컴퓨터구조
Computer Architecture – Project #2
MIPS Multi-Cycle CPU Implementation

담당교수 : 이성원 교수님 (월3 수4)

2021202003 강준우

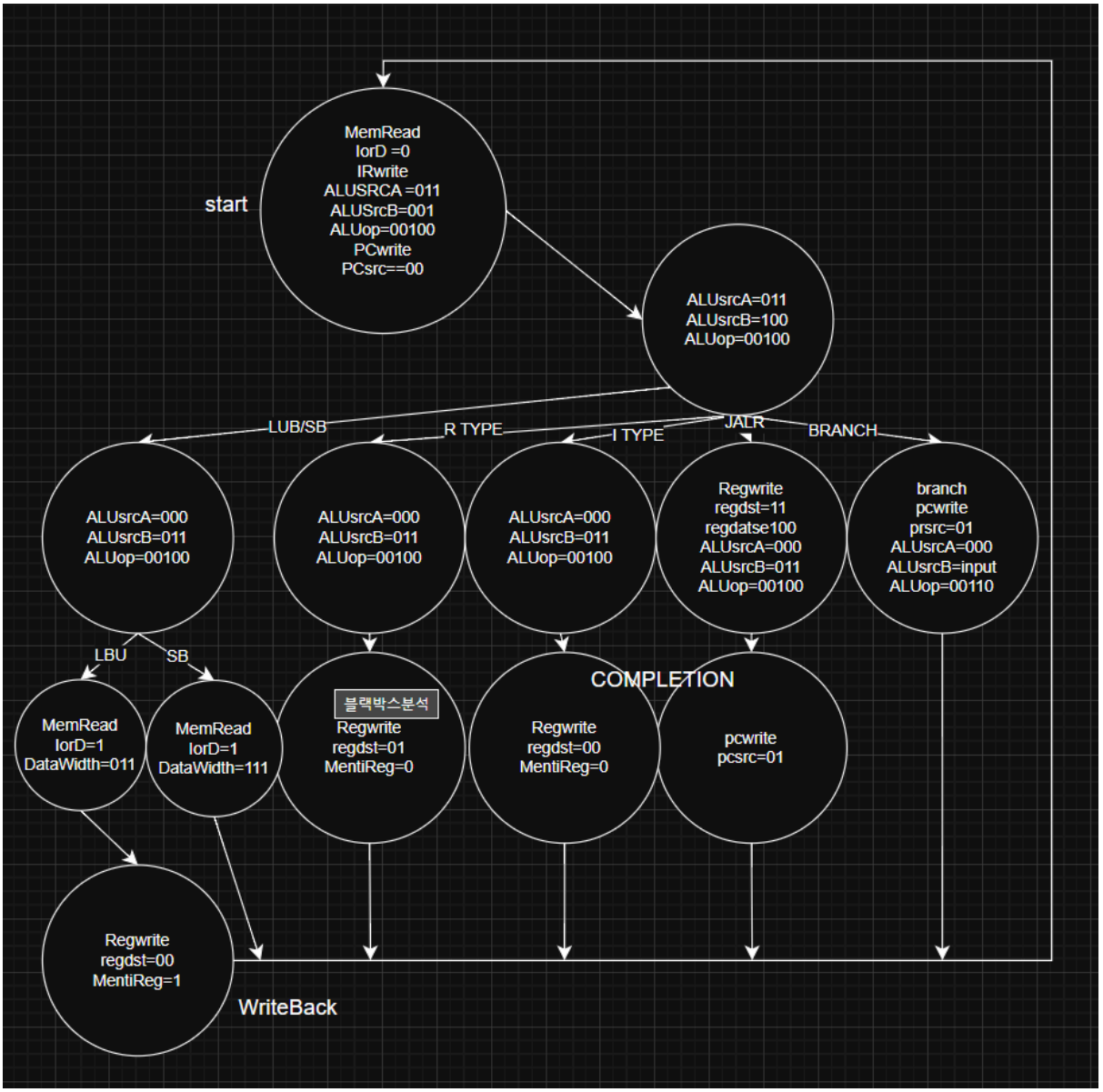
Introduction

저번 과제를 통해서 mips 를 구현해보았고 이번 과제에서는 명령어를 추가하여 더 많은 기능을 구현하도록 설계하는 것입니다. 이번 과제에서는 10개의 명령어에 대한 설명 및 구현현과 PLA 유닛에 대한 구현을 포함하여 정리합니다.

