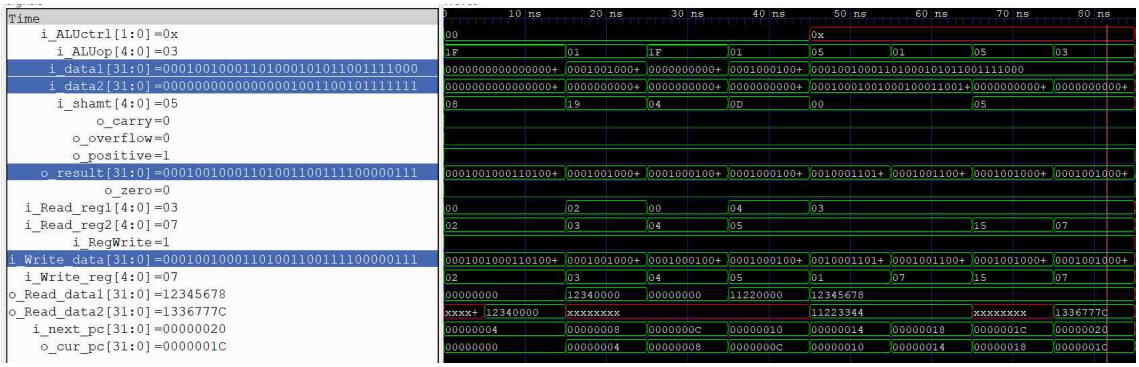
우선 레지스터 3번과 레지스터 5번에 각각 0x12345678, 0x11223344을 잘 저장한 것을 확인 할 수 있다. 그 다음 이 둘의 값을 더하여 결과값이 0x235689BC값이 잘 나왔다. 이 값을 rd인 1번 레지스터에 잘 쓴 것을 확인 할 수 있고, pc=pct+4를 해주었다.



다음 클럭에서는 rs와 rt의 값을 or 연산(결과 값이 옳다.)하여 그 결과값을 rd인 7번 레지스터에 넣어준 것을 확인 할 수있다. pc값이 pc+4를 해주었다.



그 다음 클럭에서는 rs와 sign extend한 imm값을 더하여 지정해준 15 레지스터(rt)에 저장해준다. pc= pc+4이다.



마지막으로 xori연산을 해주는데 rs와 imm값에서 1의 값이 홀수 개로 있을 때만 o_result에 1로 반환된다. 이 값이 지정해준 7번 레지스터에 잘 저장되는 것을 확인할 수있다. pc=pc+4 이다.

(2)

