컴퓨터구조 Project #1

(MIPS Single Cycle CPU Implementation)

수업명: 컴퓨터구조

교수님: 이성원 교수

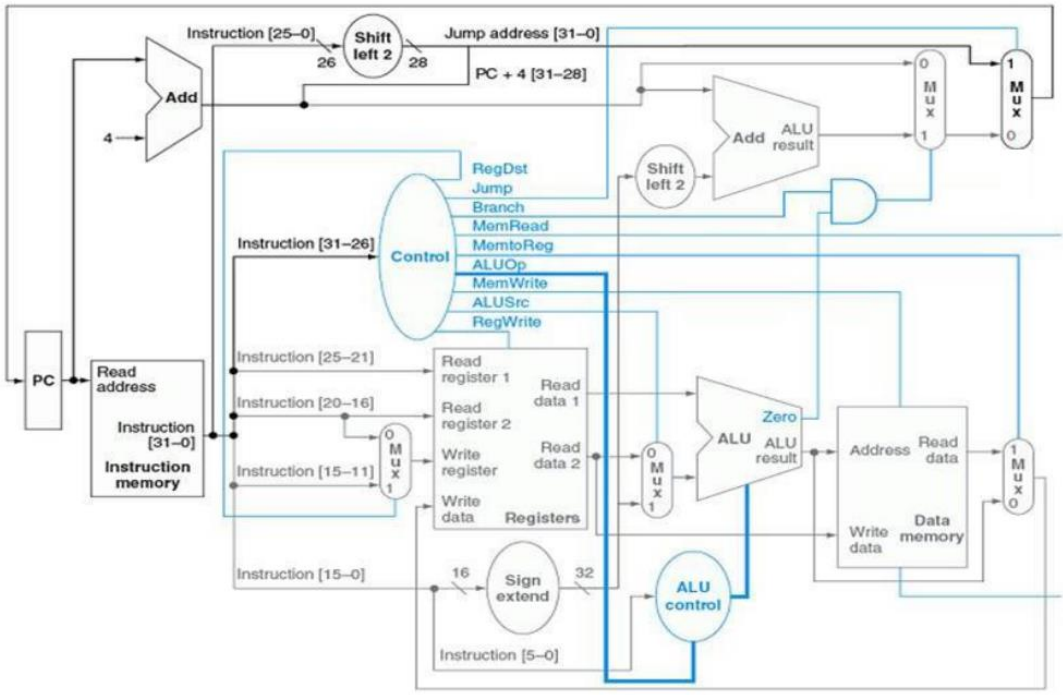
수업 시간: 월 3교시, 수 4교시

소속: 컴퓨터정보공학부

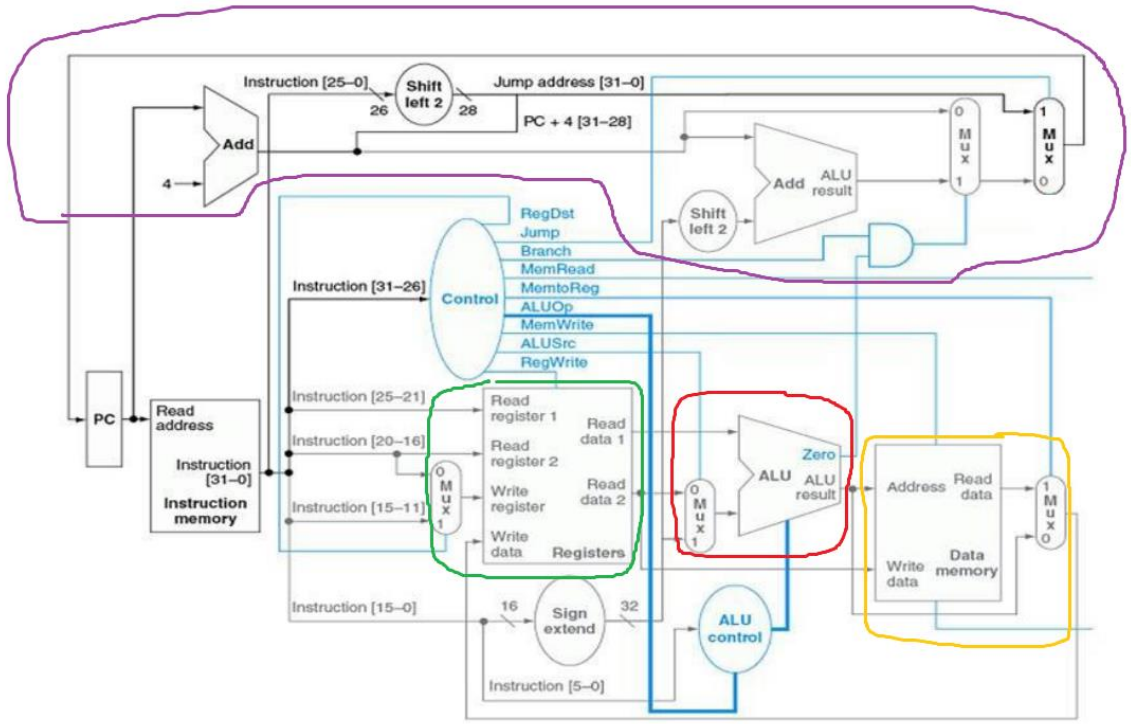
학번: 2018202074

이름: 김상우

-전체 회로 분석 및 Block도 작성



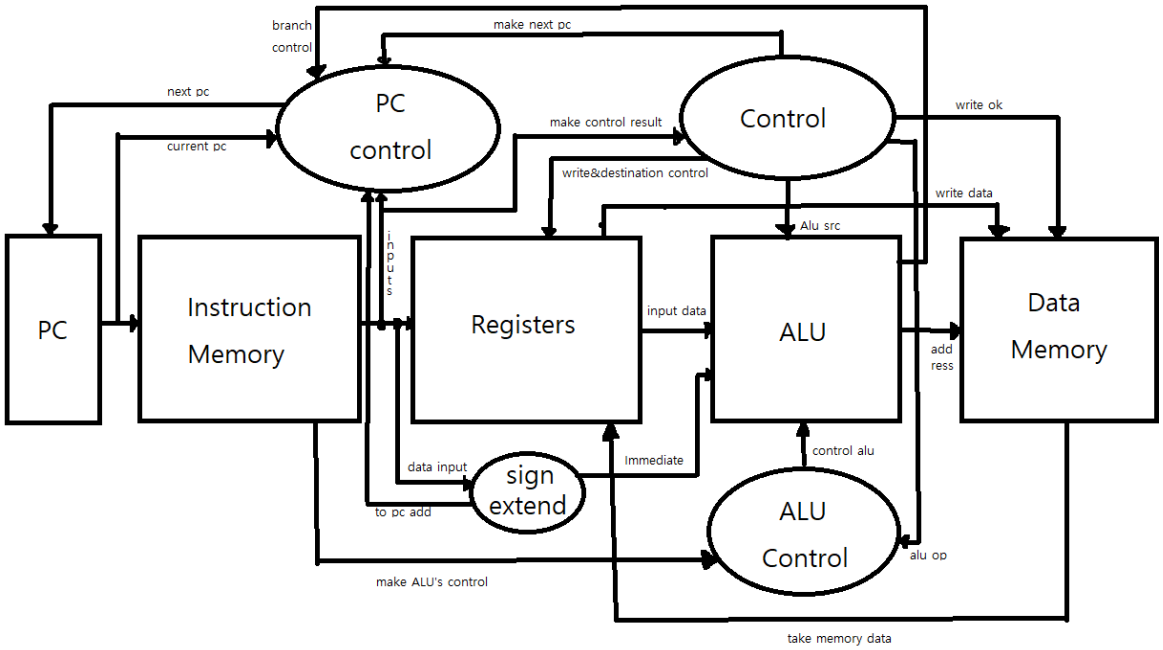
위 프로젝트는 다음 회로를 기반으로 진행된다.(Branch와 ALUzero값의 AND를 받는 MUX 수정)



이후 설명의 편의성을 위해 보라색 부분을 PC control, 초록색 부분을 Registers, 붉은 부분을 ALU, 노란 부분을 Data Memory로 칭하겠다.(노란 구역에 들어간다고 무조건 Data Memory 들어가는 건 아니다)

위 회로에서 알 수 있듯, PC는 Instruction memory를 통해 해당 위치의 명령을 불러오고 이 값들은 Control, Register, Sign extend, ALU control로 향한다. Control은 이를 바탕으로 값을 정하고 Instruction과 PC를 제외한 모든 부분에 직접적으로 영향을 미친다. Control과 Instruction memory의 값들을 바탕으로 Register는 ALU와 Memory에 값을 준다. ALU는 ALU Control과 register에서 온 값을 바탕으로 결과를 출력하고 이를 Data Memory와 PC Control로 보낸다. 이 값과 Register, 그리고 Control의 값을 바탕으로 Memory는 Register로 값을 보내 Register 값을 변경한다. PC Control의 경우 위의 결과로 본인에게 들어오는 신호들을 계산, 다음 PC값을 PC에 넘겨준다.

