컴퓨터구조 Report

Project #2 – MIPS Multi Cycle CPU

담당교수: 이성원 교수님

실습분반: 수 7교시

학번: 2019202050

이름: 이강현

[Introduction]

이번 프로젝트는 multi cycle cpu를 microprogramming technique으로 구현해보는 것으로 다음과 같은 명령어들이 잘 동작하게 하기 위해서 Finite State Machine으로 구현한다. control unit의 동작을 결정하는 dispatch ROM 과 microprogram ROM을 각각 ROM_DISP.txt, ROM_MICRO.txt에 구현한다.

addu: add unsigned 명령어로, 부호가 없는 두 개의 레지스터(rs,rt)의 값을 더하고 그 결과를 레지스터(rd)에 저장한다.

or: bitwise or 명령어로, 두 개의 레지스터(rs,rt)에 대해 비트 단위 OR 연산을 수행하고 그 결과를 레지스터(rd)에 저장한다.

addiu: add immediate unsigned 명령어로, 레지스터(rs)에 상수 값(imm)을 더하고 그 결과를 레지스터(rt)에 저장한다.

xori: exclusive or immediate 명령어로, 레지스터(rs)와 상수 값(imm)간의 비트 단위 XOR 연산을 수행하고 그 결과를 레지스터(rt)에 저장한다.

sll: shift left logical 명령어로, 레지스터(rt)에 대해 비트 단위 left shift을 shift amount 만큼 수행하고 그 결과를 레지스터(rd)에 저장한다.

srav: shift right arithmetic variable 명령어로, 레지스터(rt)에 대해 비트 단위 right shift을 레지스터(rs)의 값만큼 수행하고 그 결과를 레지스터(rd)에 저장한다.

sh: store halfword 명령어로, 메모리 상의 주소(rs + imm)에 halfword 크기의 값(rt)을 저장합니다. 이 때, 상위 16비트는 0으로 채워진다.

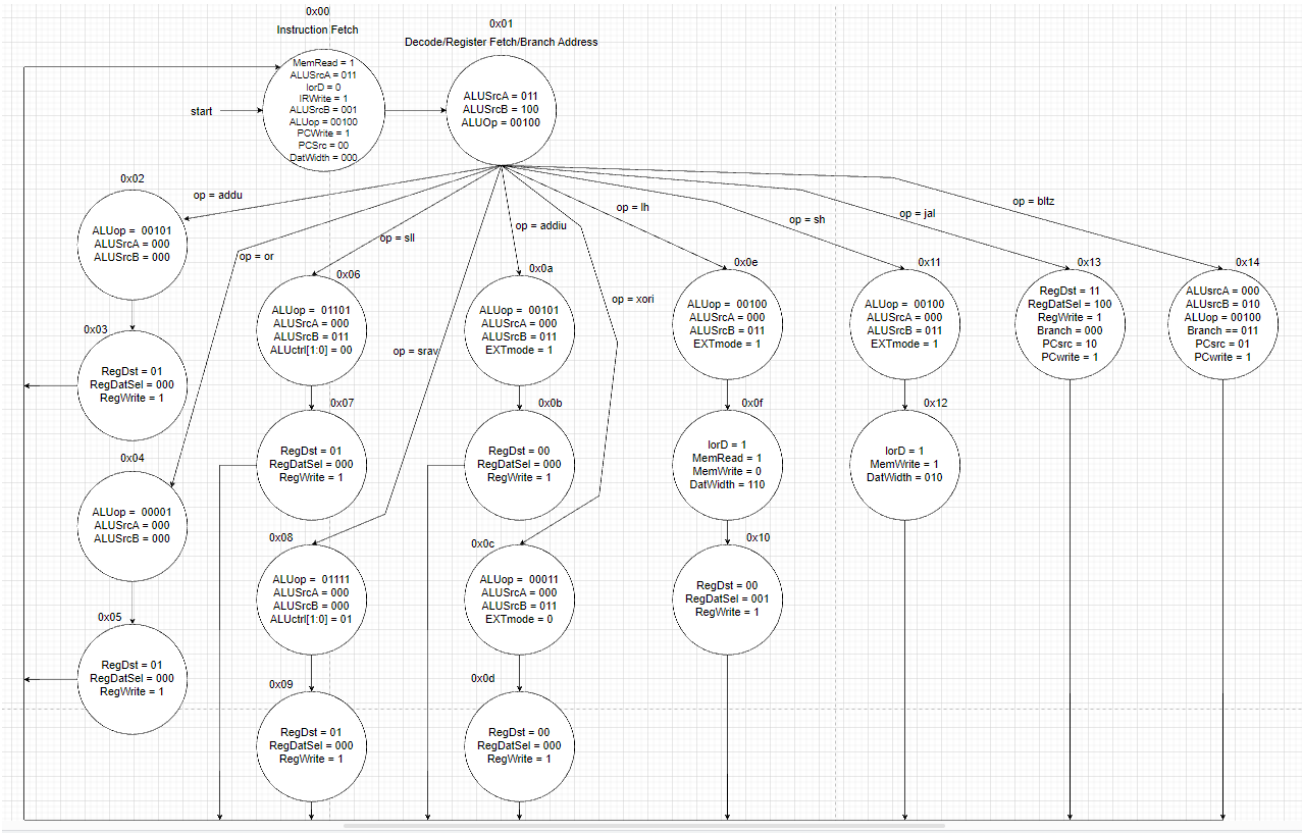
lh: load halfword 명령어로, 메모리 상의 주소(rs + imm)로부터 halfword 크기의 값을 로드하여 레지스터(rt)에 저장한다. 이 때, 상위 16비트는 부호 비트와 같은 값으로 채워진다.

bltz: branch less than zero 명령어로, 레지스터(rs)의 값이 0보다 작으면 label로 분기한다.

jal: jump and link 명령어로, 다음 명령어의 주소를 레지스터($31)에 저장한 후, label로 분기한다.

[Assignment]

1. Multi Cycle CPU FSM Diagram



위는 직접 구현한 FSM의 모습이다. state들은 ROM_MICRO에 정의되어 있고 모든 명령어들은 공통적으로 Instruction Fetch와 Decode/Register Fetch를 지니므로 이들은 ROM_MICRO에 각각 0x00과 0x01에 위치한다.

instruction이 decode되면 ROM_DISP에 정의된 방법으로 명령어의 opcode/function/regim 부분 비트를 읽고 적절한 state로 이동하게 된다.

각 명령어들은 명령이 마무리되면 다시 0x00 state로 돌아가게 되어 다음 instruction을 읽는 작업을 반복한다.

R-type 명령어들의 state signal들은 큰 차이가 없으며 이들은 opcode만 다른 경우가 대부분이기 때문에 하나의 state로 묶고 싶었으나 다음 state를 결정하는 dispatch가 1개밖에 없기 때문에 동일한 state로 이동한 후 다시 state를 분기할 수 없기에 위와 같은 state를 구성하였다.

위의 state들은 사용하는 unit들에 대한 control signal만을 명시해 두었으며 사용하지 않는 unit들에 대한 control signal들은 write signal을 제외하고 모두 don't care이다.

해당 state일 때 사용하지 않는 unit이라 할지라도 입력값이 없는 것은 아니므로 의도

치 않은 입력값이 쓰여질 수 있는 register나 Memory의 write signal은 1이 아니면 무조건 0으로 고정해 두었음을 알린다.

2. Micro-instruction의 Field 구분 및 Field 용도

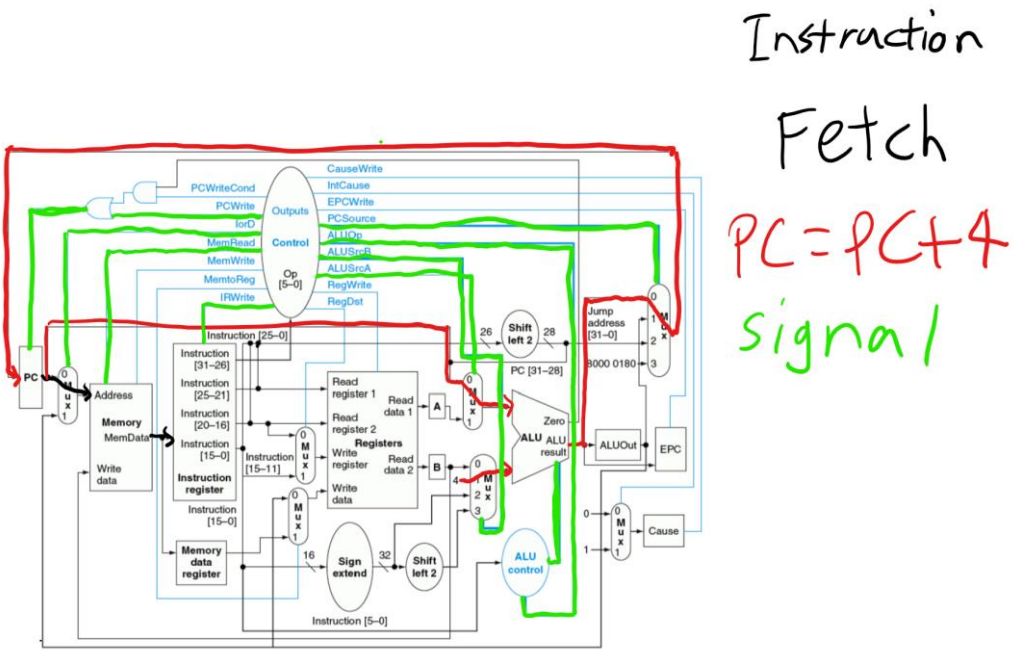
Field name	signal
ALU	EXTmode ALUsrcA ALUsrcB ALUop ALUctrl[1:0]
Register	RegDst RegDatSel RegWrite
Memory	lorD MemRead MemWrite DatWidth IRwrite
PC write control	Branch PCsrc PCwrite
sequencing	StateSel

모든 control signal들을 회로의 각 unit들을 사용할 때 주로 함께 사용되는 신호들로 묶어 구분해보았다. 사용되는 unit과 연관된 signal들을 위처럼 묶어두면 새로운 명령어를 정의할때도 state를 정의하기 편하다는 장점이 있다.

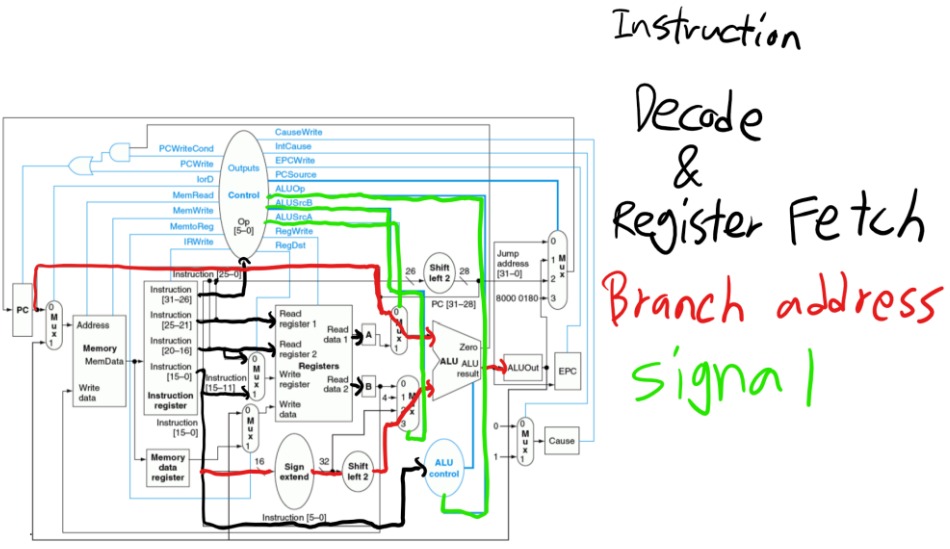
예를 들면 r-type 명령의 execution단계에서의 state를 정의한다고 가정하였을 때 ALU에서 rs와 rt를 이용하여 opcode에 맞게 연산을 진행한다. 그렇다면 ALU의 Field에 있는 signal들을 위주로 디자인하면 된다.

