Project #2

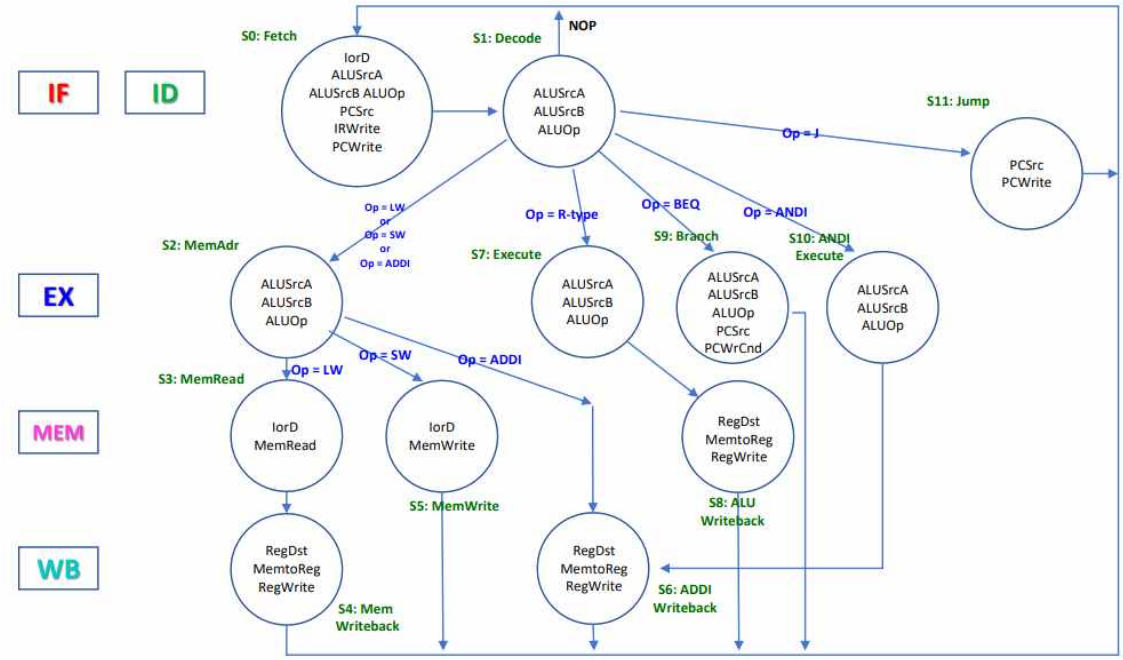
<MIPS Multi-Cycle CPU Implementation>

컴퓨터구조실험

학번: 2021202045
이름: 김예은
담당 교수님: 이성원 교수님
분반: 컴구실 (수) 분반

Ⅰ. 문제의 해석

-실험 내용에 대한 설명



Muti-cycle이란, single cycle과 달리 latency가 가장 긴 명령어에 맞춰 cycle을 조정하지 않아도 되고, ALU나 MEM을 여러 번 배치하지 않아도 되는 장점이 있다. 1 cycle에 1 명령어를 실행하던 single cycle이 아니라 state에 따라 동작하는 특징을 가지고 있는데 위의 사진이 state를 따라 path를 그린 FSM이다. 우선 FETCH, DECODE하는 동작은 각각 state 0, state 1로 모든 명령어에 공통적으로 들어가는 과정이다. FETCH의 경우 IR에 PC에 저장된 값을 옮기고, PC=PC+4를 먼저 해주어 ALU를 사용한다. 왜냐하면 EXEC부분에서 ALU를 사용할 것이므로 FETCH단계에서 먼저 사용해주는 것이다. DECODE 단계에서는 각각 알맞은 레지스터값을 읽어 A,B에 저장한 후, branch를 할 명령어인지 아닌지 모르지만, 우선 먼저 branch에 대한 명령어 주소를 ALU를 사용하여 계산해준다. 그 다음은 EXEC단계로, 이제부터 어떤 명령어인지에 따라 STATE를 골라서 간다. 위의 경우, OP값에 따라 STATE가 5갈래로 나뉘어져 가는 것을 확인할 수 있다. 우선 LW,SW의 경우, EXEC에서 접근할 메모리 주소를 계산하기 위해 ALU를 사용한다. LW의 경우, ALU결과값 주소로 가서 DATA를 가져오는 MEM STATE->해당 값을 RF에 write back하는 WB단계가 있다. SW의 경우, ALU결과값 주소로 가서 값을 적은 후 다시 STATE 0번으로 간다. R-TYPE의 경우, EXEC에서 ALUop에 따라 rs,rt의 값을 연산한 후 WB에서 ALU result값을 rd에 write back해준다. Branch의 경우, EXEC에서 조건문을 계산한 후, 조건문에 맞으면 DECODE단계에서 계산해놓은 branch 주소로 건너뛴다. jump의 경우 EXEC단계에서 jump 주소를 계산하여 jump한다.

-문제점 해결 방향

위의 FSM을 분석하여, state를 나누어 해당 state에서 어떤 동작을 하면 되는지(예를 들어, ALUop 및 RF write enable control) control 신호를 맞추어 on/off시키면서 설계하면 되겠다고 생각한다. 또한, ROM_MICRO.txt에서 어느 state값에 control 신호를 저장했는 지에 따라 ROM_DISP.txt를 적으면 된다.

Ⅱ. Micro-instruction

Micro-instruction은 컴퓨터의 CPU 내부에서 실행되는 작은 명령어로, CPU가 각각의 기능을 수행하는 방법을 결정한다. 밑의 field는 micro-instruction의 실행을 제어할 때, cpu에서 실행되는 작업을 세분화하고, 명확하게 정의함으로써 cpu가 명령어를 올바르게 수행할 수 있도록 돕는다.

field	용도
Opcode	수행될 명령어의 종류를 지정
Operand	명령어에서 사용되는 데이터나 메모리 주소를 지정한다.
Addressing mode	operand 필드에서 사용되는 주소 지정 모드를 결정한다. CPU는 이 필드를 참조하여 데이터나 메모리 주소를 어떻게 계산해야 하는지 결정한다.
Condition code	수행된 명령어의 결과에 따라 플래그를 설정하고, CPU는 이 필드를 참조하여 다음 명령어가 실행될지 여부를 결정한다.
Control signal	CPU의 제어 신호를 설정하고, CPU는 이 필드를 참조하여 실행할 동작을 결정한다.

R-TYPE: op rs rt rd shamt funct field가 각각 6,5,5,5,5,6 bits를 차지하여 총 32bits를 이룬다.

I-TYPE: op rs rt imm field가 각각 6,5,5,16 bits를 차지하여 총 32bits를 이룬다.

J-TYPE: op(6bits) + 26 bit address를 가진다. 해당 26 bits를 통해 jump address를 계산한다.

Ⅲ. 명령어 별 구현한 control signal(설계 방법)

(1) ADDU

addu/EXEC: x_x_0_xxx_0_xx_xxx_0_x_000_000_00101_0x_xxx_xx_0_xxxxxxxx_11(0x02)

ADDU의 경우, R-type명령어로, MEM에 접근하는 MEM state가 없으므로 memory access와 관련있는 control signal은 모두 돈케어(x)로 주었다. MemWrite=0, IRwrite=0, RegWrite=0, PCwrite=0을 주며 write enable은 이 state에서 사용하지 않아도, x로 주지 않았다. EXEC단계에서 ALU를 사용하므로 ALUsrcA는 register A to ALU input A인 000을, ALUsrcB는 register B to ALU input B인 000을 주었다. ALUop는 unsigned 덧셈을 해주는 00101을 주었다. 다음 state는 state+1이므로 11이다.

addu/WB: x_x_0_xxx_0_01_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00(0x03)

ALU에서 계산된 결과값을 다시 RF의 rd에 써야하므로 MemWrite=0, IRwrite=0, RegWrite=1, PCwrite=0을 주고, RegDst는 rd이므로 01로 준다. ALUOut을 RF에 써야하므로 RegDatSel은 000으로 준다. 다음 state는 다시 0번 state이므로 00으로 설정한다.

(2) OR

x_x_0_xxx_0_xx_xxx_0_x_000_000_00001_0x_xxx_xx_0_xxxxxxxx_11 //0x04: OR/EXEC

OR의 경우, R-type명령어로, MEM에 접근하는 MEM state가 없으므로 memory access와

관련있는 control signal은 모두 돈케어(x)로 주었다. MemWrite=0, IRwrite=0, RegWrite=0, PCwrite=0을 주며 write enable은 이 state에서 사용하지 않아도, x로 주지 않았다. EXEC단계에서 ALU를 사용하므로 ALUsrcA는 register A to ALU input A인 000을, ALUsrcB는 register B to ALU input B인 000을 주었다. ALUop는 bitwise or을 해주는 00001을 주었다. 다음 state는 state+1이므로 11이다.

x_x_0_xxx_0_01_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00 // 0x05: OR/WB

R-type에서의 WB이므로, ADDU와 동일한 설정을 갖는다.