이를 바탕으로 다음 Block도를 그릴 수 있을 것이다.



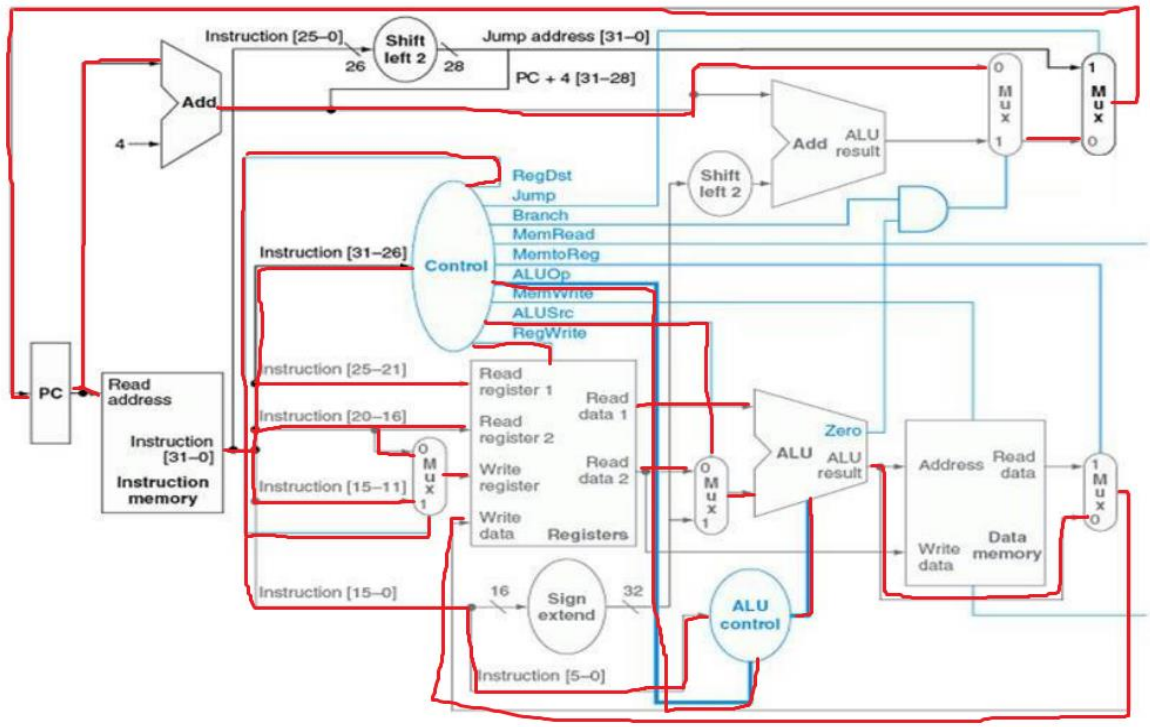
-다른 방법

위의 경우 Sign extend를 따로 빼서 PC Control과 ALU에 넣어주는 형식을 취하고 있다. 허나 ALU 자체적으로도 Shift를 비롯한 extend를 구현할 수 있다.(다만 unknown에 대해 처리가 미흡할 수 있다.) 이를 바탕으로 sign extend를 없애고 이 일을 ALU가 대신하여 회로를 단순화 시킬 수 있을 것이다. 다만, 회로의 단순화와는 별개로 ALU의 input값 중 하나를 본인이 계산 후 다시 본인의 input으로 주어야 하기 때문에 2cycle이 필요하게 될 수도 있으며 ALU의 구조 또한 완전히 달라지게 될 것이다.

또한 pc값이 register에 연결이 안되어 있는데, 이 때문에 회로 상으로는 register에 pc값을 받아올 방법이 없다.(jalr등에 활용해야 하나 해당 회로에는 없다.) 이 때문에 이후 회로에서의 설명에 해당하는 그림에서는 이를 표기하지 않으나 pc값을 직접적으로 연결할 필요가 있어보인다.