이번엔 각 명령어에 대한 흐름을 설명하려고 한다.

우선 Instruction Fetch이다.



우선 검정색으로 표시된 부분이 instruction fetch가 일어나는 부분이다. PC의 값에 해당하는 Memory주소의 값을 IR에 쓴다. 그와 동시에 빨간색으로 PC의 값을 증가시키기 위해 ALU를 사용하는 것을 볼 수 있다. 이때 사용되는 signal들은 초록색으로 표기하였다. PC값을 증가시키고 PC에 새로운 값을 넣어야 하고 Memory의 값을 읽어야하고 ALU를 사용하기 때문에 많은 signal들이 사용됨을 알 수 있다.

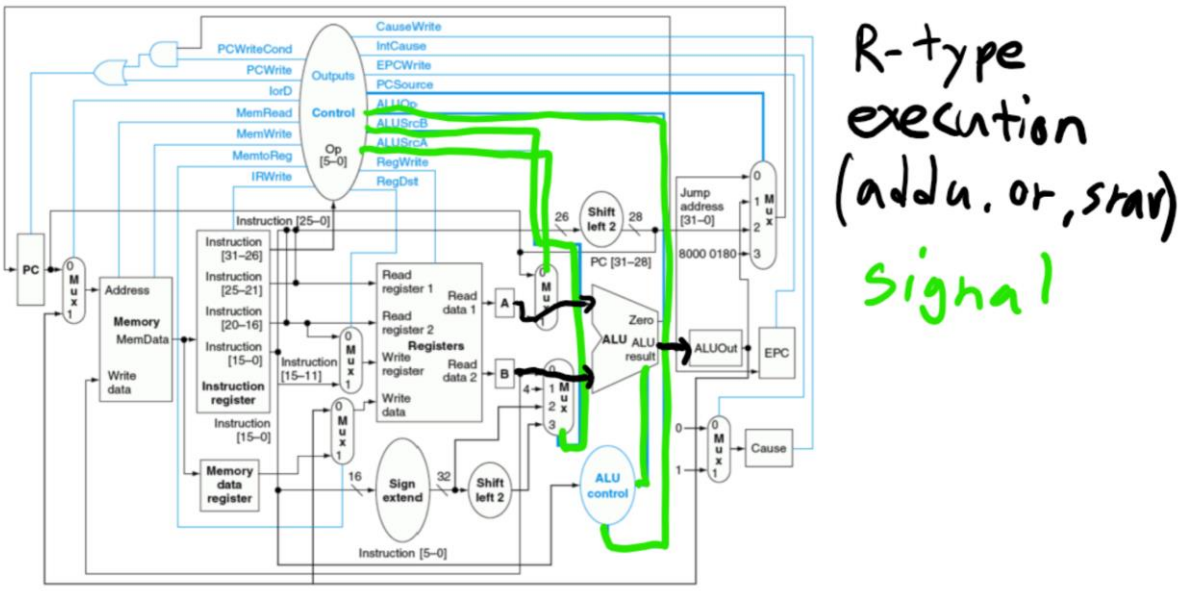


다음은 instruction decode, register fetch, branch address 단계이다.

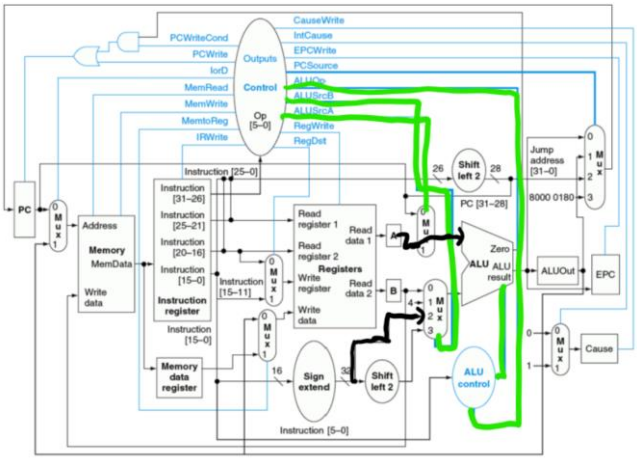
검정색으로 표시된 부분은 instruction을 비트단위로 쪼개 opcode와 function값 부분을 control unit에 넣어주고(instruction decode) register값을 읽어 A와 B에 넣는 것을 보여준다.(register fetch) 그리고 빨간색으로 표시된 부분은 해당 단계에서 ALU를 사용하지 않기에 다음에 Branch 명령이 오게 될 것을 미리 대비해 branch할 주소를 생성하여 ALUOut에 넣어주는 것을 보여준다. 생성된 주소는 다음 명령이 branch가 오게 되면 ALUOut에 저장된 값을 그대로 사용하여 PC에 넣어주면 되고 이를 통해 시간을 절약할 수 있다. branch 명령이 아니더라도 ALUOut에 새로운 값을 넣으면 생성한 주소는 사라지므로 다른 명령어의 동작에 영향을 주지 않는다.

instruction decode를 하는 단계이기 때문에 어떤 명령이 진행될 지 아직 알 수 없는 상태이기에 register를 읽기만 하게 된다. 따라서 branch address를 계산하기 위한 ALU field의 신호들만 필요로 하는 단계이다.

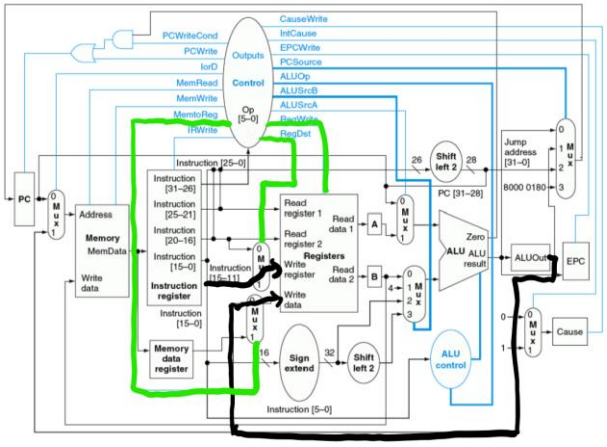
아래부터는 instruction이 decode된 상태이기 때문에 각 명령어에 맞게 signal들이 달라지고 명령에 맞는 동작을 진행한다.



우선 R-type이다. R-type명령은 이번 과제에서 addu,or,sll,srav 이렇게 4개가 존재하는데 이중 addu,or,srav를 묶어서 설명하는 이유는 모두 동일하게 rs와 rt를 사용하여 연산을 진행하고 이를 ALUOut에 넣기 때문이다. 이들의 다른 점은 ALU 내에서 어떤 연산을 진행하는지만 다를 뿐이다. 따라서 ALUop와 ALUctrl[1:0]을 제외한 모든 signal이 동일하다.

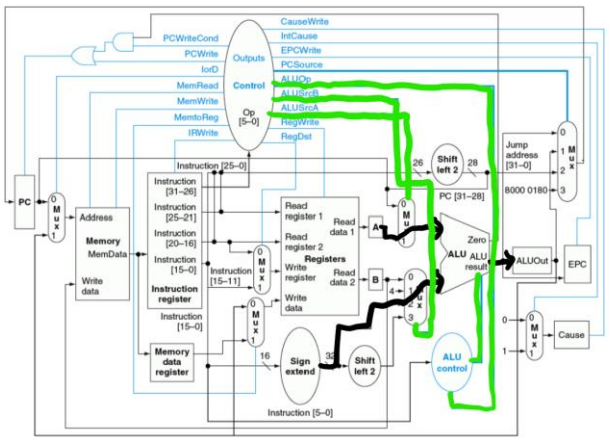


이번에는 R-type중 sll이다. sll은 r-type이지만 rs대신 shift amount의 값을 사용하기 때문에 하위 16비트의 값중 5비트를 ALU의 입력으로 사용하기 때문에 구분해 두었다.



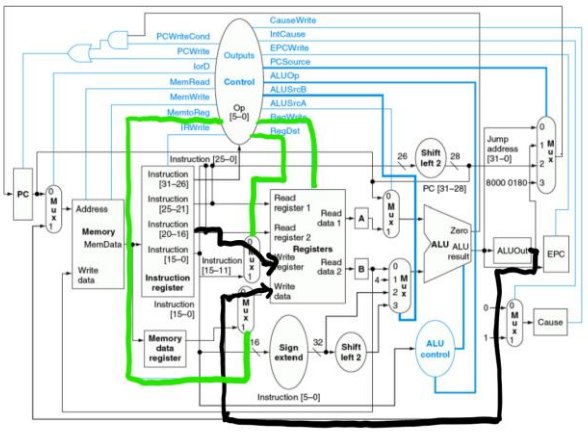
이번에는 R-type의 write back 단계이다. 이번에 주어진 r-type명령어들은 모두 rd에 ALUOut값을 저장하기 때문에 위와 같이 모두 동일한 signal과 흐름을 가진다.

addiu, xori execution signal

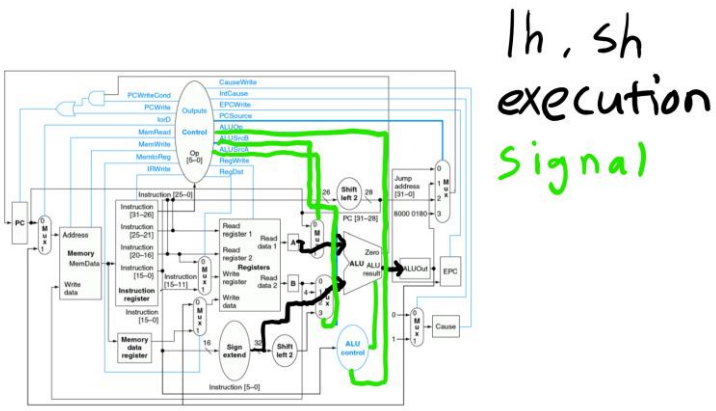


addiu와 xori 명령 또한 묶어서 설명이 가능하다. 그 이유 또한 r-type을 묶어서 설명한 이유와 동일하다. 둘 다 rs와 immediate값을 사용하여 연산을 진행하기 때문이다. 따라서 각각 unsigned add, xor연산을 한다는 차이점만 존재하기에 ALUop를 제외한 나머지 signal들이 모두 동일하다.