(3) ADDIU

x_x_0_xxx_0_xx_xxx_0_1_000_011_00101_0x_xxx_xx_0_xxxxxxxx_11 //0x06: ADDIU/EXEC

I-TYPE인 명령어이다. MEM state가 없으므로 memory access와 관련있는 control signal은 모두 돈케어(x)로 주었다. MemWrite=0, IRwrite=0, RegWrite=0, PCwrite=0을 주었다. EXEC단계에서 ALU를 사용하므로 ALUsrcA는 register A to ALU input A인 000을, ALUsrcB는 IMM값이 SEU를 거친 값인 011을 주었다. ALUop는 bitwise 덧셈을 해주는 000101을 주었다. 다음 state는 state+1이므로 11이다.

x_x_0_xxx_0_00_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00 // 0x07: ADDIU /WB

WB의 경우 rd대신 rt에 결과값을 쓰므로 RegDst는 00으로 준다. 그 외는 R-TYPE과 동일하다.

(4) XORI

x_x_0_xxx_0_xx_xxx_0_0_000_011_00011_0x_xxx_xx_0_xxxxxxxx_11 // 0x08: XORI /EXEC

I-TYPE인 명령어이다. MEM state가 없으므로 memory access와 관련있는 control signal은 모두 돈케어(x)로 주었다. MemWrite=0, IRwrite=0, RegWrite=0, PCwrite=0을 주었다. EXEC단계에서 ALU를 사용하므로 ALUsrcA는 register A to ALU input A인 000을, ALUsrcB는 IMM값이 SEU를 거친 값인 011을 주었다. ALUop는 bitwise XOR을 해주는 00011을 주었다. 다음 state는 state+1이므로 11이다.

x_x_0_xxx_0_00_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00 // 0x09: XORI /WB

WB의 경우 rd대신 rt에 결과값을 쓰므로 RegDst는 00으로 준다. 그 외는 R-TYPE과 동일하다.

(5) SLL

x_x_0_xxx_0_xx_xxx_0_x_000_000_01101_00_xxx_xx_0_xxxxxxxx_11// 0x0a: SLL /EXEC

R-type명령어로, MEM에 접근하는 MEM state가 없으므로 memory access와 관련있는 control signal은 모두 돈케어(x)로 주었다. MemWrite=0, IRwrite=0, RegWrite=0, PCwrite=0을 준다. EXEC단계에서 ALU를 사용하므로 ALUsrcA는 register A to ALU input A인 000을, ALUsrcB는 register B to ALU input B인 000을 주었다. shift는 shift

amount field에 있는 값만 큼해주므로 ALUctrl[0]=0이다. ALUop는 left shift연산을 해주는 01101을 주었다. 다음 state는 state+1이므로 11이다.

x_x_0_xxx_0_01_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00 // 0x0b: SLL /WB

다른 R-TYPE과 동일하다.

(6) SRAV

x_x_0_xxx_0_xx_xxx_0_x_000_000_01111_01_xxx_xx_0_xxxxxxxx_11 // 0x0c: SRAV $d = $t >>> $s/EXEC

R-type명령어로, MEM에 접근하는 MEM state가 없으므로 memory access와 관련있는 control signal은 모두 돈케어(x)로 주었다. MemWrite=0, IRwrite=0, RegWrite=0, PCwrite=0을 준다. EXEC단계에서 ALU를 사용하므로 ALUsrcA는 register A to ALU input A인 000을, ALUsrcB는 register B to ALU input B인 000을 주었다. shift는 rs 값만큼해주므로 ALUctrl[0]=1이다. ALUop는 sign right shift연산을 해주는 01111을 주었다. 다음 state는 state+1이므로 11이다.

x_x_0_xxx_0_01_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00 // 0x0d: SRAV /WB

다른 R-TYPE과 동일하다.

(7) SH

x_x_0_xxx_0_xx_xxx_0_1_000_011_00100_00_xxx_xx_0_xxxxxxxx_11 // 0x0e: SH /EXEC

I-type 명령어로, MEM access를 한다. EXEC단계에서는 ALU를 통해 메모리 주소값을 계산해주므로, ALUsrcA는 register A to ALU input A인 000을, ALUsrcB는 IMM값이 SEU를 거친 값인 011을 주었다. ALUop는 덧셈을 해주는 00100을 주었다. 다음 state는 state+1이므로 11이다.

1_x_1_010_0_xx_xxx_0_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00 // 0x0f: SH /MEM

ALU결과값을 가진 메모리 주소에 접근하므로 lorD는 1, MEM에서 읽는 행위는 하지않으므로 x, MemWrite=1, DataWidth는 010(half), 나머지 write enable=0으로 준다. 다음 state는 다시 0번 state이므로 00으로 준다.

(8) LH

x_x_0_xxx_0_xx_xxx_0_1_000_011_00100_0x_xxx_xx_0_xxxxxxxx_11 // 0x10: LH /EXEC

I-type 명령어로, MEM access를 한다. EXEC단계에서는 ALU를 통해 메모리 주소값을 계산해주므로, ALUsrcA는 register A to ALU input A인 000을, ALUsrcB는 IMM값이 SEU를 거친 값인 011을 주었다. ALUop는 덧셈을 해주는 00100을 주었다. 다음 state는 state+1이므로 11이다.

1_1_0_110_0_xx_xxx_0_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_11 // 0x11: LH/MEM

ALU결과값을 가진 메모리 주소에 접근하여 데이터를 읽으므로 lorD는 1, MemRead=1, MemWrite=1, DataWidth는 010(half), 나머지 write enable=0으로 준다. 다음 state는 다시 0번 state이므로 00으로 준다.

x_x_0_xxx_0_00_001_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00 // 0x12: LH/WB

다른 명령어들과 달리 mem에서 읽은 data를 RF에 적는 WB state가 하나 더 있다. 따라서 RegWrite=1이 된다. Mem을 포함하여 나머지에 대한 we는 0이다. rt에 저장하므로, RegDst는 00이다. Mem에서 읽은 값을 적으므로, RegDatSel은 001이다. 다음 state는 0번 state이므로, 00으로 준다.

(9) BLTZ

x_x_0_xxx_0_xx_xxx_0_x_000_010_10000_0x_101_01_1_xxxxxxxx_00 // 0x13: BLTZ

branch의 경우 address값은 이미 DECODE state에서 계산을 해주었으므로, ALU를 사용하여 조건문에 맞는지만 보면 된다. ALUsrcA는 register A to ALU input A인 000을, ALUsrcB는 0의 값인 010을 주었다. ALUop는 set less than인 10000을 주었다. branch의 경우 branch if not equal인 101을 주었다. PCsrc는 from ALUOut Register인 01을 주고, PCwrite=1로 주었다. 다음 state는 0번이므로 00을 준다.

(10) JAL

x_x_0_xxx_0_11_100_1_x_xxx_xxx_xxxxx_xx_000_10_1_xxxxxxxx_00 // 0x14: JAL

jump의 경우 no condition이므로 branch부분을 000으로 준다. PCwrite=1, PCsrc는 from jump address인 10으로 준다. 이 외의 we는 0으로 준다. ALU도 사용안하므로, 이것과 관련된 부분은 모두 don't care로 준다.

Ⅳ. 시뮬레이션 결과

(1) addu

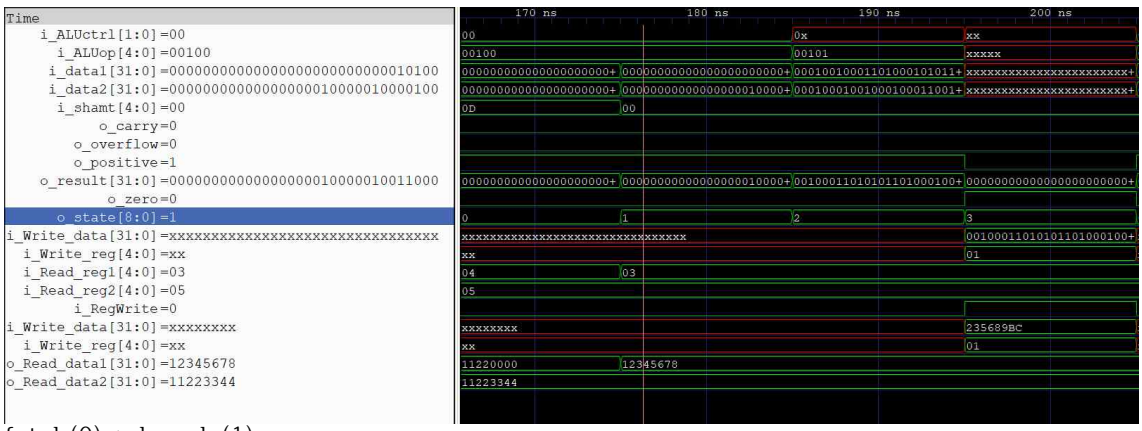
001111_00000_00010_0001_0010_0011_0100//lui $0 $2 0x1234