명령어 설명

고찰에서 언급한 10개의 명령어는 다음과 같습니다.
AND, NOR, ADDI, SLTU, SRL, SH, LB, BNE, BGEZ, JALR

구현한 Multi Cycle CPU FSM Diagram



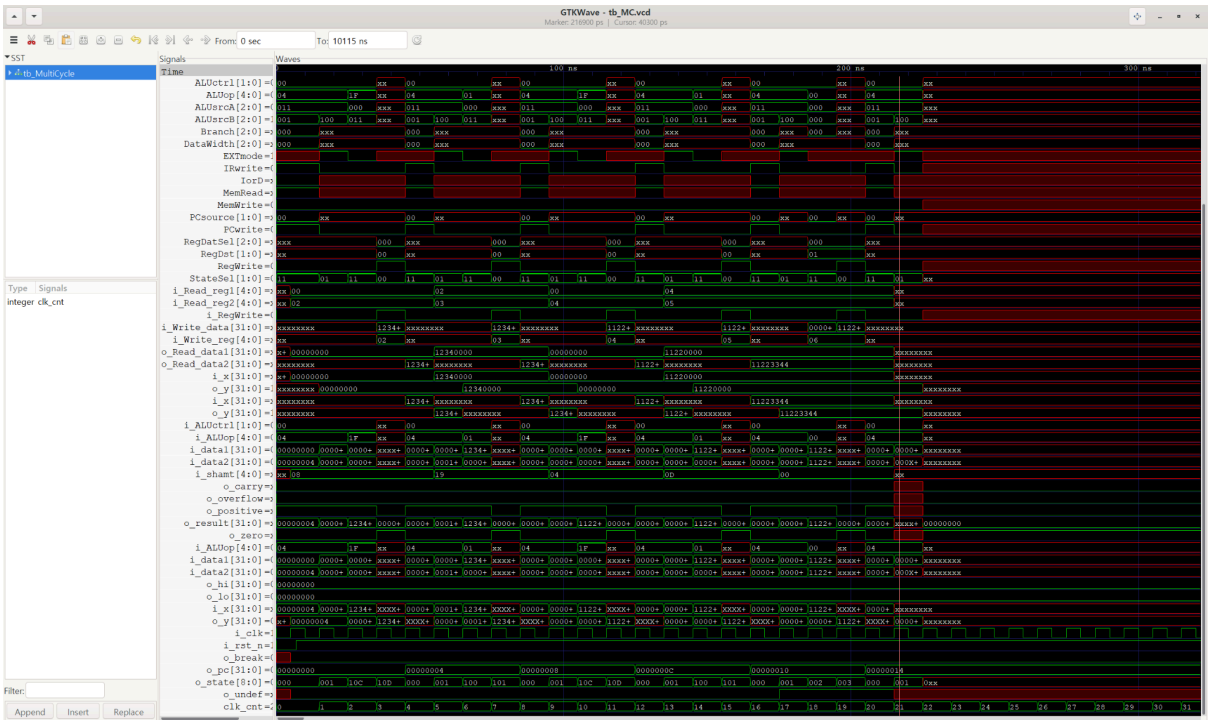
and

and 명령어는 각 자리에서 두 값이 모두 1일때만 1을 반환하는 로직을 가집니다.
x_x_0_xxx_0_01_000_0_x_000_000_00000_00_000_00_0_xxxxxxxx_11 // 0x02: AND A B
메모리 접근이 필요없고 X X 0 XXX, I타입이 아닌 R타입이라 0 RD를 사용하여 01 Write ALUOut to Register file 로 000 기록하는 단계가 아니므로 이후 0을 사용합니다. A B레지스터 사용하므로 000 000 이사용됩니다. 또한 OP코드에서 00000이고 이후 AND SAVE단계가 있으므로 11을 사용합니다.
x_x_0_xxx_0_01_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00 // 0x03: AND SAVE
RD를 사용하여 01 그리고 종료되므로 00을 마지막에 부여합니다.

2번에서 수행하니 1_00000010 // FN 100100 and 숫자를 지정했습니다,

00111100_00000010_00010010_00110100 //lui $2, 0x1234 $2 = 0x12340000
00110100_01000011_01010110_01111000 //ori $3, $2, 0x5678 $3 = 0x12345678
00111100_00000100_00010001_00100010 //lui $4, 0x1122 $4 = 0x11220000
00110100_10000101_00110011_01000100 //ori $5, $4, 0x3344 $5 = 0x11223344

00000000_10000101_00110000_00100100 // and $6, $4, $5
4번과 5번을 and연산을 통해 11220000를 6번에 저장하길 예상합니다.



결과와 같이 6번째 레지스터에서 결과를 저장하는 모습입니다.
또한 한번의 pc에서 and비교와 저장에 따른 클럭이 나뉜 모습입니다.

NOR

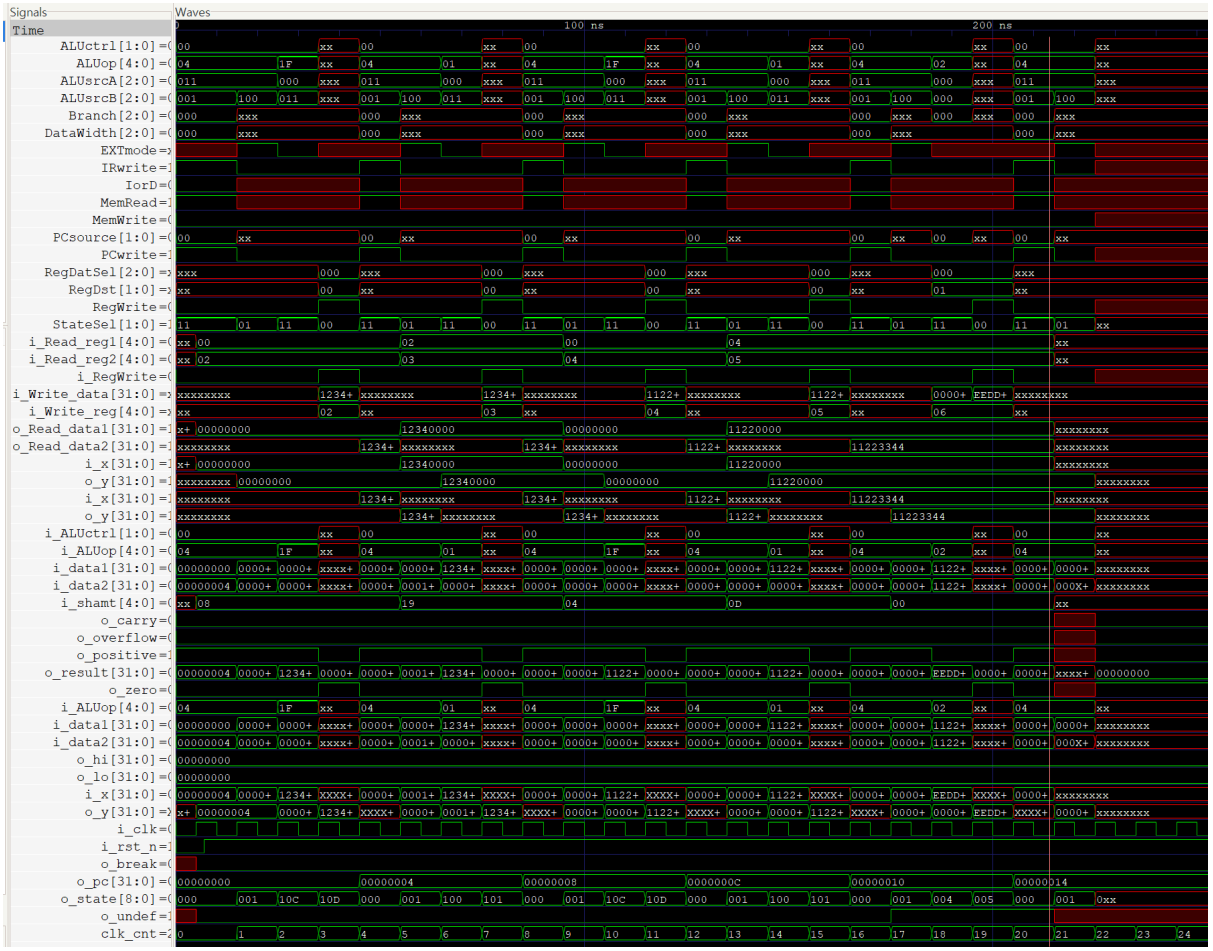
NOR 명령어는 두값이 모두 1일때만 0을 반환하는 명령어입니다.
x_x_0_xxx_0_01_000_0_x_000_000_00010_00_000_00_0_xxxxxxxx_11 // 0x04: NOR A B