001111_00000_00010_0000_0000_0000_0000

001101_00010_00011_0000_0000_0000_0001 //$3에 0x00000001저장

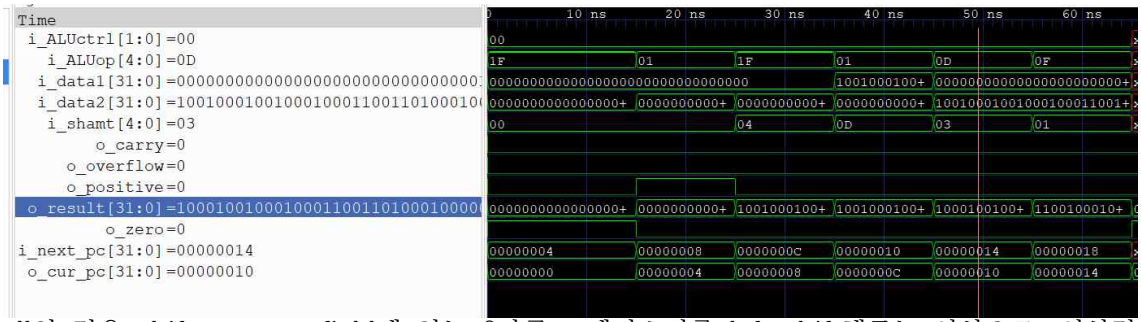
001111_00000_00100_1001_0001_0010_0010

001101_00100_00101_0011_0011_0100_0100 //$5에 0x91223344

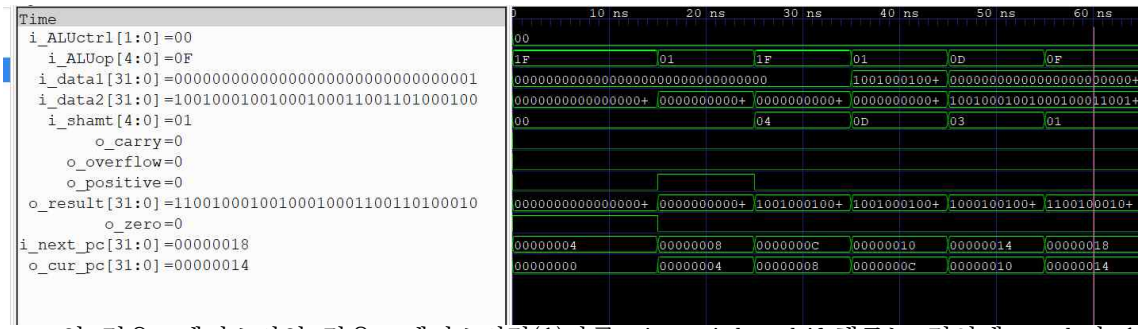
000000_00011_00101_00011_00011_000000 //sll

000000_00011_00101_11001_00001_000111//srav $d = $t >>> $s

위에 대한 testbench는 다음과 같다.



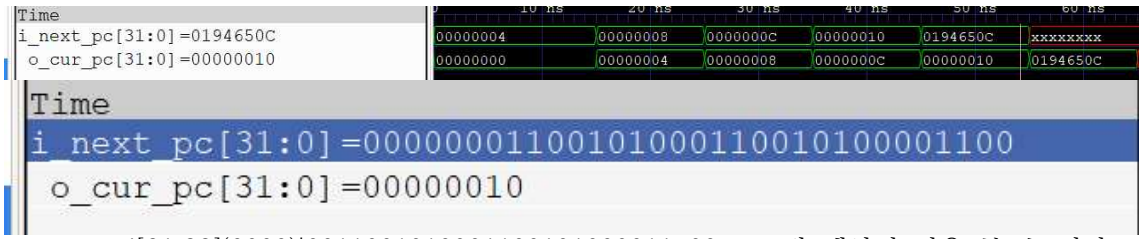
sll의 경우 shift amount field에 있는 3만큼 t 레지스터를 left shift해주는 연산으로, 연산결과 값이 잘 나온다. pc=pc+4이다.



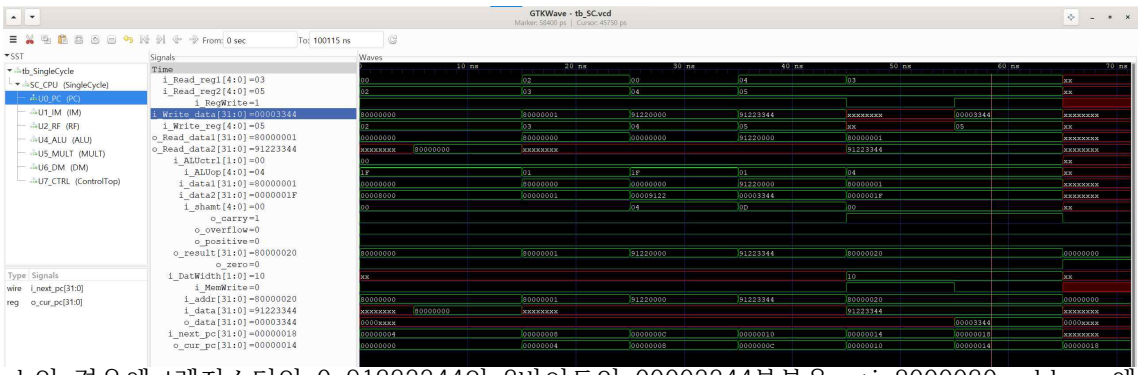
srav의 경우 t레지스터의 값을 s레지스터값(1)만큼 sign right shift해주는 것인데, msb가 1인 t 레지스터값이 오른쪽으로 한칸 움직이면서 msb가 sign에 따라 1로 나타났다. pc=pc+4

이다.
(3)
001111_00000_00010_0000_0000_0000_0000
001101_00010_00011_0000_0000_0000_0001 //$3에 0x00000001저장
001111_00000_00100_1001_0001_0010_0010
001101_00100_00101_0011_0011_0100_0100 //$5에 0x91223344

000011_00011_00101_00011_00101_000011 //jal
에 대한 testbench는 다음과 같다. alu는 쓰이지 않는 연산이므로 alu부분은 캡처하지 않았다.