001101_00010_00011_0101_0110_0111_1000//ori $2 $3 0x5678

001111_00000_00100_0001_0001_0010_0010 //lui $0 $4 0x1122

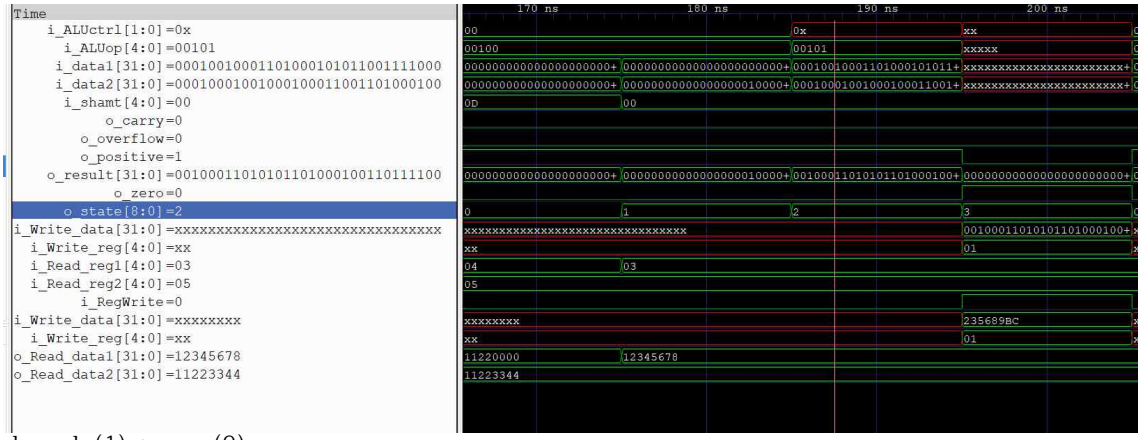
001101_00100_00101_0011_0011_0100_0100 //ori $4 $5 0x3344

000000_00011_00101_00001_00000_100001//addu(state=0x02)$1=$3+$5



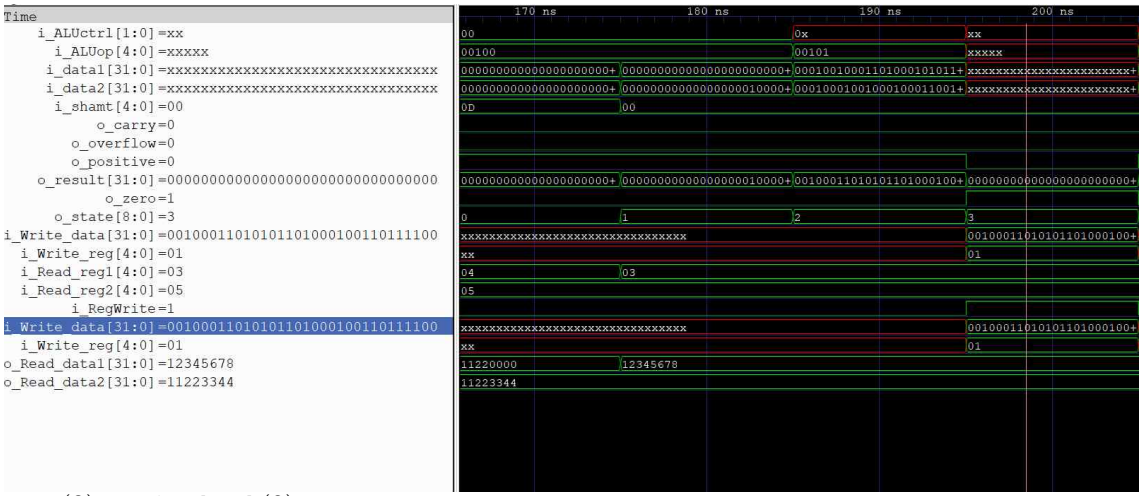
fetch(0)->decode(1)

I_read_reg1과 reg2가 각각 3번, 5번 레지스터를 잘 읽었고, data값이 12345678, 11223344로 알맞게 읽는다. decode이므로 state가 1이다.



decode(1)->exec(2)

r-type의 exec일 때 alu를 쓰므로, alu결과값을 확인해준다. addu결과값이 o_result에 잘 나오고, state도 +1이 되었다.

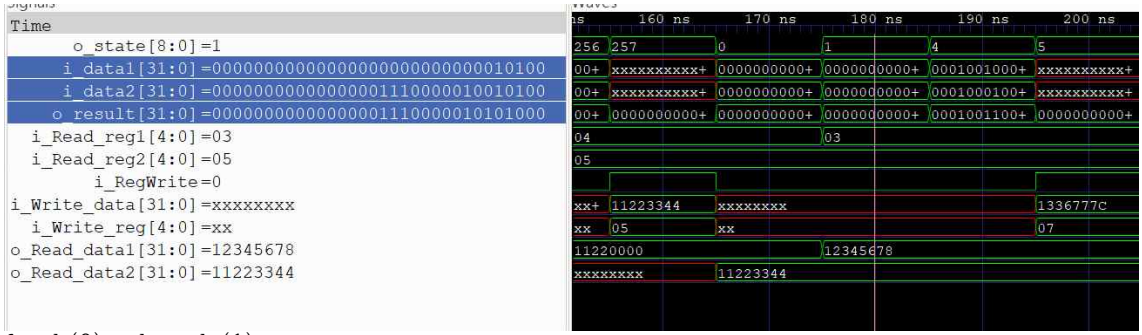


exec(2)->write back(3)

1번 레지스터에 alu result값을 잘 write하는 것을 확인할 수 있다.

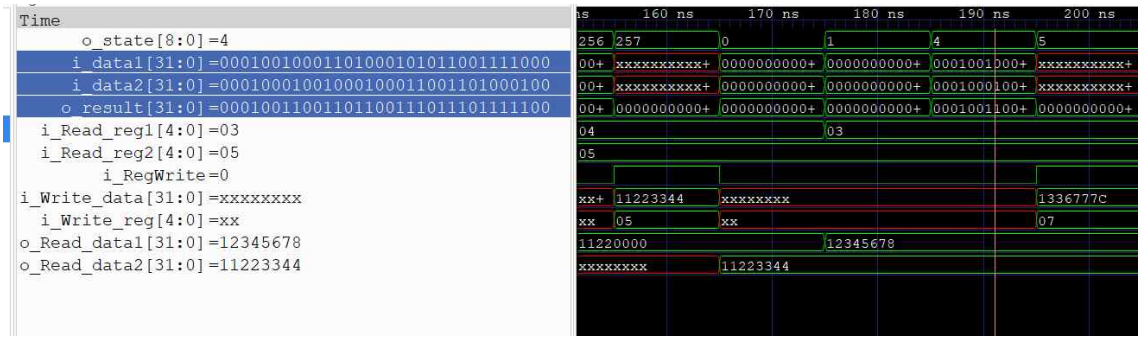
(2) or

000000_00011_00101_00111_00000_100101 //or(state=0x04) $7 = $3 | $5



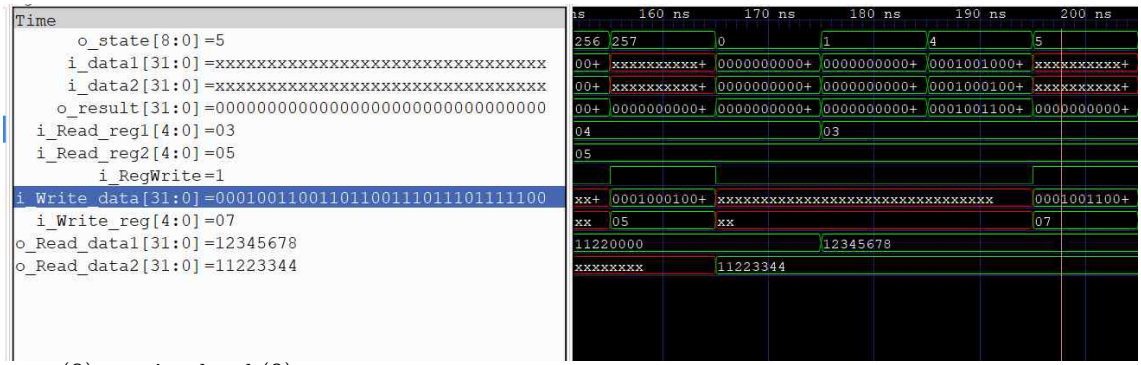
fetch(0)->decode(1)

I_read_reg1과 reg2가 각각 3번, 5번 레지스터를 잘 읽었고, decode이므로 state가 1이다.



decode(1)->exec(2)

r-type의 exec일 때 alu를 쓰므로, alu결과값을 확인해준다. or(bit 수준에서 0과1 둘중에 하나만 1이어도 1이 result에 저장) 결과값이 o_result에 잘 나오고, state도 +1이 되었다.

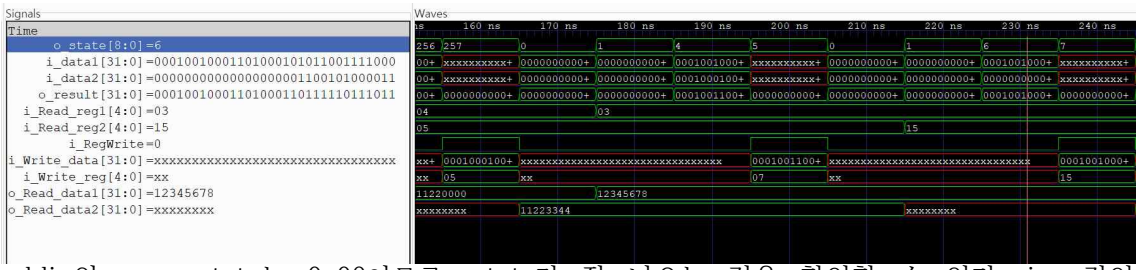


exec(2)->write back(3)

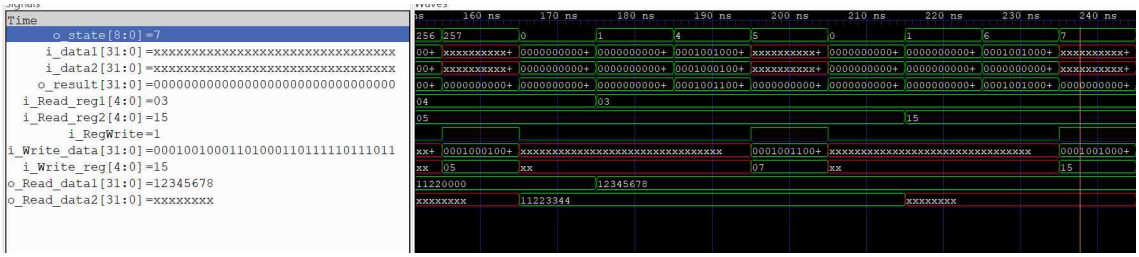
7번 레지스터(i_Write_reg)에 alu result값을 잘 write(i_RegWrite=1)하는 것을 확인할 수 있다.