이 외에도 현재 회로는 Single Cycle의 형태를 가지고 있다. 그 때문에 한 Cycle은 crytical path를 기준으로 만들어져 있어 한 Cycle당 시간이 불필요하게 많은데, 이는 Multi Cycle로 변경하여 해결할 수 있을 것이다. 즉, Cycle을 어느정도 작게 설정하고 Instruction당 1cycle이 아니라 각 Instruction는 n개의 cycle이 필요하다는 식으로 분류하는 것이다. 위 회로에서 보면 알 수 있듯이 위 회로는 data path가 instruction별로 차이가 크고 그 때문에 crytical path를 기준으로 하는 single cycle은 효과적이지 못하다.

-SUBU



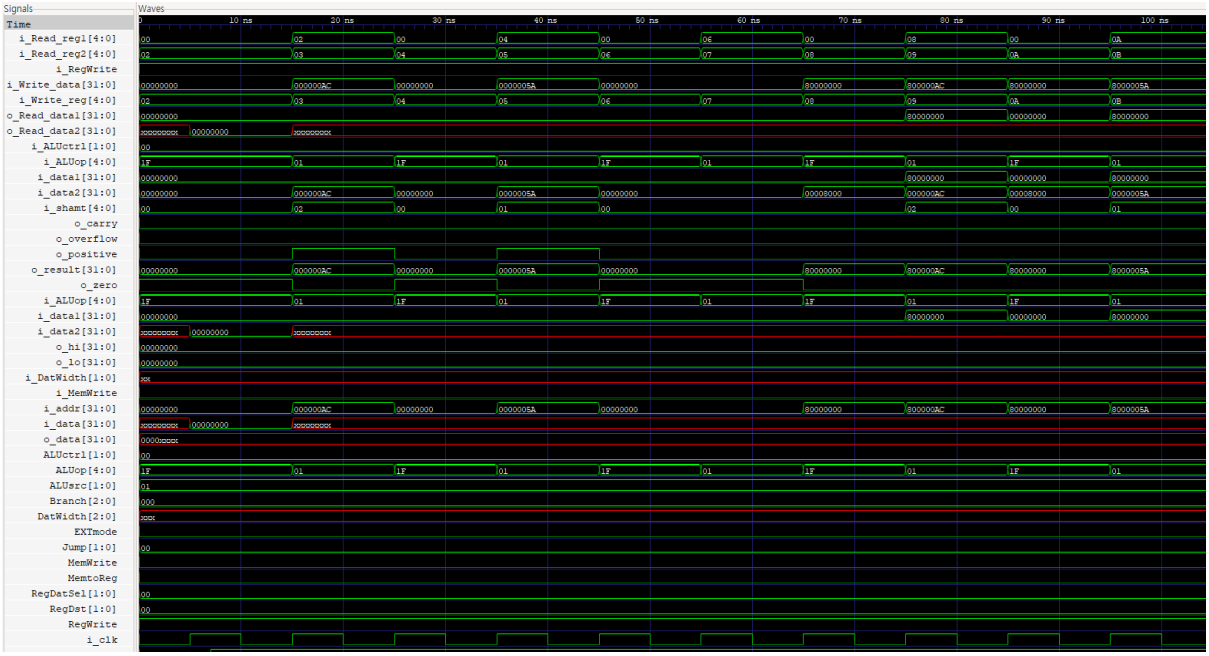
다음은 SUBU의 진행을 주어진 회로도 위에 그려 넣은 것이다. SUBU가 있는 PC값에 도달 시 PC 값은 위의 ADD로 보내 ALU와 MUX를 통해 PC+4를 진행할 수 있도록 해준다. Instruction Memory로 향한 PC값은 Memory에서 각 bit별로 내용을 꺼내어 [31-26]은 Control, [25-21]은 regitster1=$rs, [20-16]은 register2=$rt와 mux, [15-11]=$rd은 mux, [15-0]은 sign extend와 ALU control로 향한다. SUBU의 경우 Control에 의해 Write register는 [15-11]bit 값에 해당하는 register가 된다. RegWrite가 활성화 되므로 결과적으로 레지스터에 결과가 저장될 것이다. 또한 Control에 의해 ALUOP(어떤 ALU의 기능을 쓸지 결정 위에선 00111로 unsigned a-b)와 ALUSrc(위에선 $rt의 값을 사용한다)가 정해진다. 이를 기반으로 ALU가 실행된다. ALU Control로부터 동작을 받아와 Input데이터를 기반으로 ALU result를 생성한다. MemtoReg가 0이므로 ALU result가 Writedata로 이동, 결과적으로 $rs와 $rt가 unsigned sub된 값이 $rd에 저장되고 pc=pc+4가 된다.