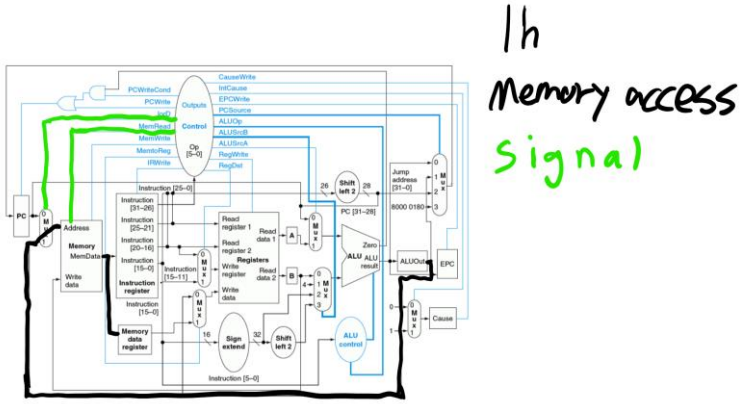
addiu, xori write back signal



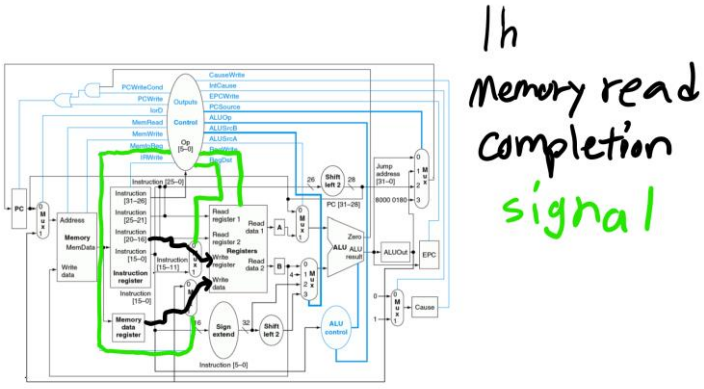
write back단계도 마찬가지이다. ALUOut의 값을 두 명령어 모두 rt에 저장하기 때문에 둘은 signal과 값의 흐름이 동일하다.



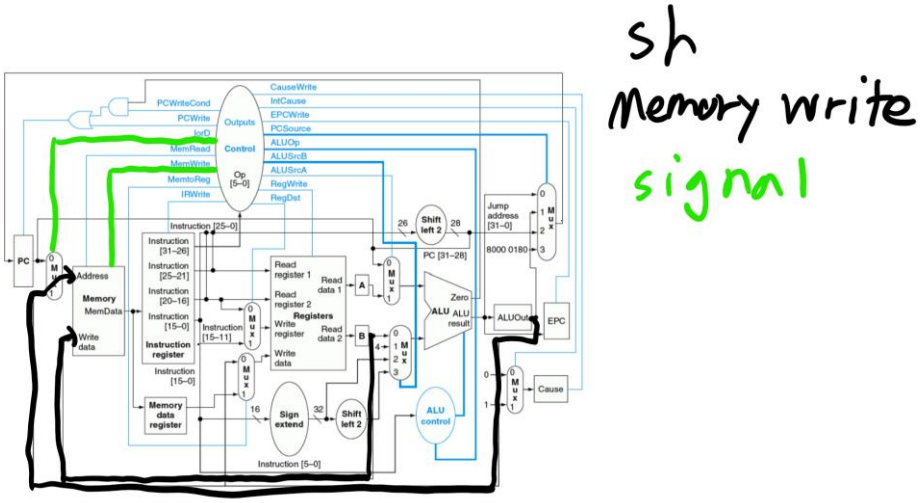
lh와 sh 명령어는 execution단계만 동일하다. 두 명령어 모두 rs와 i를 서로 add 하여 memory에 접근할 주소를 만들어내기 때문에 이 단계에서의 signal들은 동일하다.



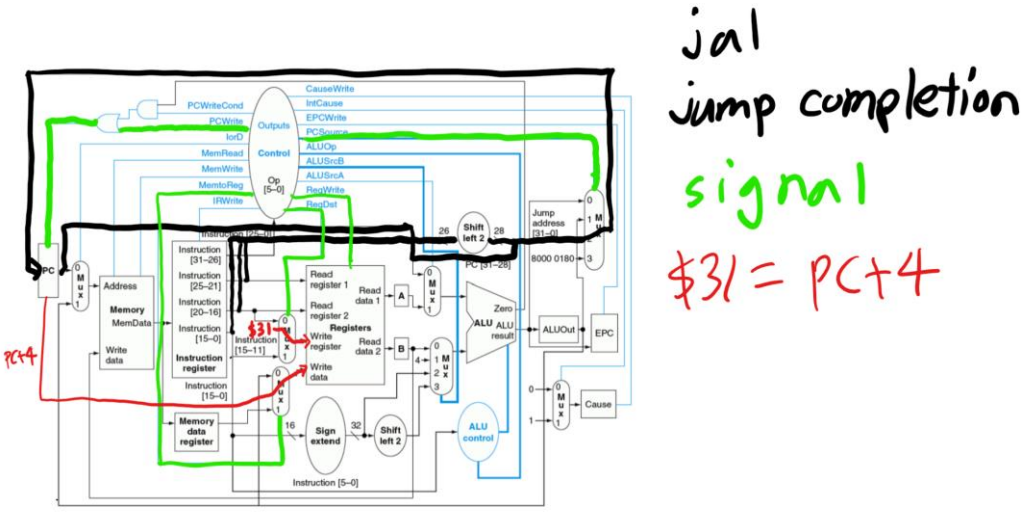
lh의 memory access단계이다. lh는 이전 단계에서 연산한 memory에 접근할 주소를 ALUOut으로부터 가져와 해당 주소가 가리키는 memory값을 MDR에 임시로 저장하게 된다. 이때 저장하는 값은 DatWidth에 의해 halfword을 extension한 값이다.



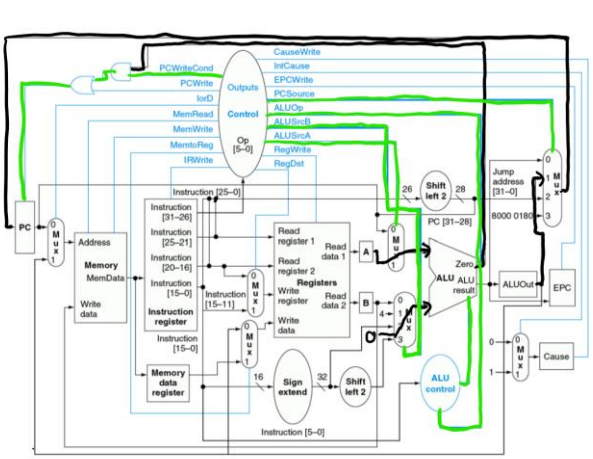
이전단계에서 MDR에 memory의 값을 저장했기 때문에 rt에 MDR의 값을 선택하여 저장한다.



sh는 memory주소를 생성한 후 lh와 다르게 바로 rt를 memory에 저장하면 되기에 한 단계를 절약할 수 있다. 따라서 ALUOut값을 memory의 address로 하여 B에 존재하는 rt를 바로 Memory에 write한다. sh 또한 halfword를 저장해야하므로 DatWidth에 의해 값이 조정된다.

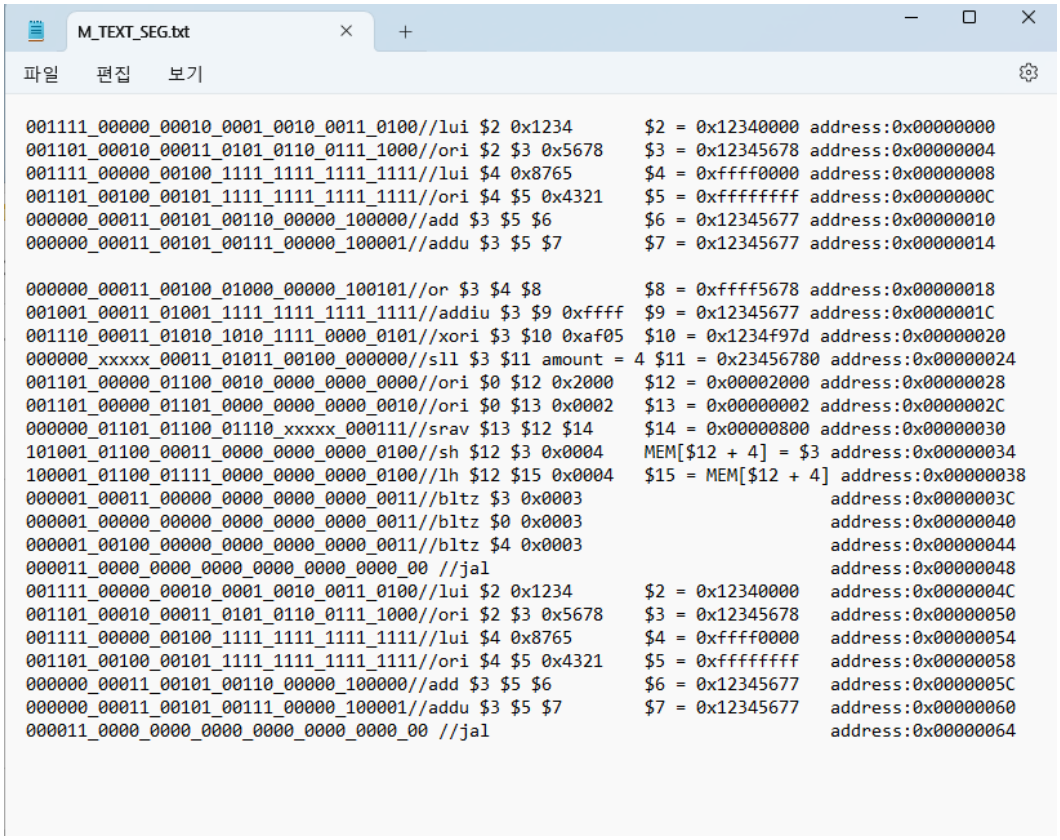


jal명령어는 unconditional branch이므로 ALU를 사용하지 않기에 해당 signal를 사용하지 않고 PC write control에 해당하는 field와 $31 레지스터에 PC값을 쓰기 위해 register field의 signal들을 사용하는 것을 확인할 수 있다. 검정색으로 표기된 부분으로 PC값과 instruction의 하위 26비트를 왼쪽으로 2비트 쉬프트 한 값을 이어붙여 새로운 PC값을 만들고 signal에 의해 해당 값을 선택하여 PC에 인가하는 것을 확인할 수 있다. 빨간색 부분은 레지스터에 PC값을 저장하는 부분이다.



bltz는 conditional branch이므로 ALU를 통한 조건확인이 필요하다. 따라서 ALU field에 있는 signal들을 사용하여 jal과 마찬가지로 PC write control field에 있는 signal들 또한 사용한다. rs값과 0을 이용해 연산을 진행하고 이를 통해 branch의 여부를 결정한다.

모든 명령어의 흐름과 핵심적으로 사용되는 signal들을 그림을 통해 설명하였다. 아래는 gtkwave를 사용하여 여러 명령어들이 어떤 흐름으로 작동되고 해당 state에서 적절한 값을 가지는지 명령어의 동작을 마치는데 state가 다른 것이 잘 반영되는지등을 검증하여 본다.

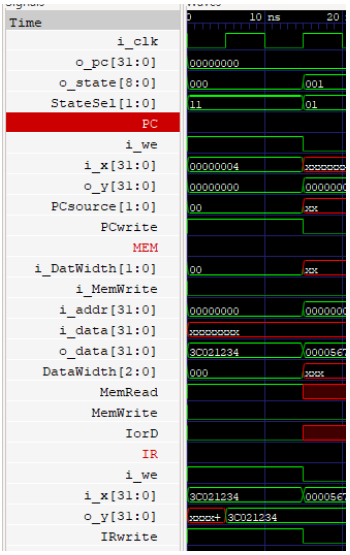


위는 예시 명령어들의 모음이다. lui ori를 register의 값을 초기에 넣어주기위해 사용하였고 addu와 실행결과를 비교하여 addu의 결과가 명확한지 판단하기 위해 add를 사용하였다. bltz는 조건에 맞게 0보다 작을 때 동작하는지 확인하기 위해 0보다 클 경우, 0일 경우, 0보다 작을경우로 나누었다.