(3) addiu

001001_00011_10101_00011_00101_000011//addiu(state=0x06) $15 = $3 + SE(i)



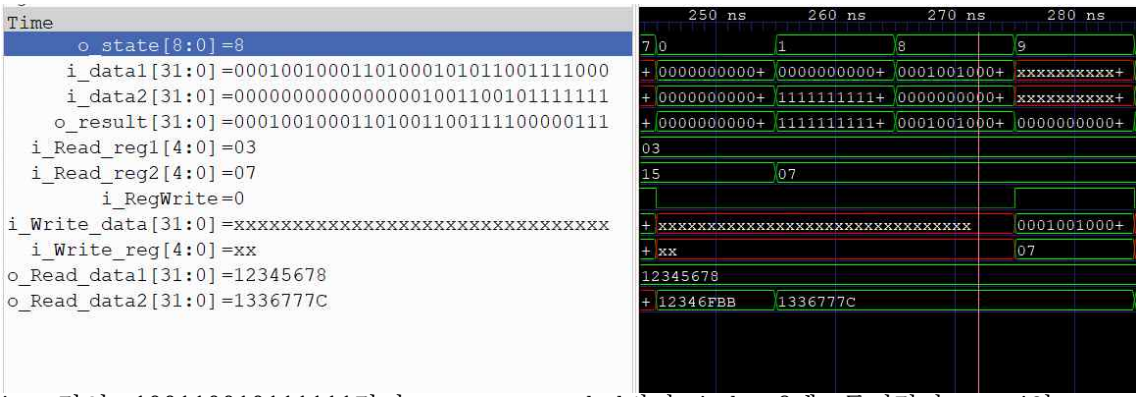
addiu의 exec state는 0x06이므로, state가 잘 나오는 것을 확인할 수 있다. imm값인 0001100101000011을 sign extended하여 i_data2로 들어간 것을 확인할 수 있다. rs값과 imm값을 unsigned 덧셈하였는데, 그 값이 예상한 값과 동일하게 잘 나오는 것을 확인할 수 있다.



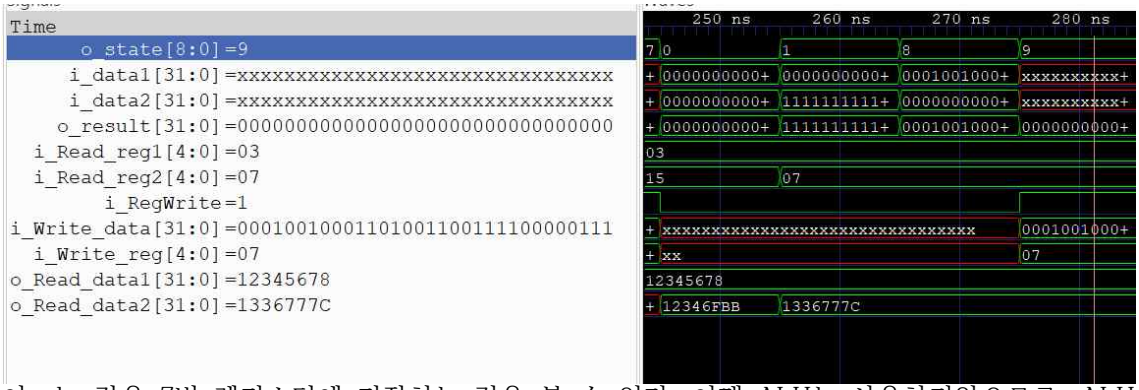
addiu의 wb state는 0x07이므로 해당 state(decimal)로 잘 이동한 것을 볼 수 있다. RegWrite enable은 1이고, 15번 레지스터에 alu result값을 잘 저장하는 것을 볼 수 있다.

(4) xori

001110_00011_00111_10011_00101_111111//xori(state=0x08) $7 = $3 ^ ZE(i)



imm값인 100110010111111값이 zero extended되어 i_data2에 들어갔다. xori의 exec state인 8(decimal)에 들어간 것을 확인할 수 있고, alu result를 보면 3번 레지스터의 값과 imm값의 xori(1이 홀수면 1) 잘 저장된 것을 확인할 수있다.



이 alu 값을 7번 레지스터에 저장하는 것을 볼 수 있다. 이때 ALU는 사용하지않으므로, ALU의 input값은 돈케어로 설정된다.

(5) srav

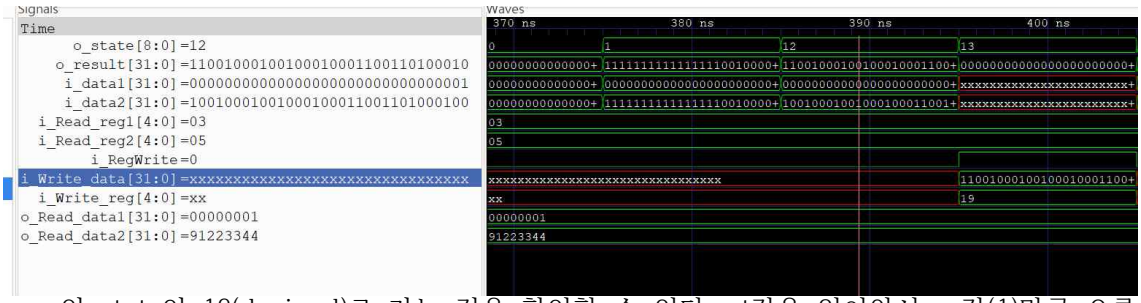
001111_00000_00010_0000_0000_0000_0000

001101_00010_00011_0000_0000_0000_0001 //$3에 0x00000001저장

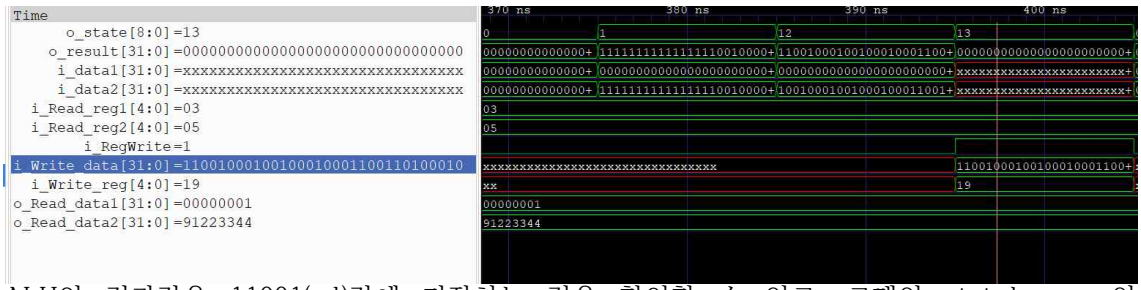
001111_00000_00100_1001_0001_0010_0010

001101_00100_00101_0011_0011_0100_0100 //$5에 0x91223344

000000_00011_00101_11001_00001_000111 //srav(state=0x0c) $d = $t >>> $s



srav의 state인 12(decimal)로 가는 것을 확인할 수 있다. rt값을 읽어와서 rs값(1)만큼 오른쪽으로 sign shift한 것이 o_result에 잘 저장되었다. rt의 MSB가 1이었으므로, 1이 오른쪽으로 한칸 더 생겨서 밀렸다.



ALU의 결과값을 11001(rd)값에 저장하는 것을 확인할 수 있고, 그때의 state는 exec의 state+1인 13(decimal)이다.