SUBU의 경우 뺄셈을 unsigned의 형태로 진행해 주어야한다. 다음을 위해 PLA_AND와 PLA_OR에 다음과 같은 값을 넣어주었다. (각 제어신호의 사용 이유는 명령어가 PLA_OR 아래 서술된 조건을 만족해야 하기 때문이다.)

PLA_AND: 000000_100011_xxxxx (R-type이므로 opcode는 000000이 되고 function이 100011이다.)

PLA_OR: 01_00_1_0_00_00_00111_xxx_0_0_000_00_xxxxx

$rd/ALU Result/Register Write/Zero Extension/Register file port B/Normal ALU input&Shift amount(no shift)/Unsigned a-b/Any data width for memory/no write data to memory/write ALU data to Register file/Use PC+4/No Jump



Testbench에 적힌 상태대로 입력됨을 확인할 수 있다.(각각 input값이 잘 들어감을 Hex로 알 수 있다.) 위 결과에서 알 수 있듯이 각 레지스터에는 다음과 같은 값들이 들어간다.

Register 3=00011(2):0x000000AC Register 00101(2):0x0000005A

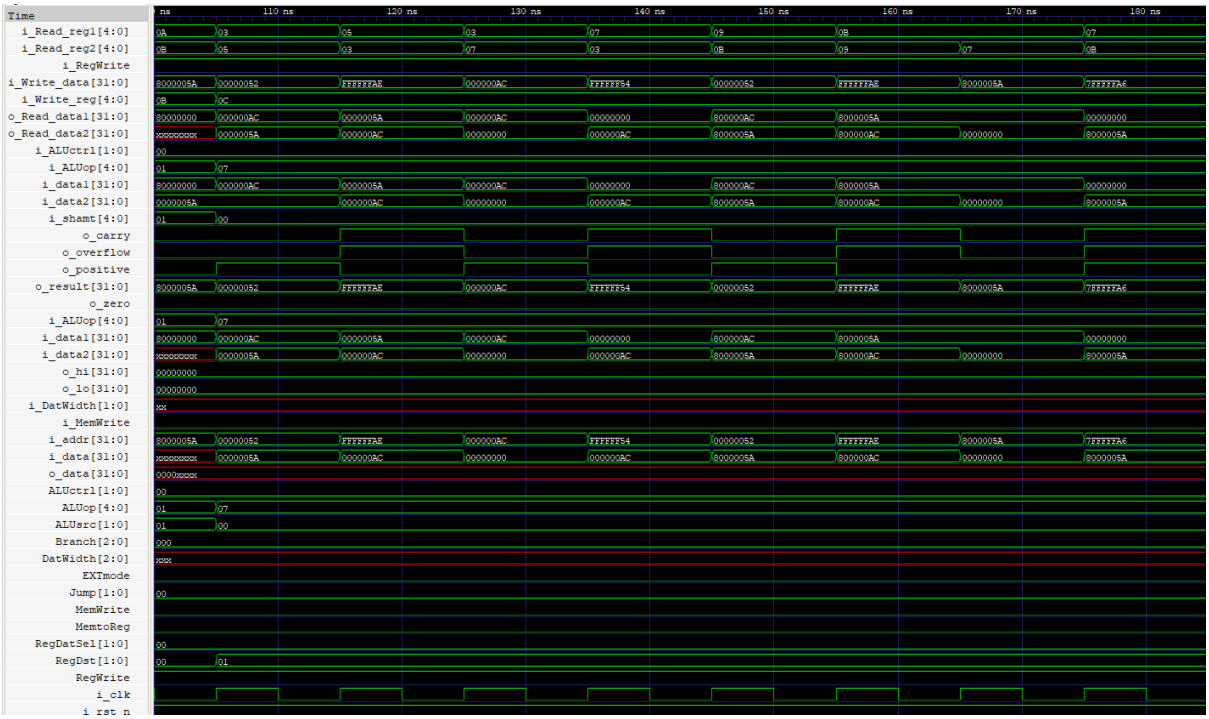
Register 00111(2):0x00000000 Register 01001(2):0x800000AC

Register 01011(2):0x8000005A

해당 값들을 레지스터에 lui와 ori를 이용해 넣은 후 테스트 벤치를 돌려본다. 이때 테스트 벤치는 각각 다음과 같이 잡았다. (양수-양수 결과로 양수/음수가 나올 때 양수-0, 0-양수일 때, MSB가 1일 때의 계산으로 결과가 양수/음수가 나올때, MSB가 1일 때 0을 뺄 때, 0에서 MSB가 1인 값을 뺄 뺄 때.) 각각의 케이스에 대해 알맞은 레지스터를 지정했다.