pc = pc+4[31:28](0000)|00110010100011001010000011+00으로 잘 계산된 것을 볼 수 있다.
(4)
001111_00000_00010_1000_0000_0000_0000
001101_00010_00011_0000_0000_0000_0001 //$3에 0x80000001저장
001111_00000_00100_1001_0001_0010_0010
001101_00100_00101_0011_0011_0100_0100 //$5에 0x91223344
101001_00011_00101_00000_00000_011111//sh MEM[$s+i]:2 = LH($t)
100001_00011_00101_00000_00000_011111//lh $t = SE (MEM [$s + i]:2)
에 대한 testbench는 다음과 같다.



sh의 경우에 t레지스터의 0x91223344의 2바이트인 00003344부분을 s+i=8000020 address에 접근하여 그 메모리의 2바이트 부분에 00003344를 저장한다. pc = pc+4이다. lh의 경우에 s+i=8000020 address의 2바이트부분을 sign extended한 값인 00003344(아까 sh를 통해 저장해준 값)값을 t레지스터인 5번 레지스터에 쓴다. pc=pc+4이다.

(4)

001111_00000_00010_1000_0000_0000_0000

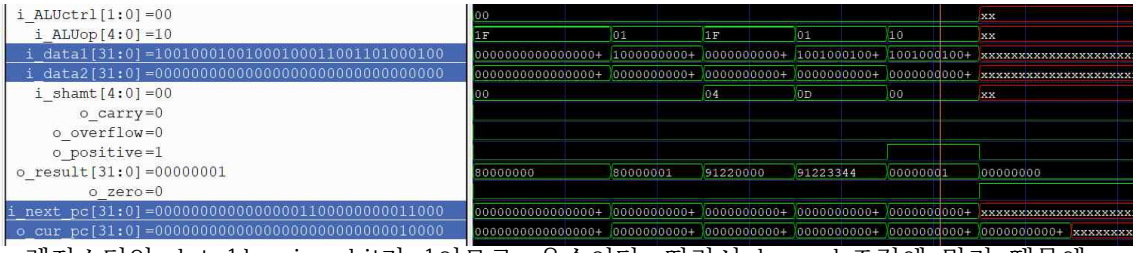
001101_00010_00011_0000_0000_0000_0001 //$3에 0x80000001저장

001111_00000_00100_1001_0001_0010_0010

001101_00100_00101_0011_0011_0100_0100 //$5에 0x91223344

000001_00101_00000_00110_00000_000001 //bltz if ($s < 0) pc += i << 2

에 대한 testbench는 다음과 같다.



rs레지스터인 data1는 sign bit가 1이므로, 음수이다. 따라서 branch조건에 맞기 때문에 pc값 주소를 계산해준다. pc = pc+4+(imm<<2)이므로, 00000110000000000000100+00000000000000000000000000010100를 더하면 next_pc값이 잘 계산되는 것을 볼 수 있다.

(5)

001111_00000_00010_1000_0000_0000_0000

001101_00010_00011_0000_0000_0000_0001 //$3에 0x80000001저장

001111_00000_00100_1001_0001_0010_0010

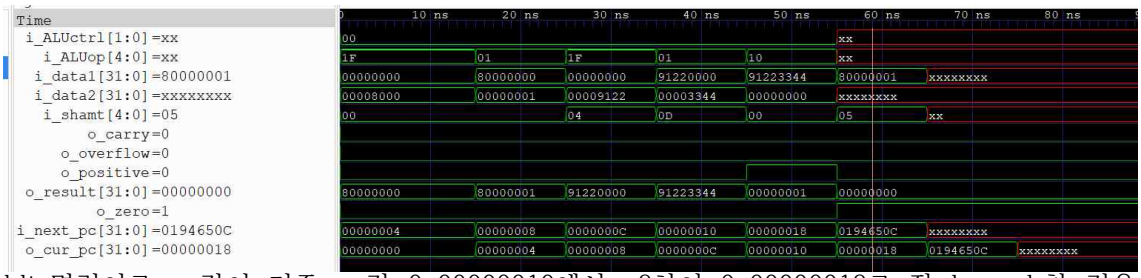
001101_00100_00101_0011_0011_0100_0100 //$5에 0x91223344

000001_00101_00000_00000_00000_000001 //bltz if ($s < 0) pc += i << 2

001111_00000_00010_0000_0000_0000_0000 //그냥 넣어준 명령어

000011_00011_00101_00011_00101_000011 //jal

에 대한 testbench는 다음과 같다.



bltz명령어로 pc값이 기존 pc값 0x00000010에서 +8하여 0x00000018로 잘 branch한 것을 확인할 수있다. 그리고 해당 명령어 주소값에 있는 jal을 잘 실행한다.