AND와 동일하지만 OP부분만 수정됩니다.
x_x_0_xxx_0_01_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00 // 0x05: NOR SAVE
AND SAVE와 동일합니다.

다음과 같이 4번에에 설정해두었습니다. 1_00000100 // FN 100111 nor

00111100_00000010_00010010_00110100 //lui $2, 0x1234 $2 = 0x12340000
00110100_01000011_01010110_01111000 //ori $3, $2, 0x5678 $3 = 0x12345678
00111100_00000100_00010001_00100010 //lui $4, 0x1122 $4 = 0x11220000
00110100_10000101_00110011_01000100 //ori $5, $4, 0x3344 $5 = 0x11223344
00000000_10000101_00110000_00100111 // nor $6, $4, $5

11101110_11011101_11001100_10111011를 예상해봅니다.



사진과 같이 nor에서 결과를 eeddccbb로 확인할 수 있습니다.

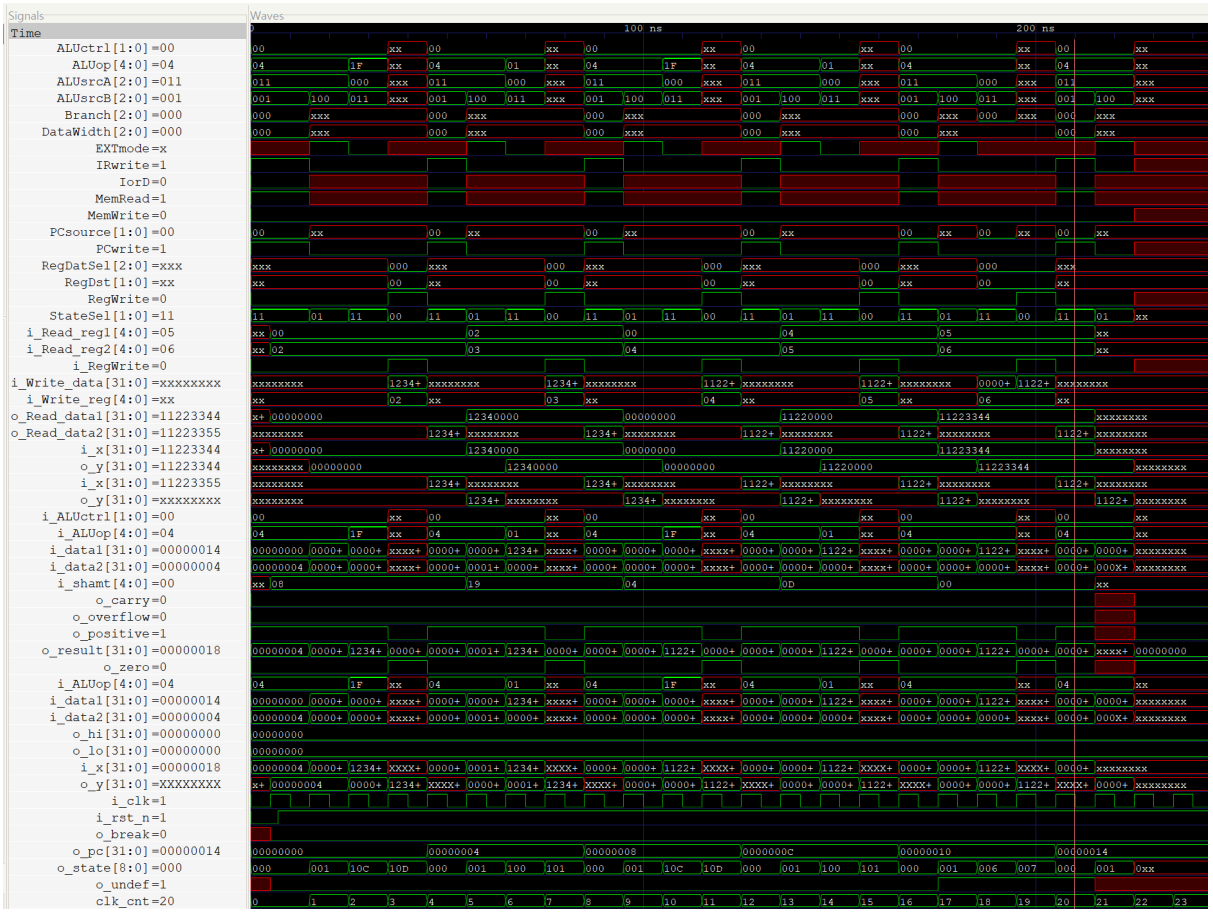
ADDI

ADDI 명령어는 즉시 덧셈을 구현합니다다.

x_x_0_xxx_0_00_000_0_x_000_011_00100_00_000_00_0_xxxxxxxx_11 // 0x06: ADDI num
x_x_0_xxx_0_00_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00 // 0x07: ADDI SAVE

이전의 명령어들과 다르게 I타입의 명렁어라 rt를 목적지로 사용하므로 00으로 regdst를 설정했습니다. ALUsrcB는 즉시값으로 011을 사용했습니다. 결과 저장에서도 rt를 지정하게 regdst를 설정했습니다.
6번째에 작동하게 설정했습니다.
1_00000110 // OP 001000 addi

[이전의 명령어들 동일]
00100000_10100110_00000000_00010001 // addi $6, $5, 10001
해당 명령어를 통해 11223344를 11223355로 더해지기를 예상합니다.