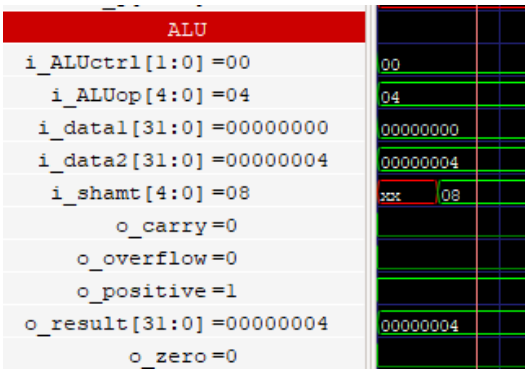
명령어들이 각각 정상적으로 동작하였을 때의 기댓값들을 명령어 옆에 명시해두었다.

먼저 **instruction fetch**단계를 검증해보자.

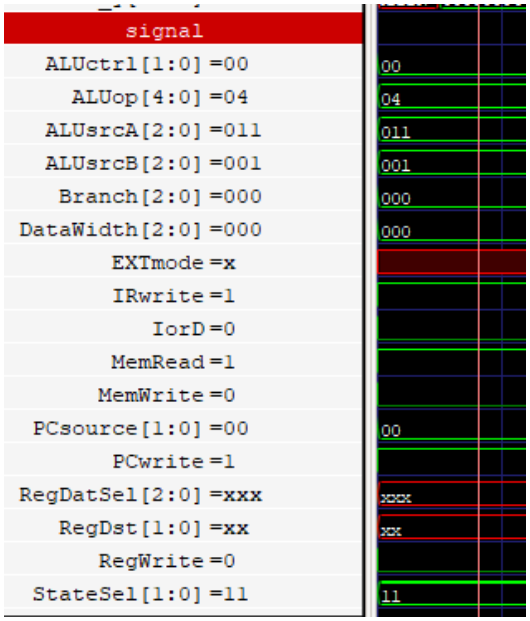
instruction fetch단계는 모든 명령어에 공통적으로 포함된 단계이며 중점적으로 봐야하는 것은 구현한 FSM대로 signal들의 값이 나오는지 PC = PC + 4를 계산하는지 PC에서 값이 나와 memory의 주소로 작용하고 해당 memory 값이 IR로 쓰여질 준비를 하는지를 체크해야한다.

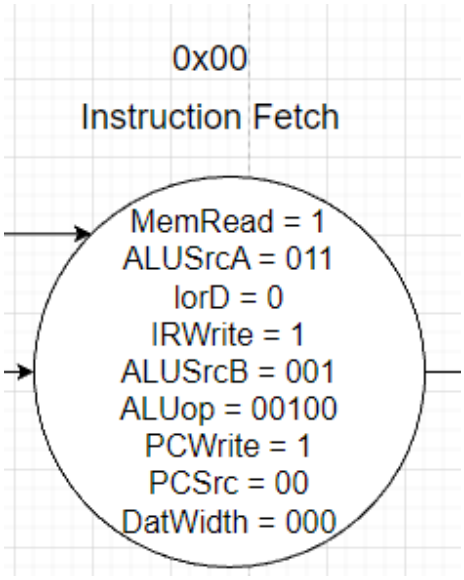


위의 사진을 보면 0x00000000에 해당하는 lui명령어를 fetch한다는 것을 알 수 있다. PC의 output o_y가 MEM의 i_addr로 인가되었고 해당 주소가 가리키는 값이 0x3C021234로 0011_1100_0000_0010_0001_0010_0011_0100으로 lui $2 0x1234로 첫번째 명령어 32비트를 잘 읽은 것을 알 수 있다. 이 값은 IR의 i_x로 전해지는 것을 보아 instruction fetch는 잘 되는 것을 알 수 있다. 그렇다면 PC = PC+4를 ALU에서 연산하는지를 확인해보자.



ALU에서 현재 PC값 00000000과 4를 더해 00000004를 만들어 낸 것을 알 수 있다. 그렇다면 signal은 올바르게 작동하는지 확인해보자.

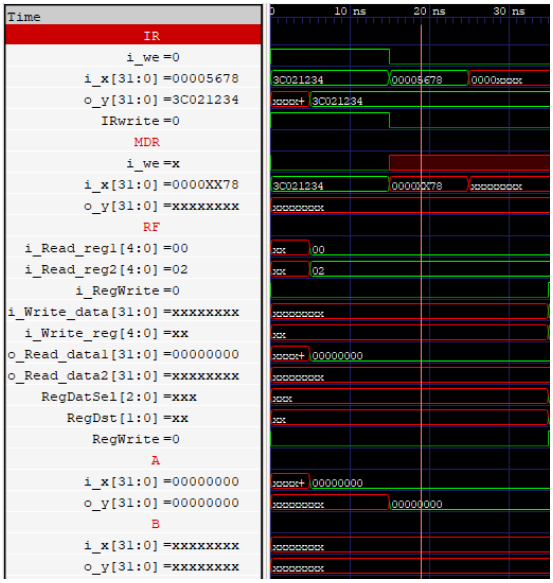




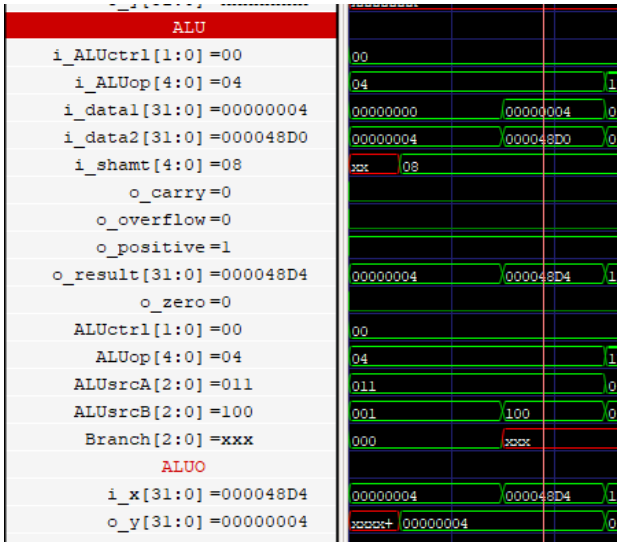
FSM에서 요구하는 signal들이 올바르게 나왔고 추가적으로 사용하지 않는 signal들은 don't care이고 write는 0이기 때문에 의미없는 값이 쓰여지지 않을 것이고 statesel또한 11로 다음 state를 실행하라는 signal이므로 0x01에 해당하는 state로 이동할 것이다.

다음은 **Decode/Register Fetch/Branch Address** 단계를 검증해본다.

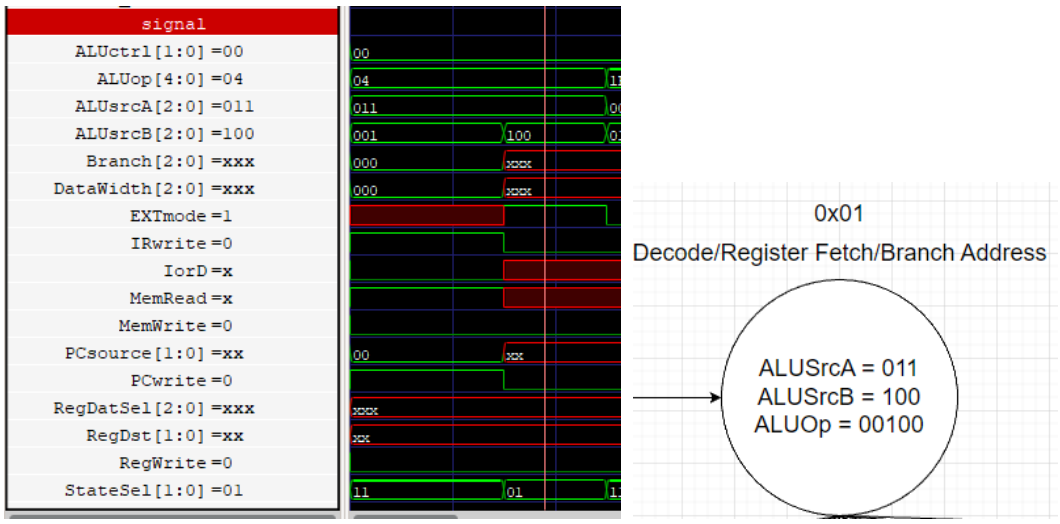
이번 단계 또한 모든 명령어가 공통적으로 포함하는 단계이며 IR에서 나온 값이 Register에 잘 전해지는지 또한 register에서 읽힌 값이 A와 B에 쓰일 준비를 하는지 그리고 branch address를 계산하여 ALUOut에 넣을 준비를 하는지 마지막으로 state에서 설정한 signal과 동일한 signal이 나타나는지 검증할 것이다.



register fetch부분이다. instruction fetch를 검증할 때 사용한 명령어의 다음 state의 사진이다. 0x3C021234라는 명령어가 비트단위로 나누어져 i_Read_reg1값을 보고 rs는 00, i_Read_reg2값을 보고 rt는 02가 잘 전달된 것을 알 수 있다. o_Read_data1에는 $0을 읽어 0, 현재 02에는 아무런 값이 존재하지 않기에 o_Read_data2에는 x가 읽혔다. 그리고 읽혀진 값은 각각 A와 B의 i_x에 전달되었다.



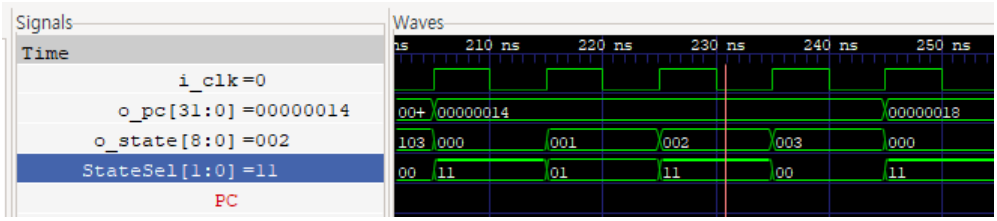
위는 ALU이다. 현재 PC+4에 0x1234를 2번 left shift한 값 0x48D0을 서로 더하여 0x48D4를 얻고 이를 ALUOut의 i_x로 전달한 것을 볼 수 있다.



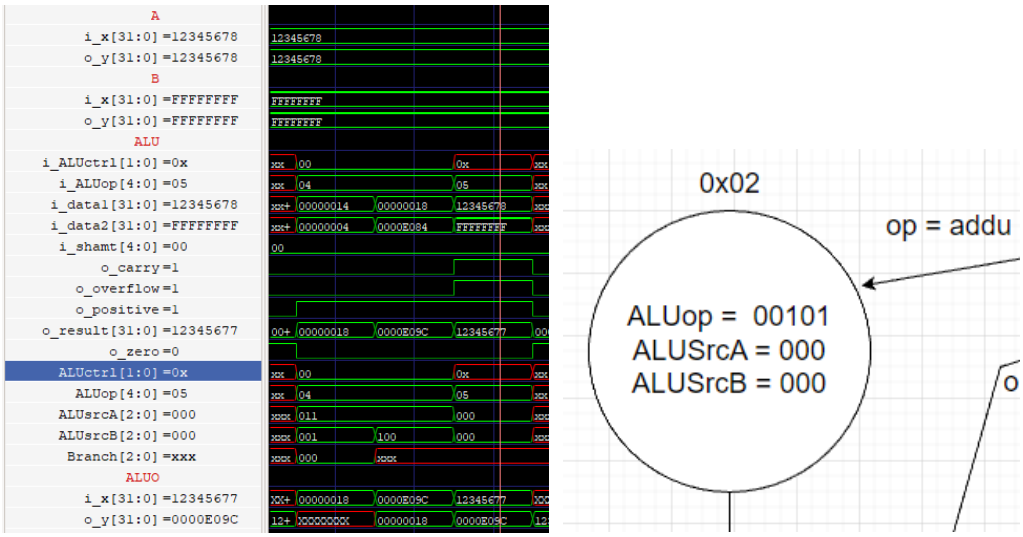
signal 또한 0x01에 해당하는 state의 조건을 만족한다.