Ⅲ. 구현한 PLA AND 및 PLA OR

(1) PLA AND

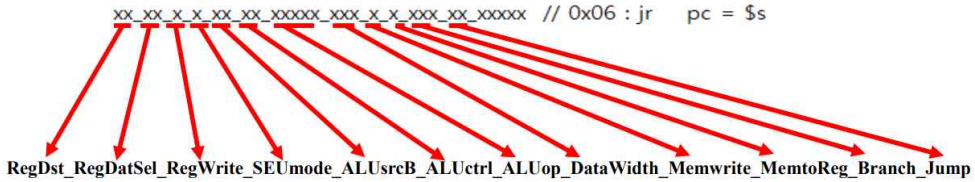
:어떤 명령어인지 분석해주는 decoder같은 역할

Opcode_Function_Regimmediate

000000_000000_xxxxx // 0x00 : sll

000000_000111_xxxxx // 0x05 : srav
000000_100001_xxxxx // 0x12 addu
000000_100101_xxxxx // 0x16 or
000011_xxxxxx_xxxxx // 0x1e jal
001001_xxxxxx_xxxxx // 0x24 addiu
001110_xxxxxx_xxxxx // 0x29 xori
xxxxxx_xxxxxx_00000 // 0x1b bltz
100001_xxxxxx_xxxxx // 0x2c lh
101001_xxxxxx_xxxxx // 0x31 sh

(2) PLA OR
:각 명령어에 따라 control을 해주는 역할



01_00_1_x_00_00_01101_xxx_0_0_000_00_xxxxx // 0x00 : sll $d = $t << a
01_00_1_x_00_00_01111_xxx_0_0_000_00_xxxxx // 0x05 : srav $d = $t >>> $s
01_00_1_x_00_0x_00101_xxx_0_0_000_00_xxxxx // 0x12 addu $d = $s + $t
01_00_1_x_00_0x_00001_xxx_0_0_000_00_xxxxx // 0x16 or $d = $s | $t
xx_xx_0_x_10_00_10000_xxx_0_0_011_00_xxxxx // 0x1b bltz if ($s < 0) pc += i << 2
10_11_1_0_xx_xx_xxxxx_xxx_0_0_000_01_xxxxx // 0x1e jal $31 = pc; pc = pc4 | i26 << 2
00_00_1_0_01_0x_00101_xxx_0_0_000_00_xxxxx // 0x24 addiu $t = $s + SE(i)
00_00_1_0_01_0x_00011_xxx_0_0_000_00_xxxxx // 0x29 xori $t = $s ^ ZE(i)
00_00_1_1_01_00_00100_110_0_1_000_00_xxxxx // 0x2c lh $t = SE (MEM [$s + i]:2)
xx_xx_0_1_01_00_00100_010_1_x_000_00_xxxxx // 0x31 sh MEM [$s + i]:2 = LH ($t)

Ⅳ.고찰

PLA AND 와 PLA OR에 대해서 감을 못잡아서 강의자료를 참고하여 이해했다. PLA AND를 decoder와 같은 역할로 생각하니 이해가 수월하게 되었다. bltz 명령어부분이 헷갈려서 계속 구현을 못했었다. 왜냐하면 branch부분 control이 branch if not positive인줄 알았는데, if not zero로 고쳐주니 testbench가 잘 나왔다.또한, bltz부분의 opcode는 명시적으로 나와있지않아서 돈케어로 뒀었는데, datasheet 표를 보고 Regimm의 경우 opcode를 000001로 둔다는 것을 파악하여 opcode를 작성해주니 잘 작동하였다. testbench를 맨처음 볼 때는 헷갈렸지만 single cycle이다보니 한 cycle 내에 모든 연산 과정 및 결과가 한 클럭 내에 나타나

서 보기 편했다. jal을 쓰면 내가 준 imm값 26bit에 따라 pc값이 이동하는데 그 뒤로 명령어를 읽지 않아 뭐지했는데 해당 pc값에 명령어를 txt파일에 적어줘야 작동할 수 있음을 깨달았다. 그래서 branch를 통해 jal이 있는 주소값으로 건너뛰고 해당 명령어를 잘 실행시키는지도 추가하여 확인해보았다.