11223355로 변한 모습을 확인할 수 있었습니다.

SLTU

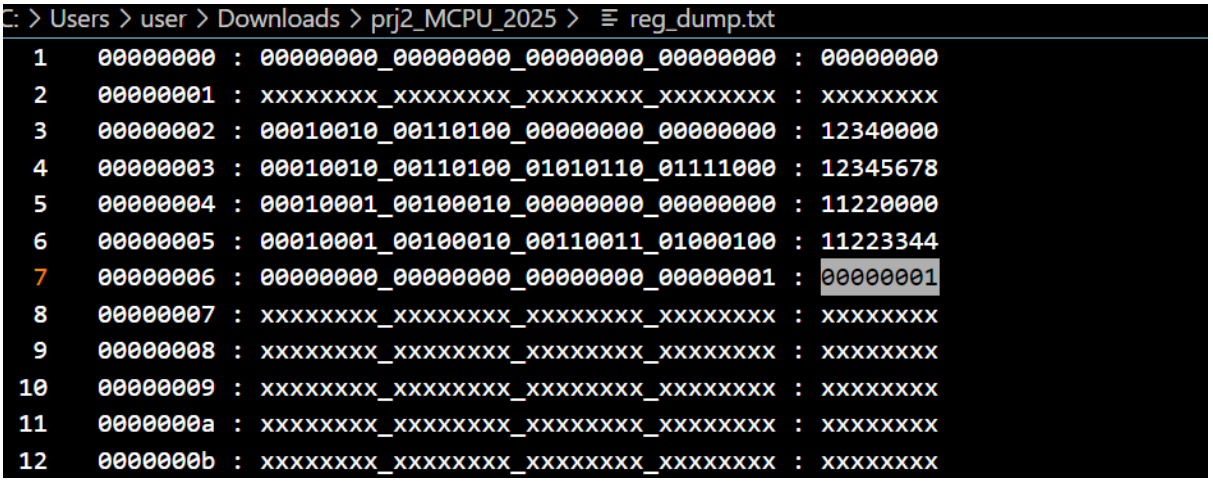
SLTU 명령어는 두 레지스터 값을 비교하여 $rs < $rt일 때 결과를 1로, 아니면 0으로 설정합니다.

x_x_0_xxx_0_01_000_0_x_000_000_10001_00_000_00_0_xxxxxxxx_11 // 0x08: SLTU A B
AND와 동일하지만 OP부분에서 비교를 위한 10001이 존재합니다.
x_x_0_xxx_0_01_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00 // 0x09: SLTU Result SAVE
AND SAVE와 동일합니다.

8번째 작동하도록 설정했습니다.
1_00001000 // FN 101011 sltu

[이전의 명령어들 동일]

00000000_10000101_00110000_00101011 // sltu $4, $5 $6
$4 <$5라면 1을 반환하게 설계했습니다. 11220000<11223344로 1을 예측합니다.



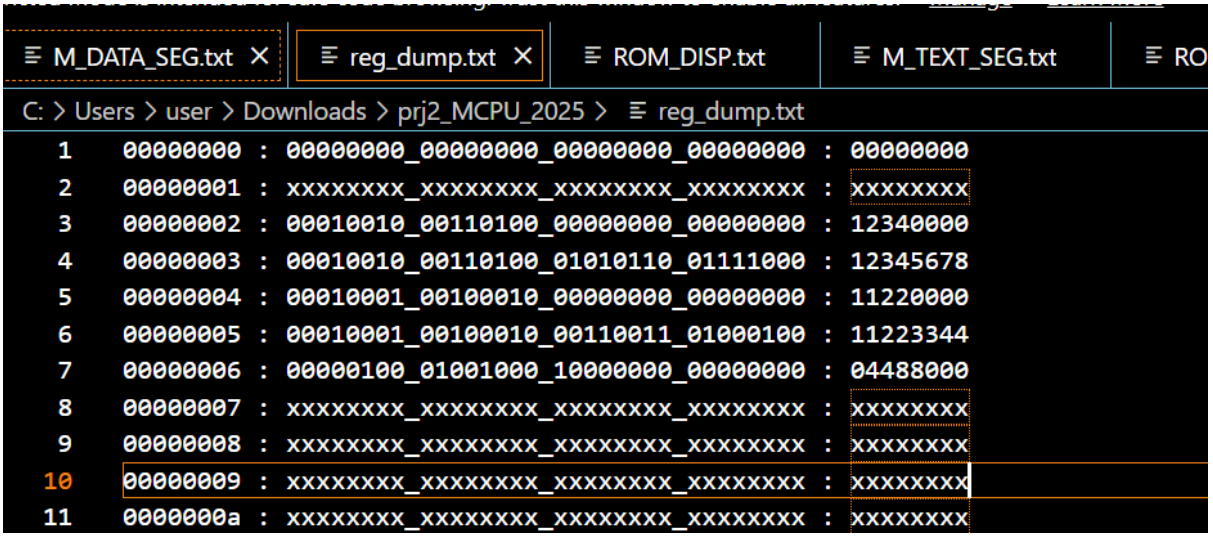
6번 레지스터 에서 1을 확인했습니다.

SRL

SRL 명령어는 레지스터를 오른쪽으로 시프트 합니다.
x_x_0_xxx_0_01_000_0_0_000_000_01110_00_000_00_0_xxxxxxxx_11 // 0x0a: SRL
AND와 동일하지만 0익스텐션을 명시합니다. 또한 OP에서 01110이 구현됩니다.
x_x_0_xxx_0_01_000_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00 // 0x0b: SRL SAVE
AND SAVE과 동일합니다.