(6) sll

001111_00000_00010_0001_0010_0011_0100//lui $0 $2 0x1234

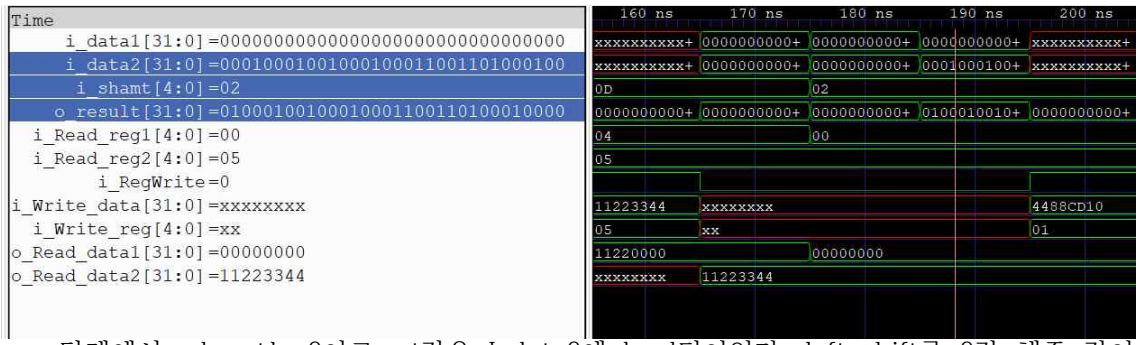
001101_00010_00011_0101_0110_0111_1000//ori $2 $3 0x5678

001111_00000_00100_0001_0001_0010_0010 //lui $0 $4 0x1122

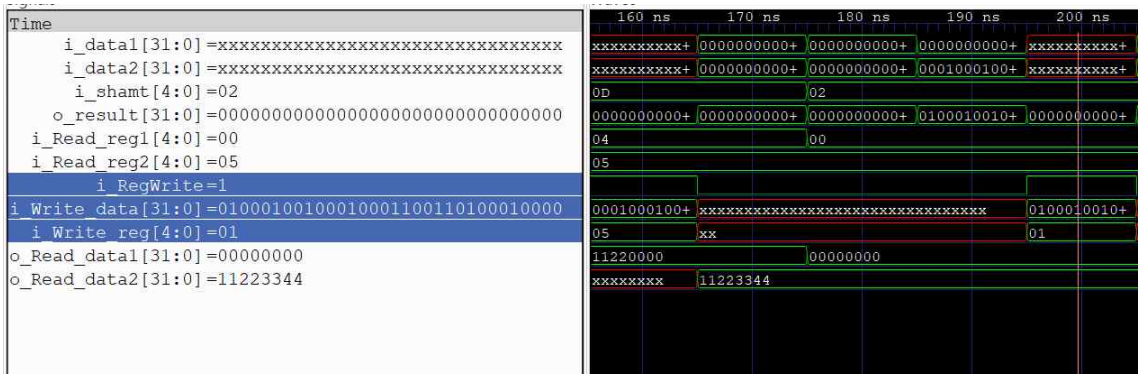
001101_00100_00101_0011_0011_0100_0100 //ori $4 $5 0x3344

000000_00000_00101_00001_00010_000000 //sll (state=0x0a) $d = $t << a

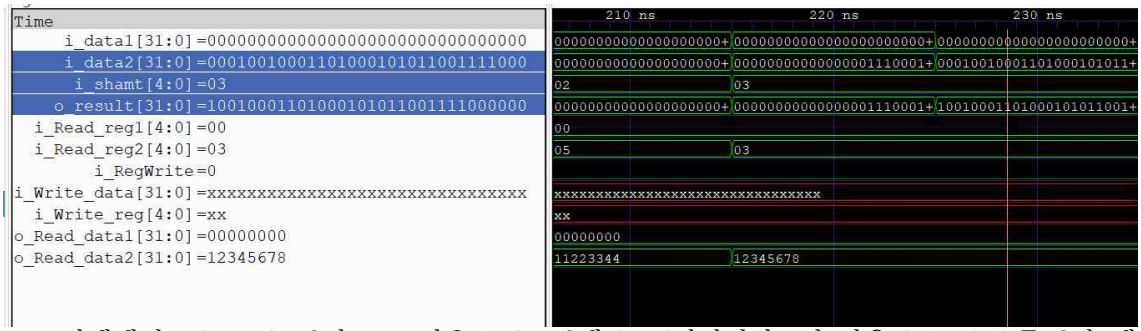
000000_00000_00011_00111_00011_000000 //sll (state=0x0a) $d = $t << a



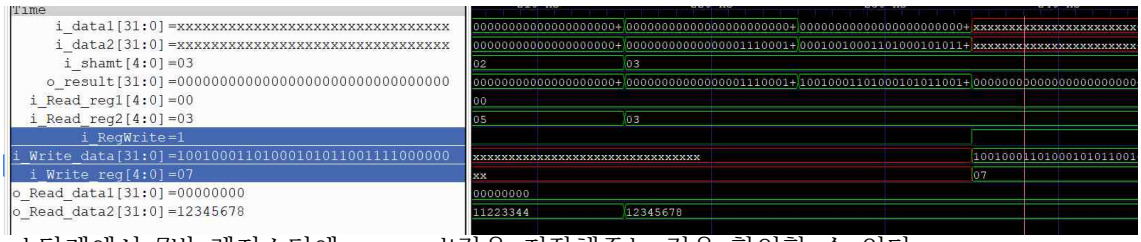
exec단계에서, shamt는 2이고, rt값은 I_data2에 load되어있다. left shift를 2칸 해준 값이 o_result에 잘 들어갔다.



wb단계에서 1번 레지스터에 o_result값을 저장해주는 것을 확인할 수 있다.



exec단계에서, shamt는 3이고, rt값은 i_data2에 load되어있다. 이 값을 left shift를 3칸 해준 값이 o_result에 잘 들어갔다.



wb단계에서 7번 레지스터에 o_result값을 저장해주는 것을 확인할 수 있다.

(7) sh

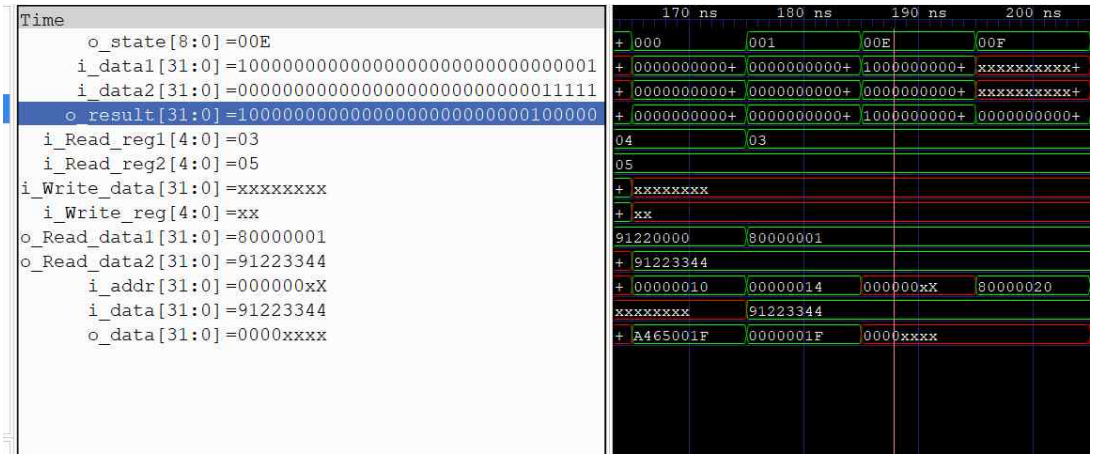
001111_00000_00010_1000_0000_0000_0000

001101_00010_00011_0000_0000_0000_0001 //$3에 0x80000001저장

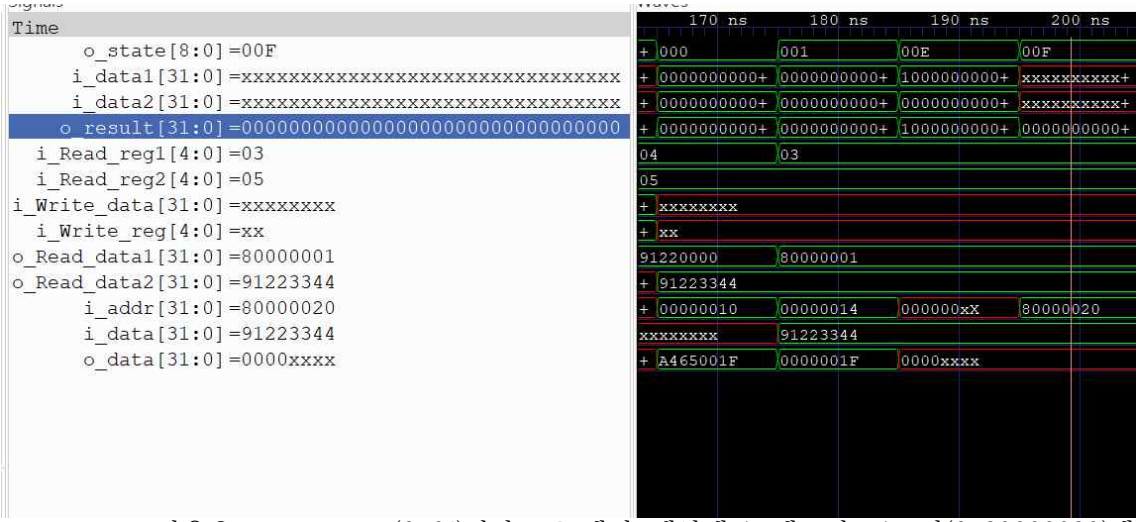
001111_00000_00100_1001_0001_0010_0010

001101_00100_00101_0011_0011_0100_0100 //$5에 0x91223344

101001_00011_00101_00000_00000_011111 //sh MEM[$s+i]:2 = LH($t) (state=0x0e)



exec단계에서 우선 mem주소를 계산해준다. rs값인 0x80000001와 imm값인 00000000011111을 더해준 값이 o_result(0x80000020)에 잘 들어간다.