0x00, 0x01에 해당하는 state를 올바르게 수행하는 것을 검증하였으니 이제 각 명령어마다 가지는 execution,memory access,writeback과 같은 state를 검증하면 될 것이다.

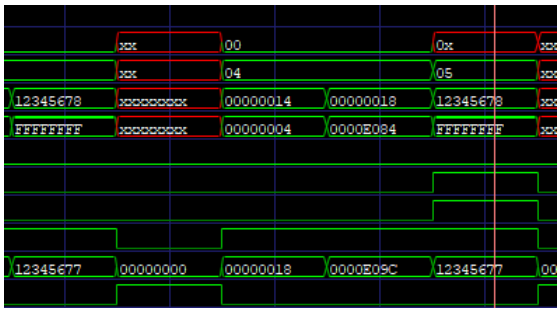
<addu>



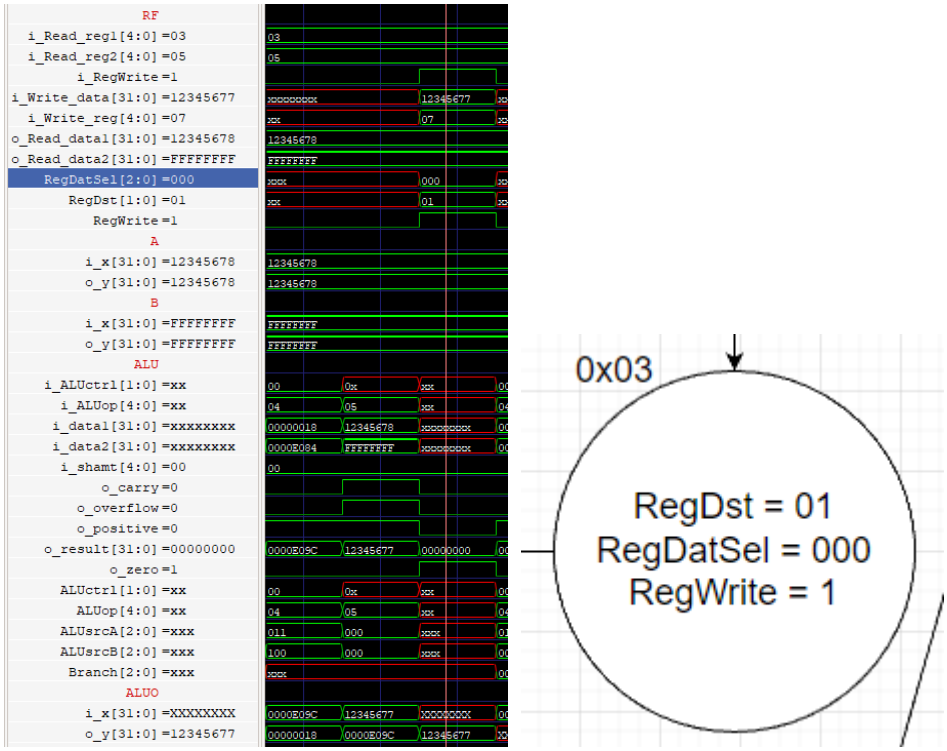
먼저 R-type으로 fetch, decode, execution, write back의 state를 거친 뒤 PC가 증가하여 다시 새로운 명령어가 실행됨을 알 수 있다.



A와 B의 값이 ALUSrcA와 ALUSrcB에 의해 선택되어 ALU의 입력으로 들어오게 되었고 이들을 unsigned add하고 그 결과가 ALUOut의 i_x에 전해진다. addu는 부호없는 add를 진행하기 때문에 0xFFFFFFFF을 양수로 판단하고 더하기 때문에 overflow가 발생하지만



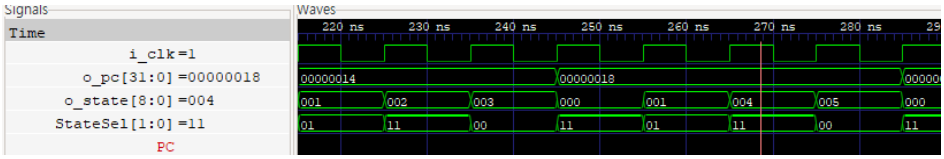
위와 같이 바로전에 실행한 add명령어의 경우 0xFFFFFFFF를 -1로 인식하여 overflow가 발생하지 않았다.



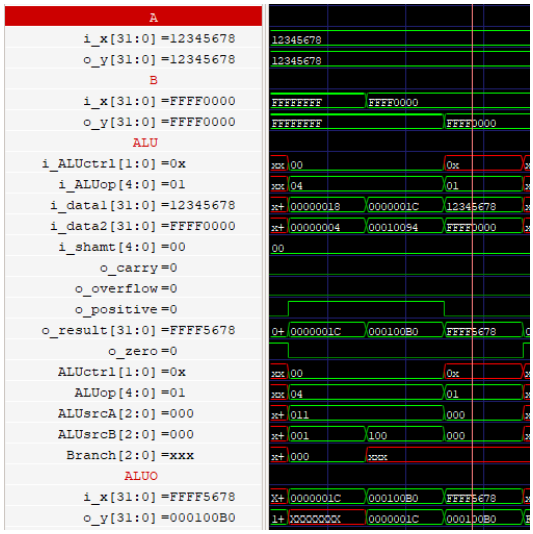
write back 단계에서는 전 단계에서 ALUOut에 입력으로 주었던 0x12345677이 쓰여졌고 해당 값이 RF의 write_data로 전해져 $7에 RegWrite가 1이므로 쓰여진다.

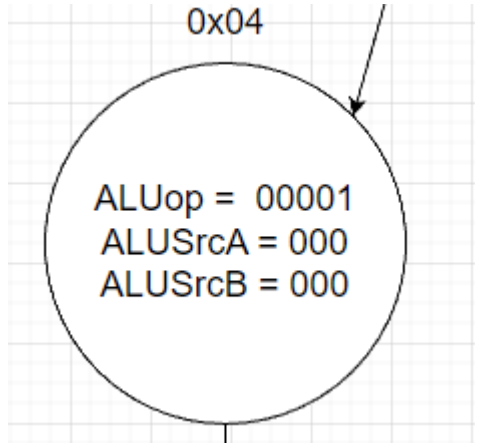
기댓값과 일치하는 결과를 얻었다.

<or>

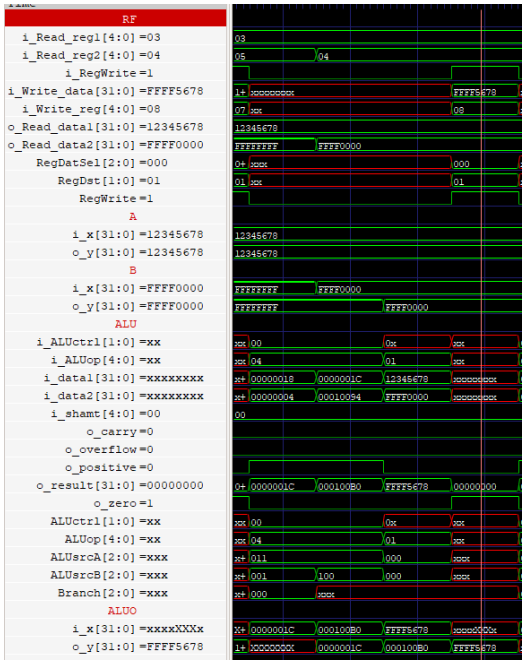


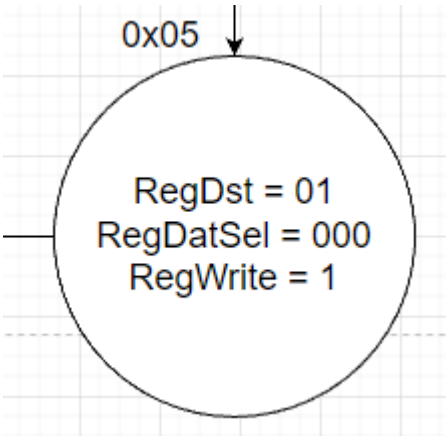
R-type으로 fetch, decode, execution, write back의 state를 거친 뒤 PC가 증가하여 다시 새로운 명령어가 실행됨을 알 수 있다.





addu와 마찬가지로 ALUSrcA, ALUSrcB를 통해 A와 B값이 ALU의 입력으로 들어가는 것을 알 수 있다. 이들은 ALUop에 의해 or연산을 진행하고 결과는 ALUOut으로 전해진다.

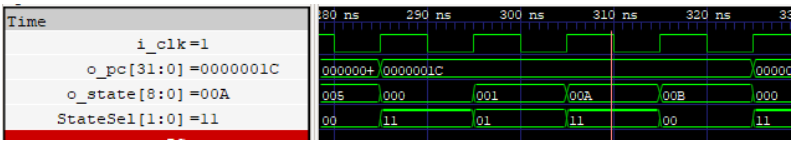




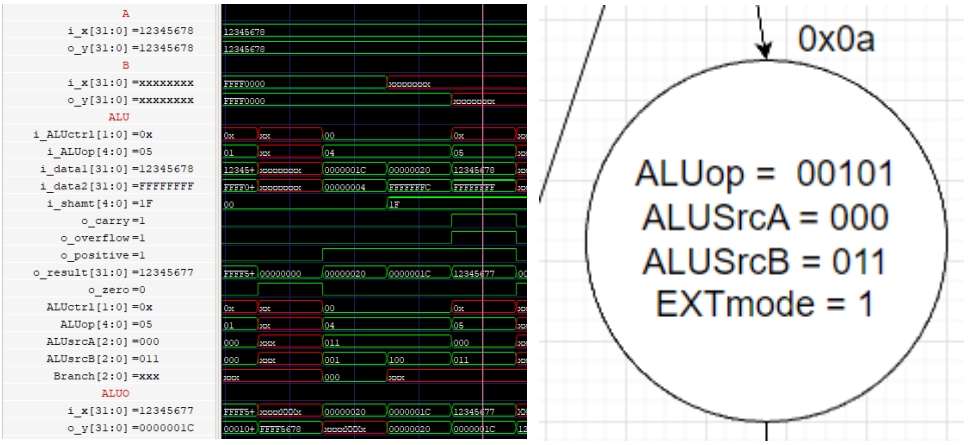
write back부분이다. ALUOut의 값이 $8에 쓰이는 것을 알 수 있다.

기댓값과 동일한 결과를 얻은 것을 확인한다.

<addiu>



fetch, decode, execution, write back의 state를 거친 뒤 PC가 증가하여 다시 새로운 명령어가 실행됨을 알 수 있다.

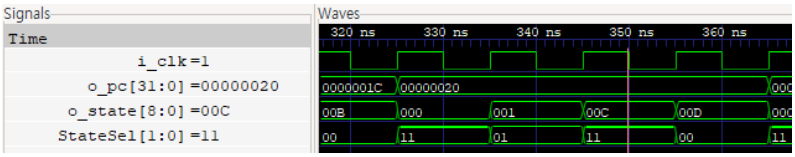


execution 단계로 rs와 sign extension을 진행한 immediate값을 addiu연산하여 ALUOut에 전달한다. 결과적으로 addu와 동일한 피연산자를 가졌을 때 동일한 결과값을 얻게 된다.