10번째에 수행하도록 설정했습니다.
1_xxxx1010 // FN 000010 srl
[이전의 명령어들 동일]
00000000_00000100_00110000_10000010 // srl $6, $4, 2

1122인 4번 레지스터를 2번 시프트 즉 4배처리해서 4488을 예상합니다.



결과적으로 시프트 과정에서 상위0 추가를 통해서 04488이 구현되었다.

00111100_00000010_00010010_00110100 //lui $2, 0x1234 $2 = 0x12340000

00110100_01000011_01010110_01111000 //ori $3, $2, 0x5678 $3 = 0x12345678

00111100_00000100_00010001_00100010 //lui $4, 0x1122 $4 = 0x11220000

00110100_10000101_00110011_01000100 //ori $5, $4, 0x3344 $5 = 0x11223344

00000000_00000100_00110000_10000010 // srl $6, $4, 2

SH

SH 명령어는 rt에 저장된 하프워드(16비트) 값을 메모리 주소 rs + immediate에 저장하는 명령어입니다. I-type 명령어이며, rt를 목적지로 사용하므로 00으로 regdst를 설정했습니다. ALUsrcB는 즉시값으로 011을 사용했습니다. 이번 저장 명령어에서는 메모리에 저장을 수행하므로 RegWrite 대신에 MemWrite부분을 1로 설정하였습니다.

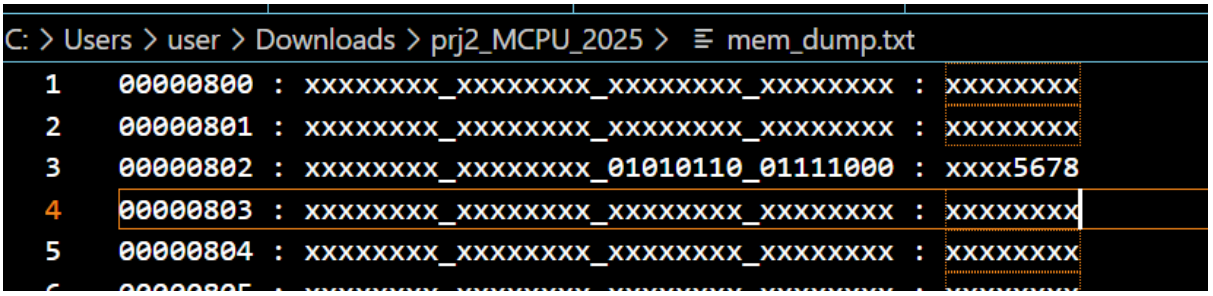
x_x_0_xxx_0_xx_xxx_0_1_000_011_00100_0x_xxx_xx_0_xxxxxxxx_11 // 0x0c: SH
메모리 저장은 다음 단계에서 수행됩니다. 또한 레지스터 사용을 하므로 1이 부여되고 SEU output to ALU input B로 인해서 011이 부여됩니다. sh또한 덧셈을 수행하므로 OP에서는 ADD와 동일한 기능이 부여되고 이후 SAVE단계를 위한 11로 마무리 됩니다.
1_x_1_110_0_xx_xxx_0_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00 // 0x0d: SH SAVE
저장을 위해서 메모리 기능등이 부여됩니다. 16비트 하프를 위해서 110이 부여됩니다.

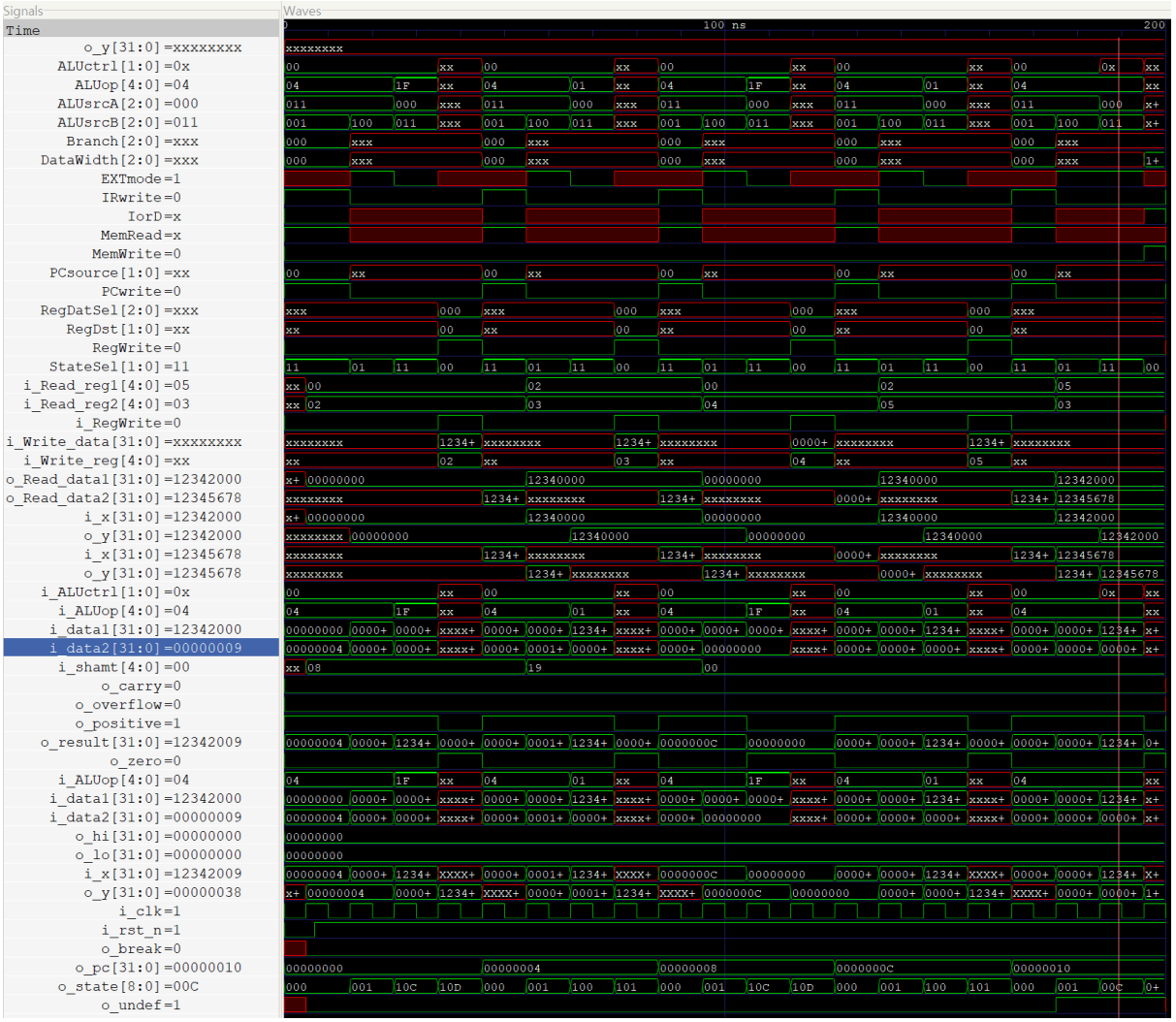
12번째에 수행하도록 설정했습니다.
1_00001100 // OP 101001 sh