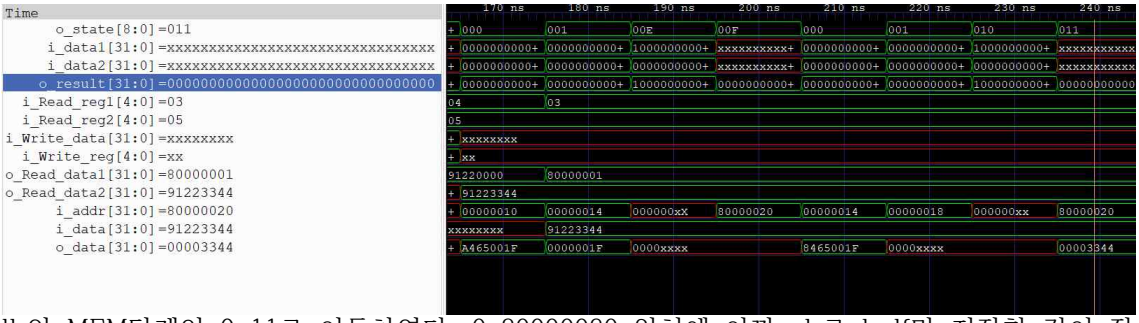
exec state 다음은 mem state(0x0f)이다. alu에서 계산해준 메모리 주소값(0x80000020)에 접근하는 것을 확인할 수 있고(i_addr), rt값인 91223344가 I_data로 들어가는 것을 확인할 수있다. 해당 메모리 주소 값에는 00003344가 들어 있다. 확인은 동일한 메모리 주소값에 들어가 data를 load하는 lh에서 한다.

(8) lh

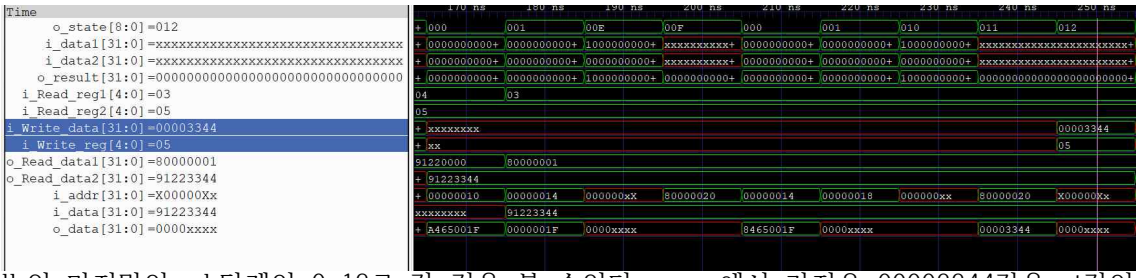
100001_00011_00101_00000_00000_011111 //lh $t = SE (MEM [$s + i]:2)(state=0x10)

lh의 exec state인 0x10으로 이동하는 것을 확인할 수있다. sh와 동일하게 메모리 주소값을

계산해준다. sh와 동일한 rs,imm값을 주었으므로, 0x80000020값이 계산이 되는 것을 확인할 수 있다.



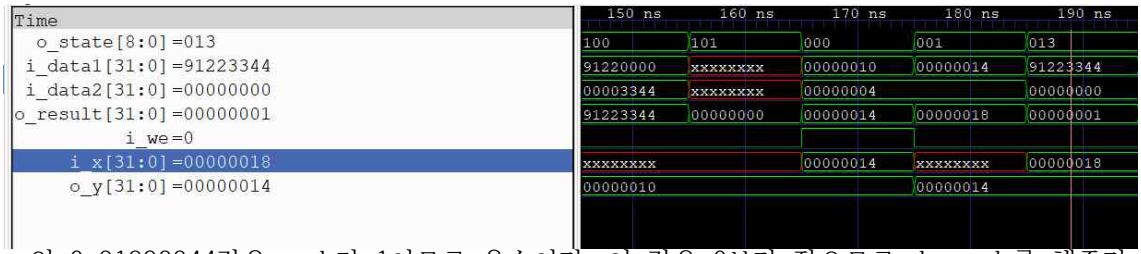
lh의 MEM단계인 0x11로 이동하였다. 0x80000020 위치에 아까 sh로 half만 저장한 값이 잘 있는지 확인한다. 해당 메모리에 갔더니 해당 데이터가 o_data로 나오는데, 00003344가 나오는 것을 통해 sh가 잘되었음을 확인이 가능하다.



lh의 마지막인 wb단계인 0x12로 간 것을 볼 수있다. mem에서 가져온 00003344값을 rt값인 5번 레지스터에 잘 저장해주는 것을 확인할 수 있다.

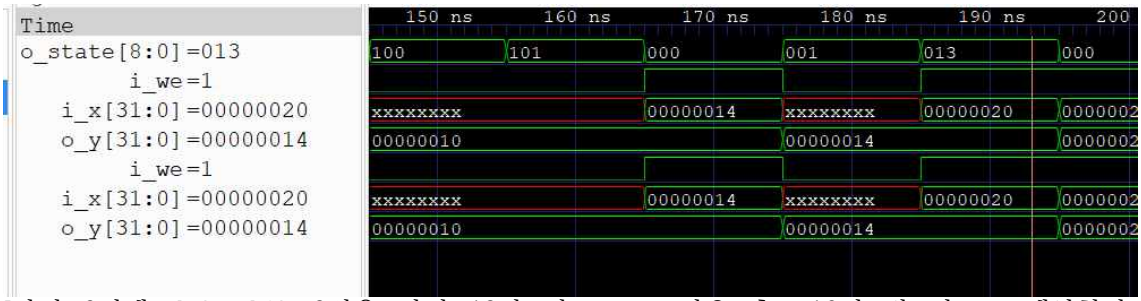
(9) bltz

000001_00101_00000_00000_00000_000001 //bltz if ($s < 0) pc += i << 2(state=0x13)



rs의 0x91223344값은 msb가 1이므로 음수이다. 이 값은 0보다 작으므로, branch를 해준다. I값은 1을 주었으므로 left shift 2번을 해주었으므로 최종적으로 PC= PC+4+4이다. PC+4는 FETCH단계에서 해주고 0x13(branch exec)부분에서 또 +4를 해주어 PC값은 최종적으로 0x00000010->0x00000018로 +8이 된 것을 확인할 수있다.

000001_00101_00000_00000_00000_000011 //bltz if ($s < 0) pc += i << 2(state=0x13)

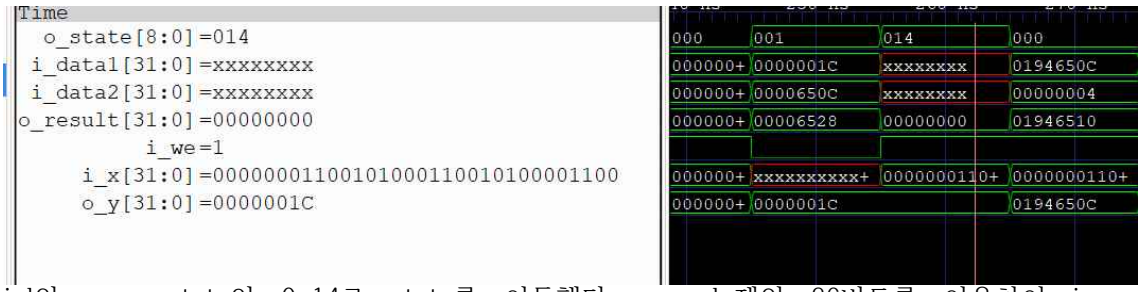


I값이 3인데, left shift 2번을 하면 12가 되므로, pc값은 총 +16이 될 것으로 예상한다.

testbench를 보면, branch exec state에서 0x00000010에서 0x000020이 된 것을 확인할 수 있다.

(10) jal

000011_00011_00101_00011_00101_000011 //jal(0x14)



jal의 exec state인 0x14로 state를 이동했다. opcode제외 26비트를 이용하여 jump address를 pc값을 만든다. 기존 pc에서 맨앞 0000을 떼고, 00011001010001100101000011+00을 해주어 pc값을 만들어 그 pc값으로 jump를 해주었다.