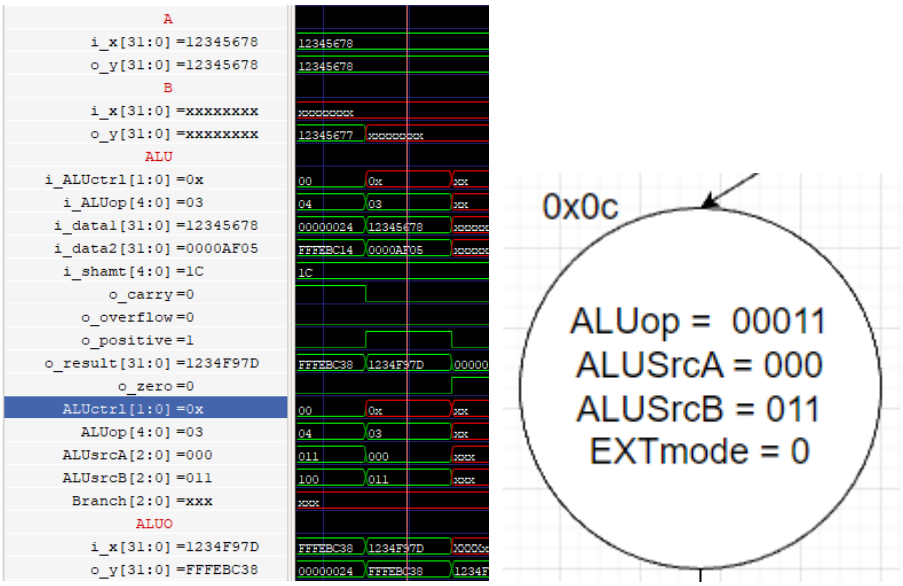
ALUOut에서 나오는 값이 $9에 저장되는 것을 확인할 수 있다.

기댓값과 동일한 결과를 얻은 것을 확인한다.

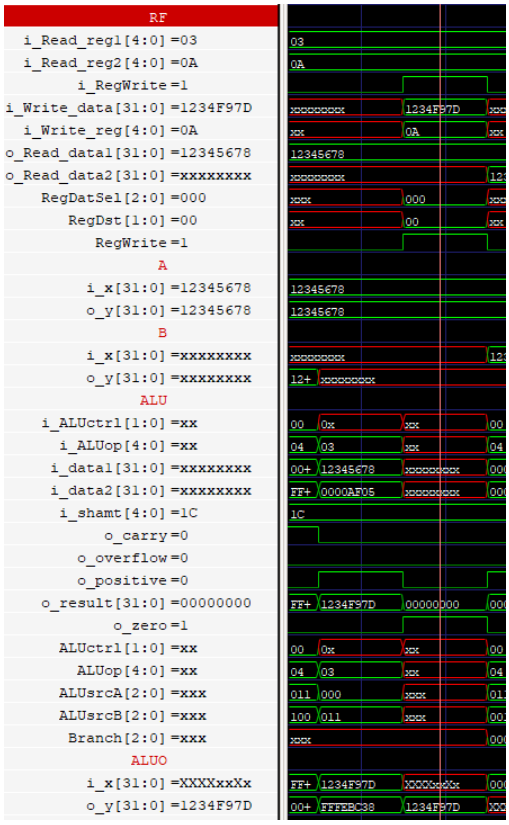
<xori>

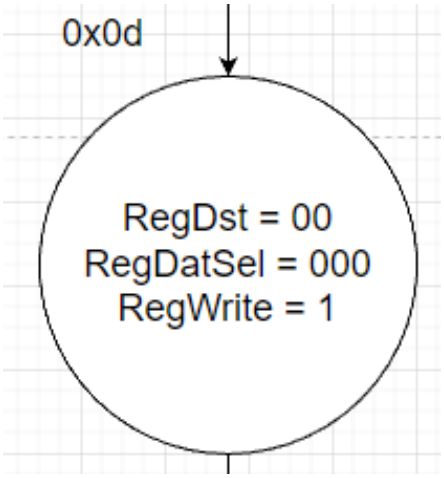


fetch, decode, execution, write back의 state를 거친 뒤 PC가 증가하여 다시 새로운 명령어가 실행됨을 알 수 있다.



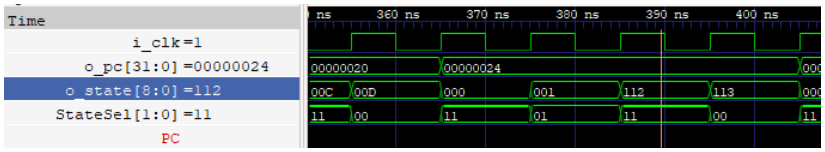
execution 단계로 rs와 zero extension한 immediate값을 xori연산하여 ALUOut에 넣는 모습이다.



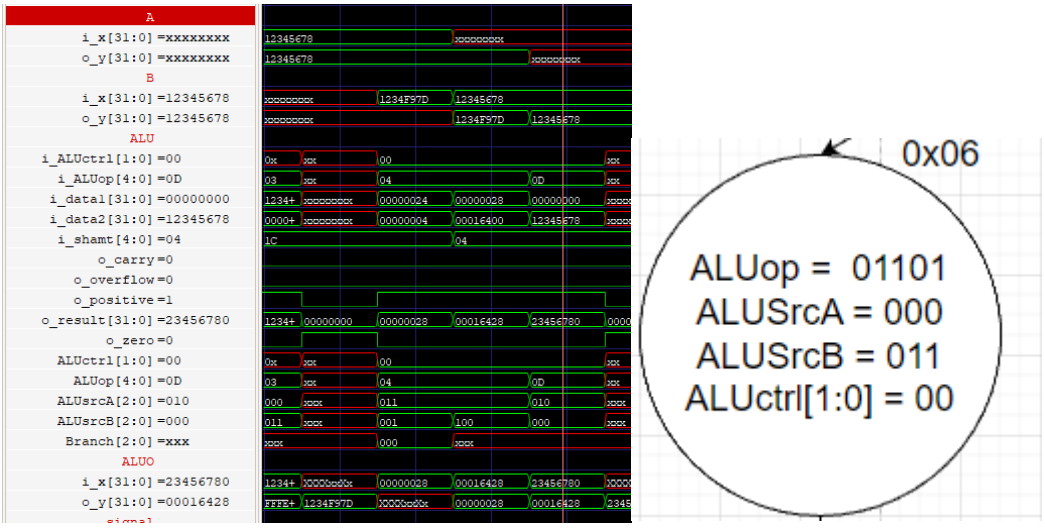


write back단계로 ALUOut의 값이 $10에 저장되는 것을 확인한다. 기댓값과 동일한 결과값을 얻었다.

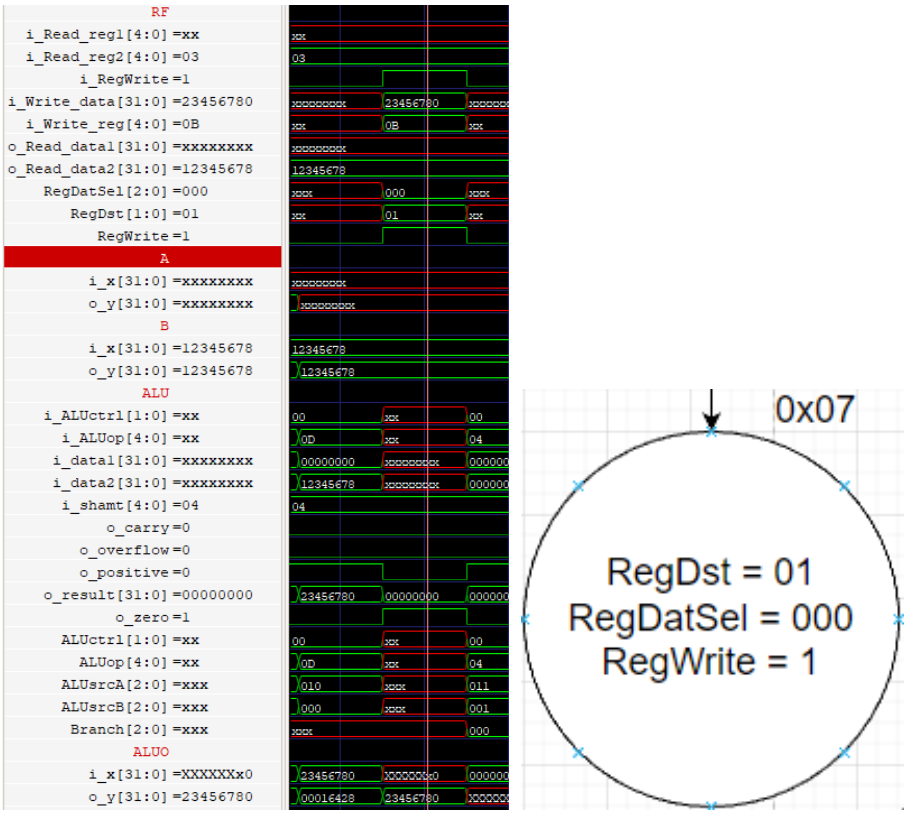
<sll>



fetch, decode, execution, write back의 state를 거친 뒤 PC가 증가하여 다시 새로운 명령어가 실행됨을 알 수 있다.

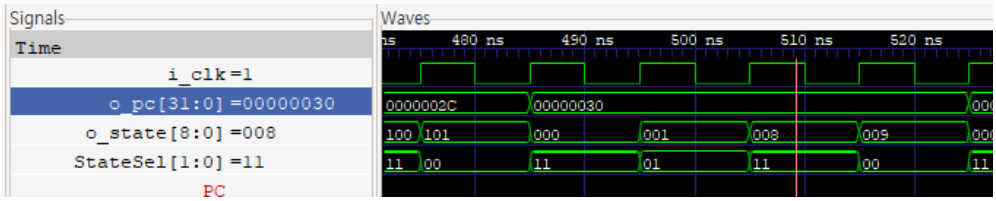


execution 단계로 rt의 값을 shift amount값만큼 shift left한 후 ALUOut에 저장하는 것을 확인할 수 있다.

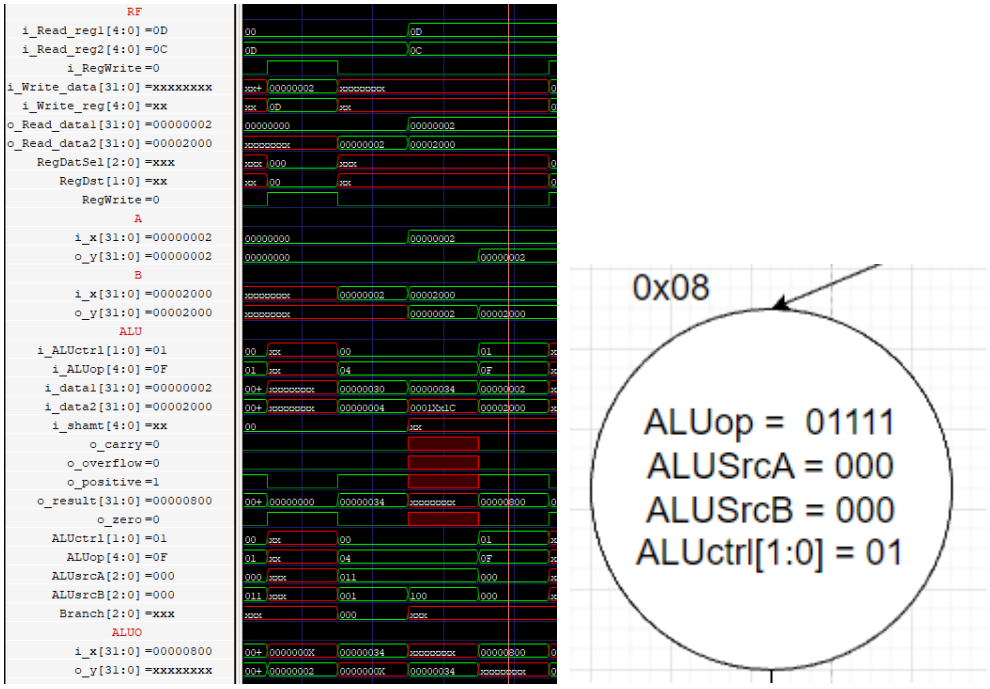


write back 단계로 ALUOut의 값을 $11에 저장한다. 기댓값과 같은 결과값을 얻었다.