00111100_00000010_00010010_00110100 // lui $2
00110100_01000011_01010110_01111000 // ori $3, $2
00111100_00000100_00000000_00000000 // lui $4
00110100_10000101_00100000_00000000 // ori $5, $4

10100100_10100011_00000000_00001000 // sh $3, 8($5)

이번 명령어는 메모리 환경에 따라 특수하게 변경하였습니다. 해당 명령어에서는 5번 레지스터와 9를 더하는 모습입니다. 이 때 5번 주소에는 000020000 + 000000009를 통해 00002009가 완성되고 이후 해당 메모리를 뒤에 4자리인 2009를 기준으로 하프로 잘라옵니다.
이후 오른쪽으로 2번 시프트를 수행하여 2진수로 표현하면 0000 1000 0000 0010이고 10진수로 표현하면 0802가 구현됩니다.





LB

lb (Load Byte) 명령어는 메모리 주소 [$s + offset]의 1바이트를 부호 확장하여 $t 레지스터에 저장합니다.

이전의 sh와 반대로 메모리에 접근 후 레지스터에 저장하니 MemWrite대신에 RegWrite 부분을 1로 설정하였습니다. 또한 이번에는 LB에서 읽는 부분과 저장하는 부분이 분할되어 총 3개의 단계를 거쳐 수행합니다.

x_x_0_xxx_0_xx_xxx_0_1_000_011_00100_0x_xxx_xx_0_xxxxxxxx_11 // 0x0e: LB

레지스터를 사용하므로 RegWrite가 1로 설정되지 않고, ALU 입력 A는 레지스터 값을, 입력 B는 SEU 출력을 사용하므로 각각 000과 011이 설정됩니다. 주소 계산은 덧셈이므로 ALUop은 ADD에 해당하는 00100이 부여됩니다. 이후 StateSel은 11로 설정됩니다.

1_1_0_110_0_xx_xxx_0_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_11 // 0x0f: LB READ

데이터 메모리에 접근하기 위해 IorD는 1로, MemRead는 1로 설정되며, 바이트 단위 읽기와 부호 확장을 위해 DataWidth는 111로 설정됩니다. 이후 StateSel은 11이 유지됩니다.

x_0_0_xxx_0_00_001_1_x_xxx_xxx_xxxxx_xx_xxx_xx_0_xxxxxxxx_00 // 0x10: LB SAVE

목적지 레지스터는 rt이므로 RegDst는 00으로, 저장할 값은 MDR이므로 RegDatSel은 001, 실제 쓰기를 위해 RegWrite는 1로 설정되며, 이후 명령어 실행을 위해 StateSel은 00으로 끝냅니다.

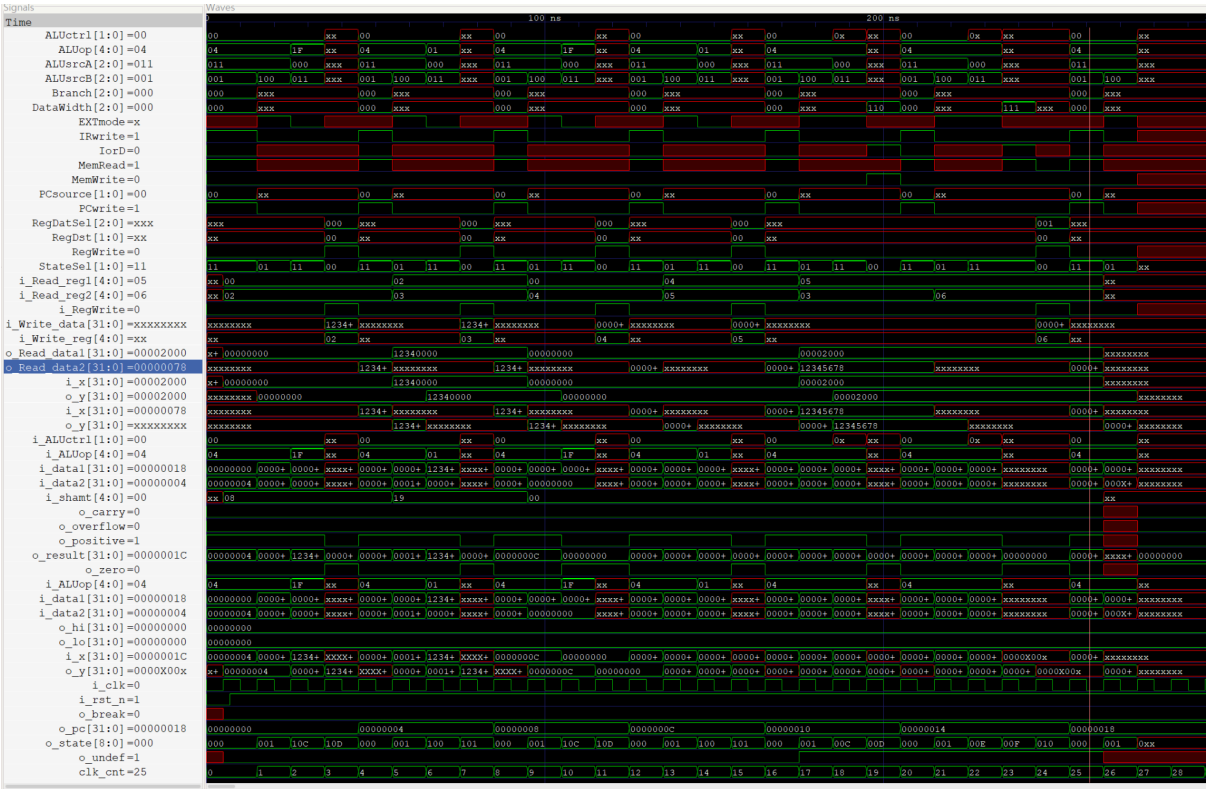
11번째에 수행하도록 설정했습니다.