(11) 반복된 명령어(한번 더 검증)

001111_00000_00010_1000_0000_0000_0000

001101_00010_00011_0000_0000_0000_0001 //$3에 0x80000001저장

001111_00000_00100_1001_0001_0010_0010

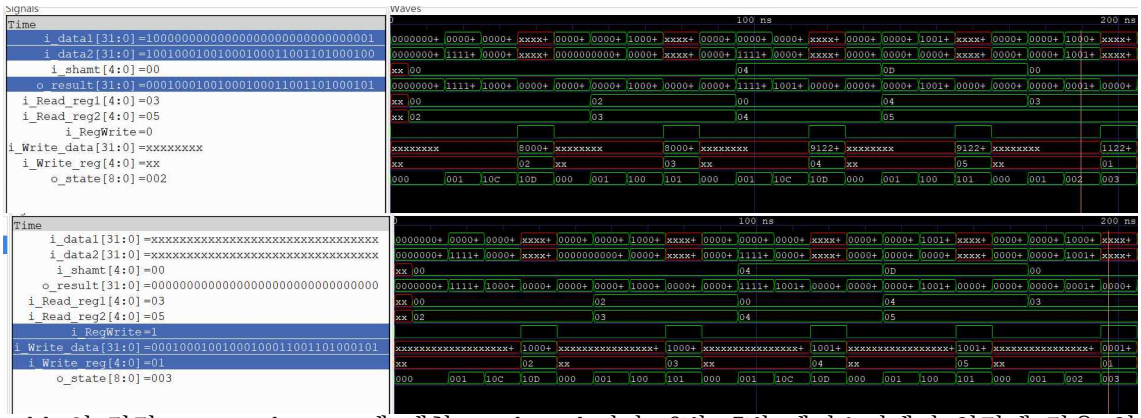
001101_00100_00101_0011_0011_0100_0100 //$5에 0x91223344

000000_00011_00101_00001_00000_100001//addu(state=0x02)$1=$3+$5

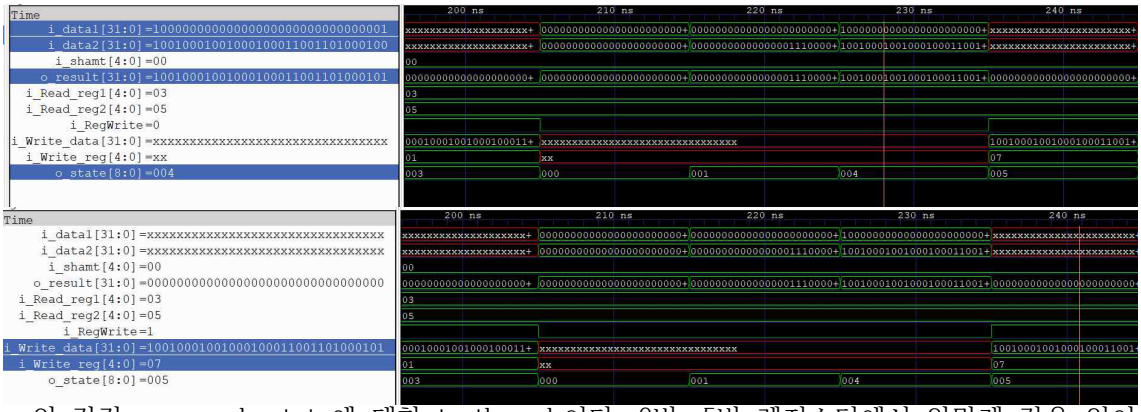
000000_00011_00101_00111_00000_100101 //or(state=0x04) $7 = $3 | $5

001001_00011_10101_00011_00101_000011//addiu(state=0x06) $15 = $3 + SE(i)

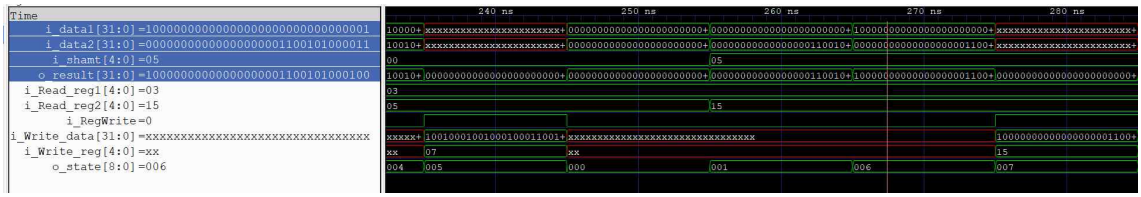
001110_00011_00111_10011_00101_111111//xori(state=0x08) $7 = $3 ^ ZE(i)

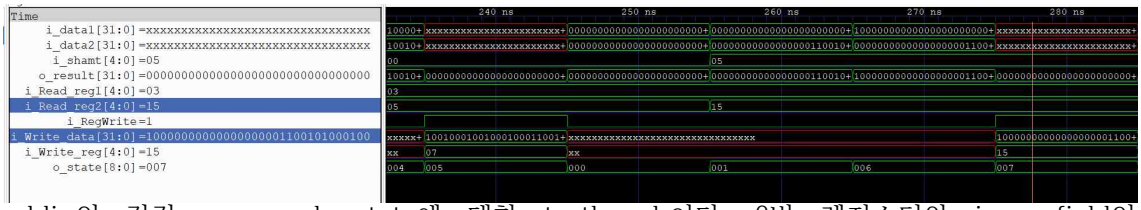


addu의 각각 exec, wb state에 대한 testbench이다. 3번, 5번 레지스터에서 알맞게 값을 읽어와 ALU에서 unsigned add연산을 해 준 후, 그 값을 wb state에서 1번 레지스터에 쓴 것을 확인할 수있다.

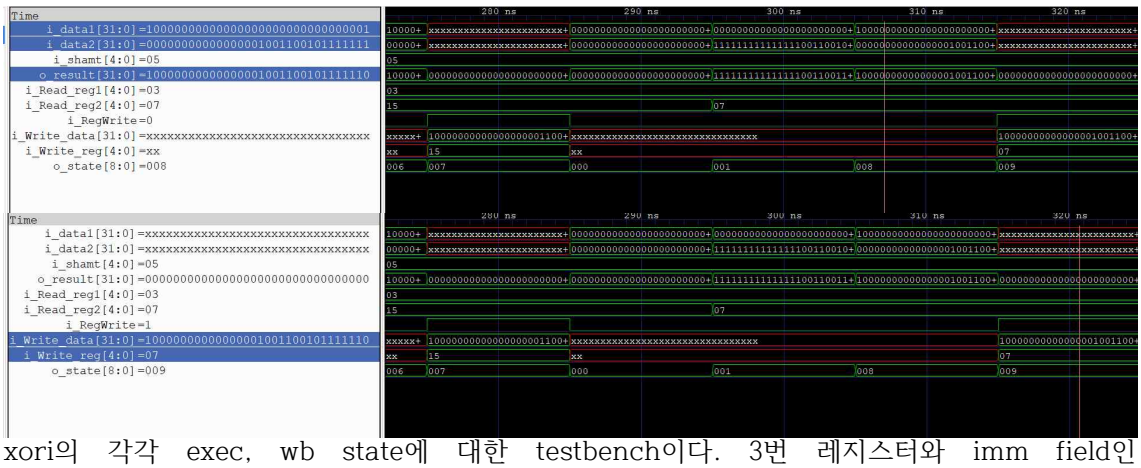


or의 각각 exec, wb state에 대한 testbench이다. 3번, 5번 레지스터에서 알맞게 값을 읽어와 ALU에서 or연산을 해 준 후, 그 값을 wb state에서 7번 레지스터에 쓴 것을 확인할 수있다.





addiu의 각각 exec, wb state에 대한 testbench이다. 3번 레지스터와 imm field인 00011_00101_000011에서 알맞게 값을 읽어와 ALU에서 unsigned add 연산을 해 준 후, 그 값을 wb state에서 15번 레지스터에 쓴 것을 확인할 수 있다.

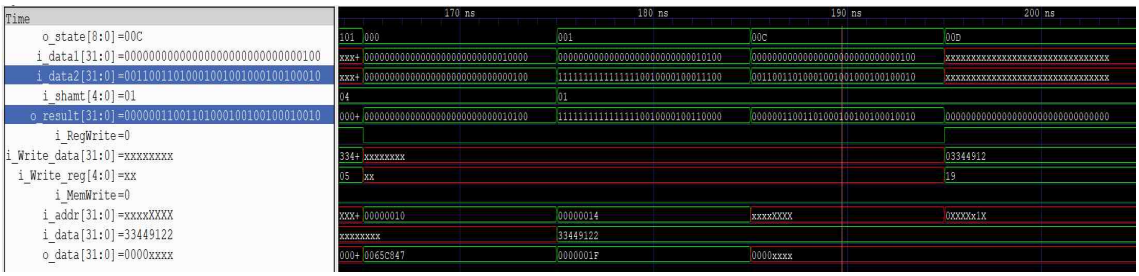


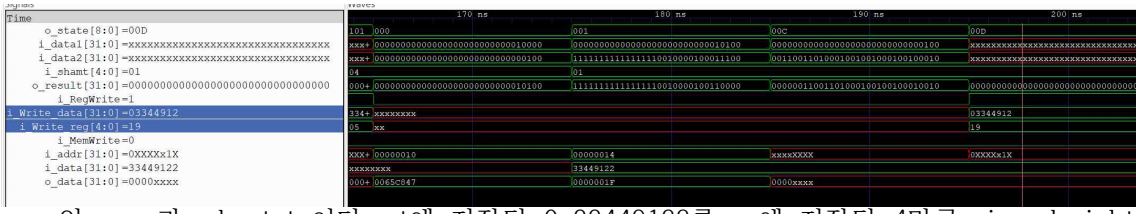
xori의 각각 exec, wb state에 대한 testbench이다. 3번 레지스터와 imm field인 10011_00101_111111에서 알맞게 값을 읽어와 ALU에서 xor연산을 해 준 후, 그 값을 wb state에서 7번 레지스터에 쓴 것을 확인할 수 있다.