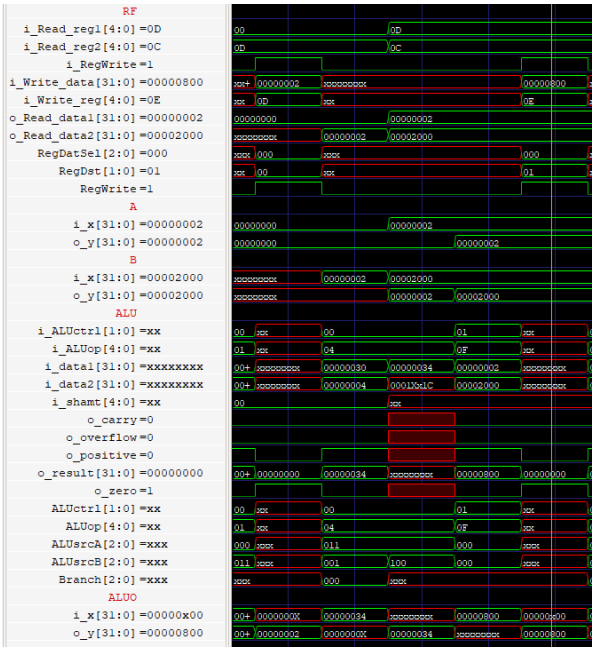
<srav>

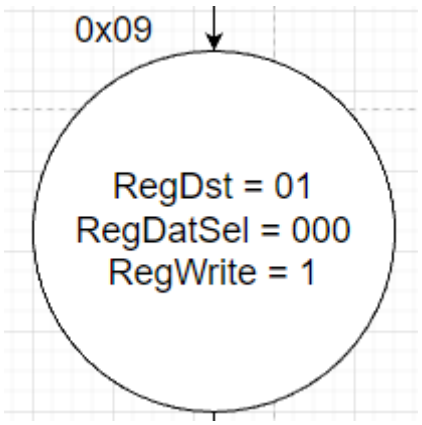


fetch, decode, execution, write back의 state를 거친 뒤 PC가 증가하여 다시 새로운 명령어가 실행됨을 알 수 있다.



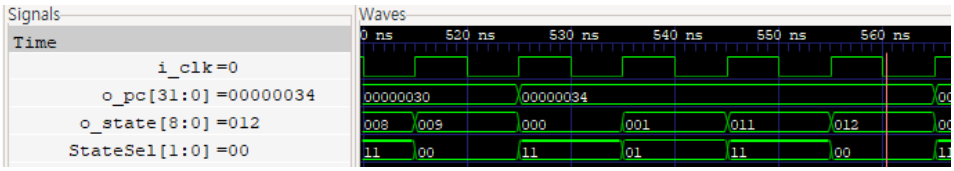
sll과의 차이점은 sll은 shift amount값을 이용하여 쉬프트 연산을 진행하지만 srav는 레지스터값을 이용하여 쉬프트연산을 하기 때문에 A값이 존재한다. B의 값을 A값만큼 right arithmetic shift를 진행한 후 ALUOut에 전달한다.



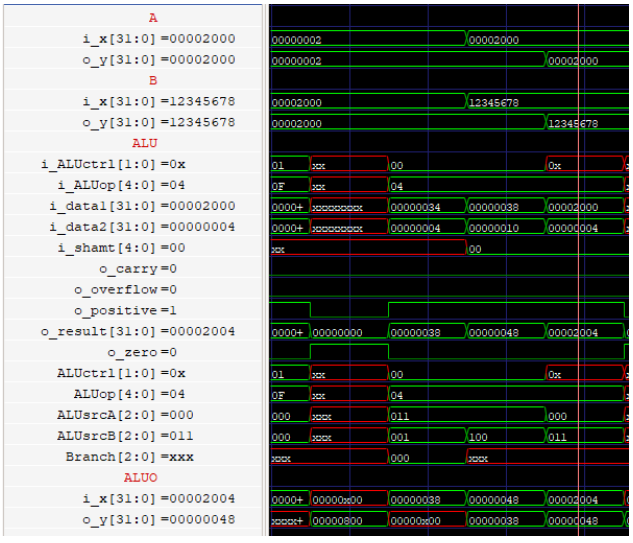


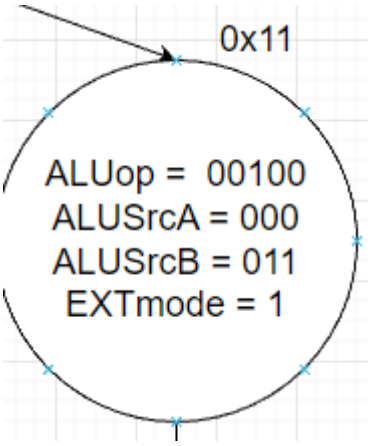
그 후 ALUOut의 값을 $14에 저장하게 된다. 기댓값과 결과값이 동일하다.

<sh>



fetch, decode, execution, memory access의 state를 거친 뒤 PC가 증가하여 다시 새로운 명령어가 실행됨을 알 수 있다.

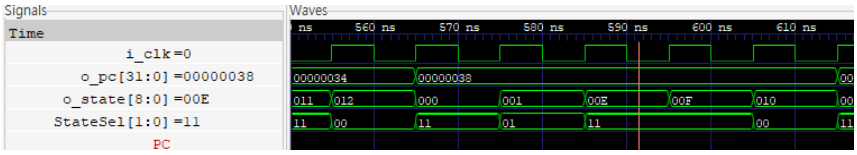




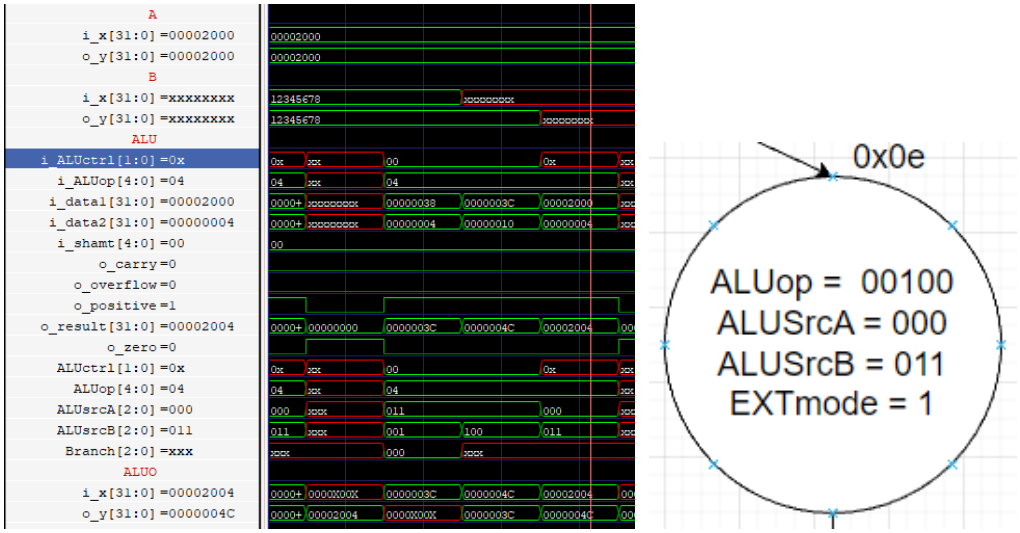
우선 rs와 상수값을 서로 ALU에서 add연산을 진행한 후 ALUOut에 저장한다.

ALUOut에 있는 값을 Memory address로 전달하고 위에서 B에 존재했던 rt값을 write data로 전달한다. 추가적으로 DatWidth는 010이므로 halfword만 저장하게 된다.

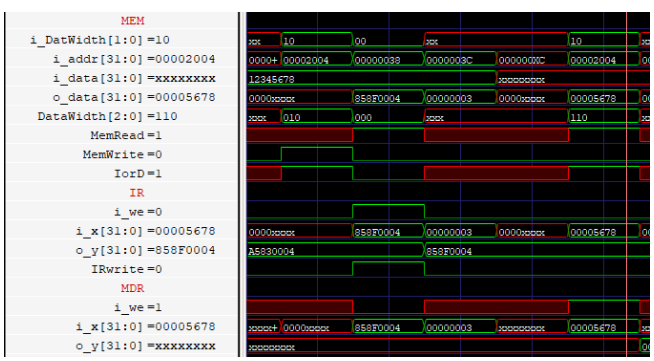
<lh>

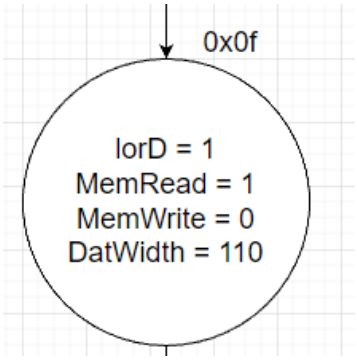


fetch, decode, execution, memory access, write back의 state를 거친 뒤 PC가 증가하여 다시 새로운 명령어가 실행됨을 알 수 있다

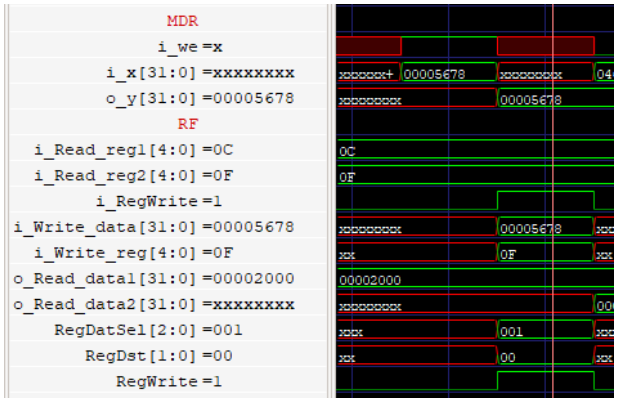


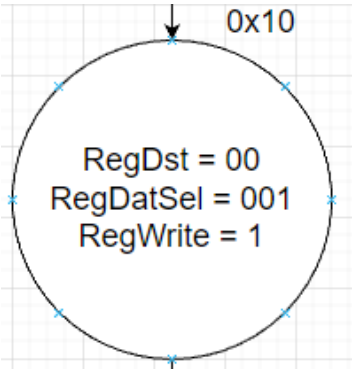
sh와 마찬가지로 A값과 상수를 이용하여 add연산을 통해 memory에 접근할 주소를 생성한다. 결과는 ALUOut에 저장한다.