1_00001011 // OP 100000 lb

00111100_00000010_00010010_00110100 // lui $2
00110100_01000011_01010110_01111000 // ori $3, $2
00111100_00000100_00000000_00000000 // lui $4
00110100_10000101_00100000_00000000 // ori $5, $4

10100100_10100011_00000000_00001000 // sh $3, 8($5)
10000000_10100110_00000000_00001000 // lb $6, 8($5)

해당 코드는 앞선 sh 수행 코드에서의 연장선으로 800번까지 이동을 위해 5번 레지스터를 이용하고 이후 8을 통해 8/4 = 2 를 통해 802번 주소에 접근했습니다. 이후 5678중에 1바이트만 읽기때문에 2개인 뒤에 78만 가져와서 $6에 저장되기를 예상합니다.



78 값을 확인할 수 있었습니다.

reg_dump.txt | ROM_DISP.txt | M_TEXT_SEG.txt | ROM_MICRO.txt

C: > Users > user > Downloads > prj2_MCPU_2025 > reg_dump.txt

```
1   00000000 : 00000000_00000000_00000000_00000000 : 00000000
2   00000001 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
3   00000002 : 00010010_00110100_00000000_00000000 : 12340000
4   00000003 : 00010010_00110100_01010110_01111000 : 12345678
5   00000004 : 00000000_00000000_00000000_00000000 : 00000000
6   00000005 : 00000000_00000000_00100000_00000000 : 00002000
7   00000006 : 00000000_00000000_00000000_01111000 : 00000078
8   00000007 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
9   00000008 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
10  00000009 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
11  0000000a : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
```

레지스터에서 확인했습니다.

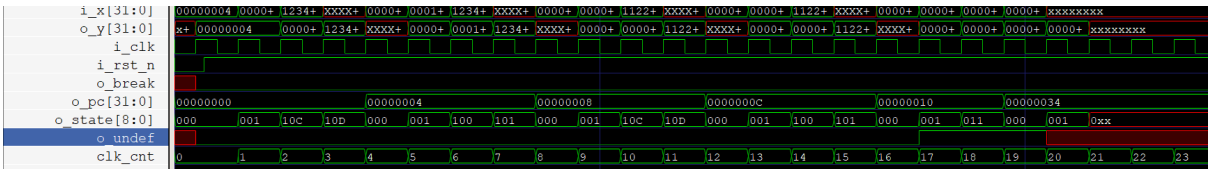
BNE

bne는 레지스터의 값을 비교하여 같다면 주소이동을 수행하는 명령어 입니다.
x_x_0_xxx_0_xx_xxx_0_x_000_000_00110_0x_101_01_1_xxxxxxxx_00 // BNE
bne의 경우 저장이나 추후 처리가 없으므로 해당 줄에서 처리가 끝납니다. 또한 해당 명령어에서는 Branch if not equal 를 수행하므로 101과 ALUOut Register를 선택합니다.
또한 PCwrite로 점프를 구현합니다.
ALU 입력 A는 레지스터 A, 입력 B는 레지스터 B를 사용하므로 각각 000, 000이 설정됩니다.
ALU는 두 값을 빼서 비교 결과를 판단해야 하므로 ALUop은 SUB에 해당하는 00110를 사용합니다.

17번째에서 수행하도록 구현했습니다.
1_00010001 // OP 000101 bne

00111100_00000010_00010010_00110100 //lui $2, 0x1234 $2 = 0x12340000
00110100_01000011_01010110_01111000 //ori $3, $2, 0x5678 $3 = 0x12345678
00111100_00000100_00010001_00100010 //lui $4, 0x1122 $4 = 0x11220000
00110100_10000101_00110011_01000100 //ori $5, $4, 0x3344 $5 = 0x11223344

00010100_10100100_00000000_00001000 //bne $5, $3
해당 명령어는 의도적으로 3번과 5번 레지스터에서 다른 값을 제공하고 이후 비교를 통해 pc값의 증가를 확인하려 합니다. 10번째 주소 이후 점프를 통해 +4 + 20을 통해 34로 점프를 예상합니다,



BGEZ

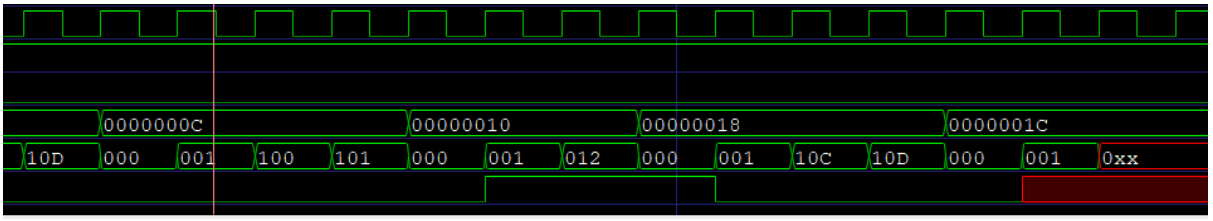
bgez는 레지스터의 값이 0이상이면 분기하는 명령어입니다.
x_x_0_xxx_0_xx_xxx_0_x_000_010_00110_0x_111_01_1_xxxxxxxx_00 // 0x12: BGEZ
동일하게 한 클럭에서 수행이 완료됩니다. 또한 명령어에서 Branch if positive를 요구하여 111을 세팅하였습니다. ALU 입력 A는 비교 대상이 되는 레지스터 값을 받기 위해 000으로 설정되며, 입력 B는 0과 비교하기 위해 010으로 설정했습니다. Branch if positive를 위해서 111을 브랜치에서 할당했으며 PC를 ALUOut로 점프하기 위한 요소들을 01로 부여했습니다. pcWRITE도 할당했습니다.