001111_00000_00010_0000_0000_0000_0000

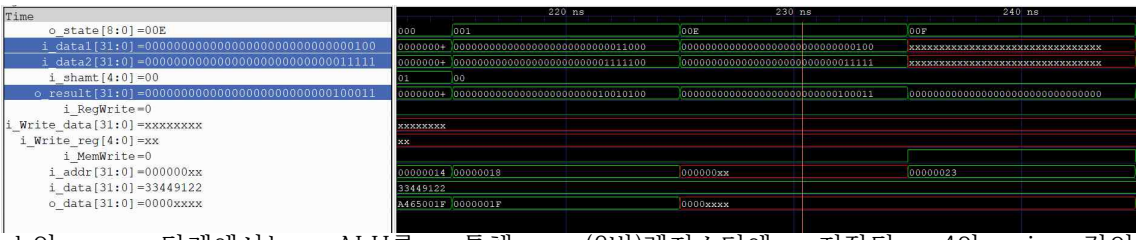
001101_00010_00011_0000_0000_0000_0100 //$3에 0x00000004저장
001111_00000_00100_0011_0011_0100_0100
001101_00100_00100_1001_0001_0010_0010 //$5에 0x33449122

000000_00011_00101_11001_00001_000111 //srav(state=0x0c) $d = $t >>> $s
101001_00011_00101_00000_00000_011111 //sh MEM[$s+i]:2 = LH($t) (state=0x0e)
100001_00011_00101_00000_00000_011111 //lh $t = SE (MEM [$s + i]:2) ->i type(state=0x10)

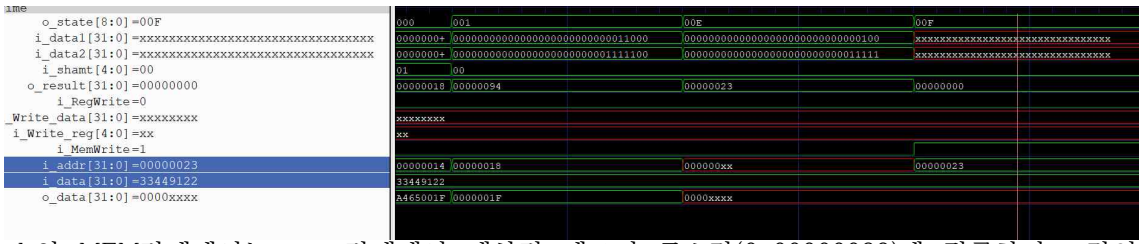




srav의 exec과 wb state이다. rt에 저장된 0x33449122를 rs에 저장된 4만큼 signed right shift를 해줄 것을 예상한다. o_result를 보면, rt값이 오른쪽으로 4칸씩 움직인 것을 확인할 수있다. msb가 0이므로 0이 앞에 4개 추가되고 [3:0]인 0010이 없어졌다. wb에서는 19번 레지스터에 o_result값이 저장된 것을 확인할 수있다.

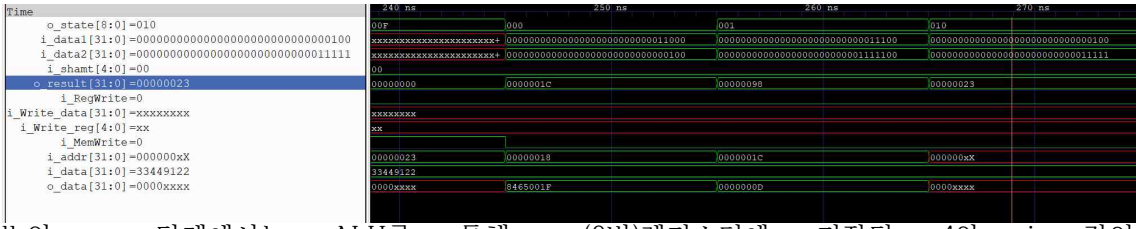


sh의 exec단계에서는 ALU를 통해 rs(3번)레지스터에 저장된 4와 imm값인 0000000000011111을 더하여 메모리 주소를 계산한다. 그렇게 계산된 값은 HEX값으로 0x00000023이다.

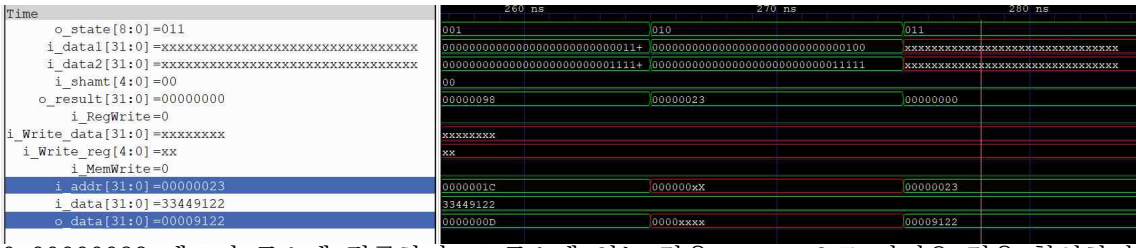


sh의 MEM단계에서는 exec단계에서 계산된 메모리 주소값(0x00000023)에 접근하여 rt값인 0x33449122의 half부분을 저장한다. half부분만 저장되었는지는 다음 lh를 통해 알아볼 수

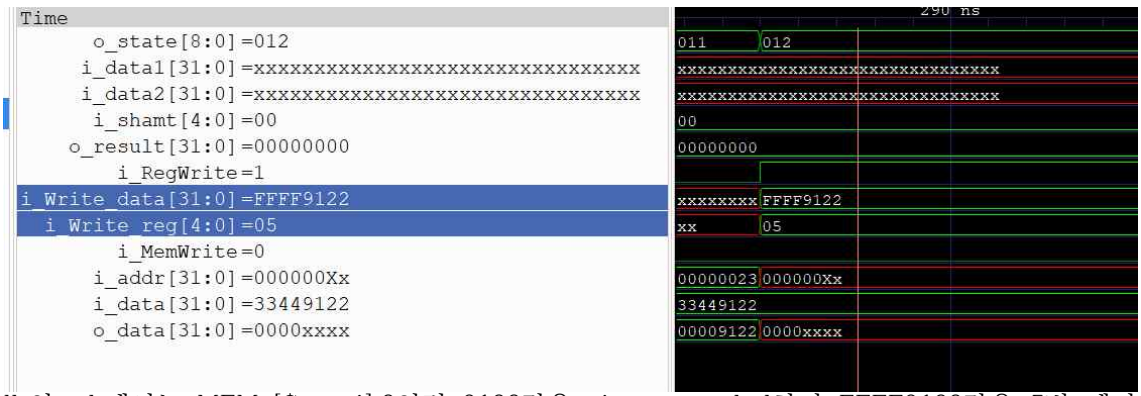
있다.



lh의 exec단계에서는 ALU를 통해 rs(3번)레지스터에 저장된 4와 imm값인 0000000000011111을 더하여 메모리 주소를 계산한다. 그렇게 계산된 값은 HEX값으로 0x00000023이다.

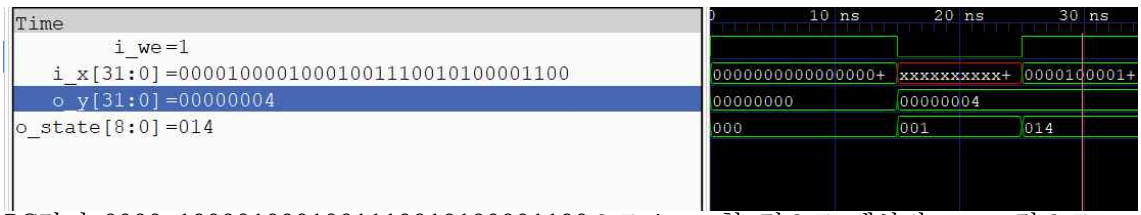


0x00000023 메모리 주소에 접근하여 그 주소에 있는 값을 output으로 가져온 것을 확인하니 0x00009122인 것을 확인할 수 있다. 즉, sh에서 33449122값에서 half인 9122만 해당 주소 값에 써진 것을 확인했다.



lh의 wb에서는 MEM [$s + i]:2였던 9122값을 sign extended하여 FFFF9122값을 5번 레지스터에 LOAD하는 것까지 확인할 수 있다.

000011_10000_10001_00111_00101_000011 //jal(0x14)



PC값이 0000+10000100010011100101000011100으로 jump할 것으로 예상되고, 그 값으로 pc값이 바뀐 것을 위의 test bench를 통해 확인할 수 있다.

Ⅴ. 고찰

이번 프로젝트는 저번 프로젝트와 연관성이 있어서 이해하는 데에 어렵지 않았다. 저번에는 single cycle이므로 한 cycle 내에 alu계산 및 write back, memory access까지 다 해줘야 해서 control signal들이 한 번에 많이 필요하고, adder의 경우도 pc를 계산하고, alu 연산도 해줘야해서 여러개 필요했다. 하지만 multi-cycle로 구현하니까 state에 따라 움직이기 때문에 복잡할진 몰라도 한 cycle마다의 control signal들을 많이 해줄 필요가 없어 덜 헷갈리고 좀 더 가벼워진 느낌의 설계였다. 다만, state가 내가 구현한 state로 다른 명령어들은 잘 나오는데, SLL 명령어의 경우 작동(EXEC,WB)은 잘 해주는데, state값이 이상한 값이 출력되어 이 부분은 왜 그런지 잘 모르겠지만, 이 부분 때문에 꽤 시간을 잡아먹었다. multi-cycle의 핵심은 FSM이라는 부분을 설계를 통해 더 확실히 이해할 수 있던 시간이었고, 가장 단계가 많은 LOAD를 다른 명령어들이 기다려 줄 필요가 없는 multi-cycle의 장점도 잘 이해할 수있던 시간이었다.