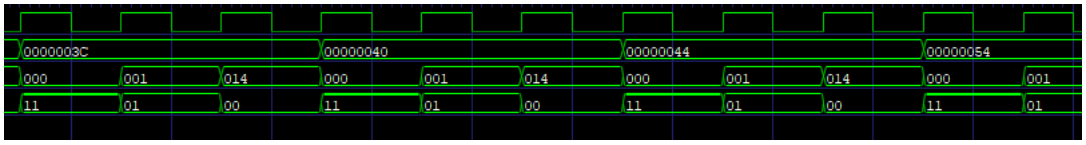
ALUOut의 값을 Memory의 address로 보낸 후 가리키는 값을 DatWidth가 110이므로 halfword만큼 읽은 후 sign extension을 진행한다. 그 후 해당 값을 MDR에 저장한다.



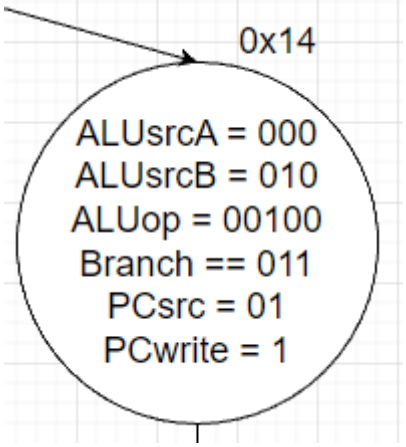


MDR의 값을 Register의 Write_data로 보내고 $15에 저장한다.

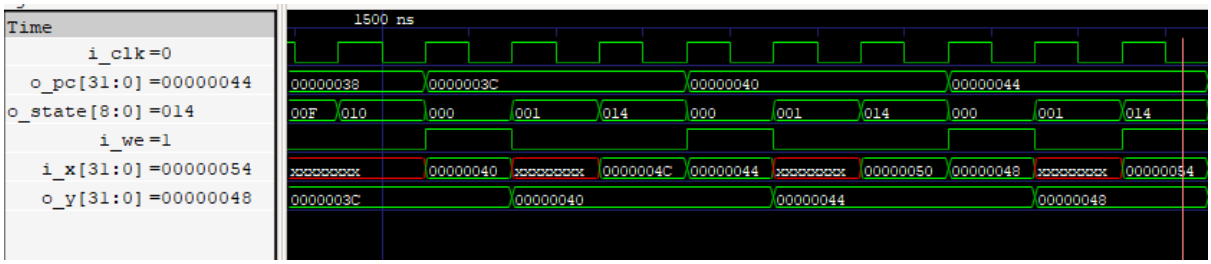
<bltz>



양수일 때, 0일 때, 음수일 때의 경우이다. 음수일 때만 branch를 시행했음을 알 수 있다. 추가적으로 branch 명령은 주소를 decode할 때 계산해두기 때문에 decode후 한 state만에 명령어가 처리된다.

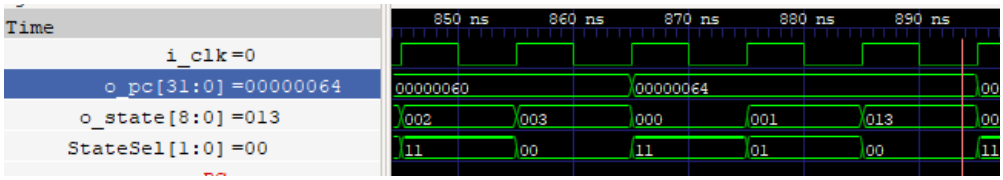


주황색 선이 가리키는 부분부터 3칸 단위로 branch가 일어날지 말지를 결정하는 ALU 연산이 일어나는 곳이다. 첫번째는 0x12345678, 두번째는 0, 세번째는 0xFFFF0000인 rs와 0을 비교하게 되고 add연산을 사용하였다. branch는 둘을 더하였을 때 결과가 negative라면 branch를 진행하게 끔 구성하였다. PCwrite는 모두 1로 셋되어있으나 아래 사진을 보면

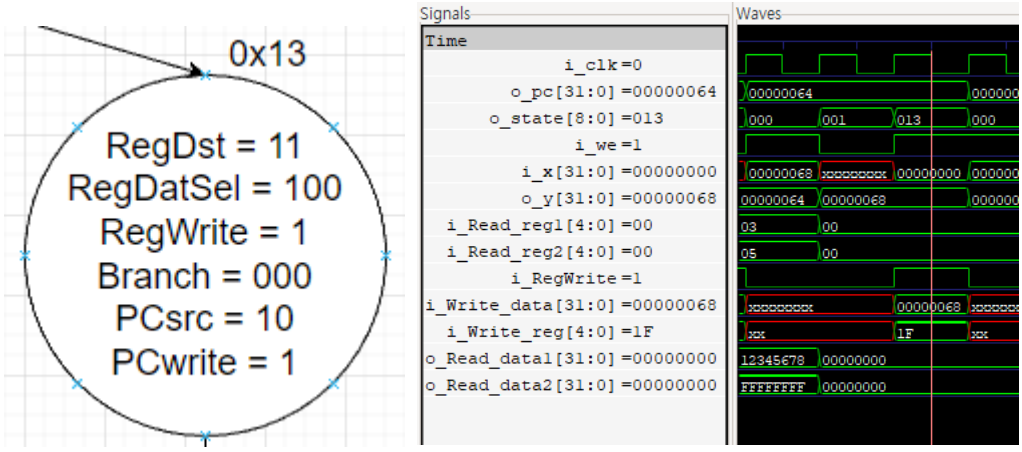


PC의 i_we값은 branch조건을 만족하였을 때만 1로 셋되어있는 것을 볼 수 있다. 이는 pc_write와 ALU의 연산결과를 함께 combinational logic으로 묶어 i_we의 값을 결정하게끔 했음을 알 수 있다.

<jal>



branch와 마찬가지로 decode이후 한 state만에 결과를 알 수 있고 이는 비트를 쉬프트하고 이어붙이기 때문에 ALU같은 unit을 사용하지 않기에 가능하다.



PC+4값을 $31에 써야하므로 위와 같은 신호들을 가지고 추가적으로 PCsrc로 jump명령어를 사용하여 unconditional한 분기를 한다.

모든 명령어들의 실행결과를 검증하여 보고 구현한 FSM의 각 state에 정의된 signal들 또한 잘 나타나면서 unit들을 control하는 것을 확인하였다. 명령어 예시를 모두 돌려본 결과 reg_dump.txt를 통해 원하는 기댓값을 그대로 얻어낸 것을 확인할 수 있었다. 아래는 각 파일들의 값들이다.

```
001111_00000_00010_0001_0010_0011_0100//lui $2 0x1234       $2 = 0x12340000 address:0x00000000
001101_00010_00011_0101_0110_0111_1000//ori $2 $3 0x5678    $3 = 0x12345678 address:0x00000004
001111_00000_00100_1111_1111_1111_1111//lui $4 0x8765       $4 = 0xffff0000 address:0x00000008
001101_00100_00101_1111_1111_1111_1111//ori $4 $5 0x4321    $5 = 0xffffffff address:0x0000000C
000000_00011_00101_00110_00000_100000//add $3 $5 $6         $6 = 0x12345677 address:0x00000010
000000_00011_00101_00111_00000_100001//addu $3 $5 $7        $7 = 0x12345677 address:0x00000014
000000_00011_00100_01000_00000_100101//or $3 $4 $8          $8 = 0xffff5678 address:0x00000018
001001_00011_01001_1111_1111_1111_1111//addiu $3 $9 0xffff  $9 = 0x12345677 address:0x0000001C
001110_00011_01010_1010_1111_0000_0101//xori $3 $10 0xaf05  $10 = 0x1234f97d address:0x00000020
000000_xxxxx_00011_01011_00100_000000//sll $3 $11 amount = 4 $11 = 0x23456780 address:0x00000024
001101_00000_01100_0010_0000_0000_0000//ori $0 $12 0x2000   $12 = 0x00002000 address:0x00000028
001101_00000_01101_0000_0000_0000_0010//ori $0 $13 0x0002   $13 = 0x00000002 address:0x0000002C
000000_01101_01100_01110_xxxxx_000111//srav $13 $12 $14     $14 = 0x00000800 address:0x00000030
101001_01100_00011_0000_0000_0000_0100//sh $12 $3 0x0004    MEM[$12 + 4] = $3 address:0x00000034
100001_01100_01111_0000_0000_0000_0100//lh $12 $15 0x0004   $15 = MEM[$12 + 4] address:0x00000038
000001_00011_00000_0000_0000_0000_0011//bltz $3 0x0003                        address:0x0000003C
000001_00000_00000_0000_0000_0000_0011//bltz $0 0x0003                        address:0x00000040
000001_00100_00000_0000_0000_0000_0011//bltz $4 0x0003                        address:0x00000044
000011_0000_0000_0000_0000_0000_0000_00 //jal                                 address:0x00000048
001111_00000_00010_0001_0010_0011_0100//lui $2 0x1234       $2 = 0x12340000   address:0x0000004C
001101_00010_00011_0101_0110_0111_1000//ori $2 $3 0x5678    $3 = 0x12345678   address:0x00000050
001111_00000_00100_1111_1111_1111_1111//lui $4 0x8765       $4 = 0xffff0000   address:0x00000054
001101_00100_00101_1111_1111_1111_1111//ori $4 $5 0x4321    $5 = 0xffffffff   address:0x00000058
000000_00011_00101_00110_00000_100000//add $3 $5 $6         $6 = 0x12345677   address:0x0000005C
000000_00011_00101_00111_00000_100001//addu $3 $5 $7        $7 = 0x12345677   address:0x00000060
000011_0000_0000_0000_0000_0000_0000_00 //jal                                 address:0x00000064
```

M_Text_Seg.txt

```
00000000 : 00000000_00000000_00000000_00000000 : 00000000
00000001 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000002 : 00010010_00110100_00000000_00000000 : 12340000
00000003 : 00010010_00110100_01010110_01111000 : 12345678
00000004 : 11111111_11111111_00000000_00000000 : ffff0000
00000005 : 11111111_11111111_11111111_11111111 : ffffffff
00000006 : 00010010_00110100_01010110_01110111 : 12345677
00000007 : 00010010_00110100_01010110_01110111 : 12345677
00000008 : 11111111_11111111_01010110_01111000 : ffff5678
00000009 : 00010010_00110100_01010110_01110111 : 12345677
0000000a : 00010010_00110100_11111001_01111101 : 1234f97d
0000000b : 00100011_01000101_01100111_10000000 : 23456780
0000000c : 00000000_00000000_00100000_00000000 : 00002000
0000000d : 00000000_00000000_00000000_00000010 : 00000002
0000000e : 00000000_00000000_00001000_00000000 : 00000800
0000000f : 00000000_00000000_01010110_01111000 : 00005678
```

reg_dump.txt

```
00000800 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000801 : xxxxxxxx_xxxxxxxx_01010110_01111000 : xxxx5678
```

mem_dump.txt

<Conclusion>

이번 프로젝트는 Multi Cycle CPU를 microprogramming을 통해 구현해보는 것이었다. 주어진 signal들을 연관되어 있는 것들끼리 field로 나누고 field를 기준으로 각 명령어마다 state에서 어떤 unit을 control 해야 하는 가에 대해 생각하면서 구현하니 수월하게 진행할 수 있었다. 또한 Dispatch를 통해 decode이후 state의 분기점을 구현하니 깔끔했다. 한가지 아쉬운 점은 dispatch가 추가적으로 주어졌다면 R-type이나 lh,sh같이 execution이 공통되거나 write back이 공통된 신호들을 가진 state들을 묶어서 더욱 간결한 FSM을 구현할 수 있지 않았을까 라는 생각을 했다. 하나의 instruction을 처리할 때 functional unit들을 한번밖에 사용할 수 없었던 single cycle과는 달리 multi cycle은 사용하지 않고 있는 unit들을 재사용이 가능하다는 점에서 개선된 점을 발견할 수 있었다.