18번째에서 수행하도록 구현했습니다.
1_00010010 // OP 000110 blez

00111100_00000010_00010010_00110100 //lui $2, 0x1234 $2 = 0x12340000
00110100_01000011_01010110_01111000 //ori $3, $2, 0x5678 $3 = 0x12345678
00111100_00000100_00010001_00100010 //lui $4, 0x1122 $4 = 0x11220000
00110100_10000101_00110011_01000100 //ori $5, $4, 0x3344 $5 = 0x11223344

00000100_10100001_00000000_00000001 // bgez $5, 1
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
00111100_00000010_00010010_00110111 //lui $2, 0x1234 $2 = 0x12340000

해당 명령어를 통해 $5에 양수를 넣었으므로 1번 점프하는 양상을 기대하고 있습니다.



결과에서 다음 pc주소로 10 > 18로 이동한것을 확인할 수 있으며, 이후 우리가 임의로 넣어둔 명령어를 잘 인식하고 있는 모습을 확인할 수 있습니다.

JALR

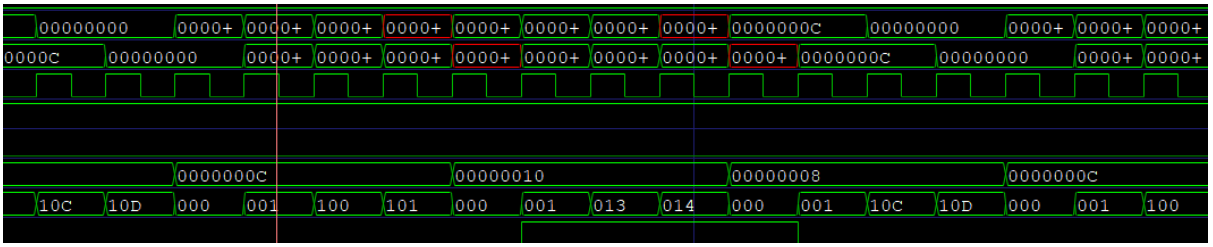
jalr은 입력받은 레지스터로 점프합니다. 이번 과제에서는 $31에 rs명령어를 저장하는 기능을 구현합니다.

x_x_0_xxx_0_11_100_1_x_000_010_00100_0x_xxx_xx_0_xxxxxxxx_11 // 0x13: JALR
x_x_0_xxx_0_xx_xxx_0_x_xxx_xxx_xxxxx_xx_000_01_1_xxxxxxxx_00 // 0x14: JALR SAVE

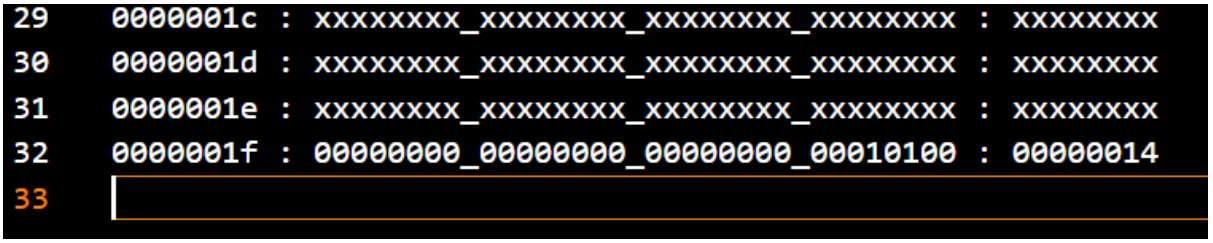
레지스터는 rd이므로 RegDst는 11로, 저장할 데이터는 현재 PC이므로 RegDatSel은 100으로 RegWrite는 1로 설정했습니다. 점프 후 저장하는 기능을 위해서 2개의 클럭을 사용합니다. 또한 PC + 4를 만들기 위해 ALU에서 덧셈 연산을 하므로 00100을 사용합니다.

00111100_00000010_00010010_00110100 // lui $2, 0x1234
00110100_01000011_01010110_01111000 // ori $3, $2, 0x5678 ($3 = 0x12345678)
00111100_00000100_00000000_00000000 // lui $4, 0x1122
00110100_10000101_00000000_00001000 // ori $5, $4, 0x3344 ($5 = 0x00000008)

00000000_10100000_00000000_00001001 // jalr $5
해당 자료를 기반으로 5번 레지스터를 8로 맞추고 jalr명령어를 통해서 5번레지스터의 8번 pc값을 로드합니다.



jalr다음 8번 pc값으로 들어가 반복하는 모습을 확인할 수 있습니다.



레지스터도 1f , 10진수로 31번째 값에 14주소 값이 저장되어 있는 모습을 확인할 수 있습니다.
무한반복에 갖히는 모습 또한 확인하였습니다.

고찰
이번 프로잭트에서는 싱글 사이클을 이어 멀티사이클 구조의 mips를 수행했습니다. 수행하면서 느낀 부분은 메모리의 조건이 달라져서 sh와 lb를 수행하면서 800을 직접 넣어야하는 경험을 하기도 했습니다. 또한 ROM DISP에서 각각의 명령어가 몇번째 ROM MICRO에 있는지 확인하는 과정이 추가되었는데, 이를 통해서 이런 환경이 구현되는 컴퓨터마다 다르다면 혼선이 발생할 수 있겠구나 라는 생각을 하기도 했습니다. 정해진 규칙이 있는지 궁금증이 생기기도 했습니다.

또한 이번 과제에서 한 수행에서 클럭이 3개 2개등등 할당되는 모습을 통해서 싱글과 멀티의 차이점을 확인할 수 있었습니다.

Reference

프로젝트2 제안서 2025.pdf