(reg3-reg5, reg5-reg3, reg3-reg7 ,reg7-reg3,reg9-reg11,reg11-reg9,reg11-reg7,reg7-reg11)

해당 코드를 돌리기 전, SUBU는 unsigned에서 진행되는 만큼, 결과가 0아래로 내려갈 경우 음수를 표현하지 못해 오류가 생길 것이라 예상했다.

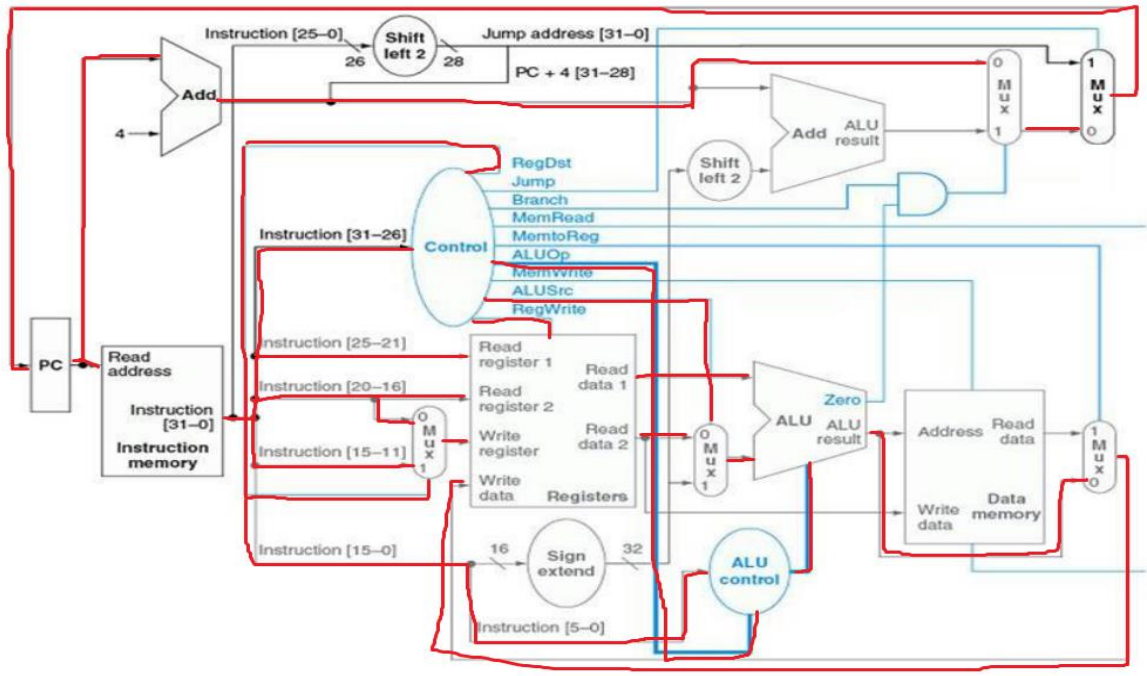


다음은 결과이다. I_Read_reg1,2와 i_data1,2의 값으로 알맞은 레지스터에서 값이 내보내짐을 확인할 수 있었다. 또한 i_Write_reg가 0x0C임을 통해 지정했던 0x0C에 저장되고 있음도 확인했다. O_result를 통해 ALU의 결과를 확인할 수 있었다.

Input1(hex)	Input2(hex)	Expected result(hex)	Result(hex)
000000AC	0000005A	00000052	00000052
0000005A	000000AC	FFFFFFAE(error)	FFFFFFAE(error)
000000AC	00000000	000000AC	000000AC
00000000	000000AC	FFFFFF54(error)	FFFFFF54(error)
800000AC	8000005A	00000052	00000052
8000005A	800000AC	FFFFFFAE(error)	FFFFFFAE(error)
8000005A	00000000	8000005A	8000005A
00000000	8000005A	7FFFFFAE(error)	7FFFFFAE(error)

예상대로 SUBU는 unsigned에서 진행되는 만큼 음수가 나올 시 적절한 값을 출력해주지 못했다. 허나 결과값이 0이상의 값일 경우 올바르게 표현되고 있음을 확인할 수 있었다.

-XOR



다음은 XOR의 진행을 주어진 회로도 위에 그려 넣은 것이다. XOR가 있는 PC값에 도달 시 PC값은 위의 ADD로 보내 ALU와 MUX를 통해 PC+4를 진행할 수 있도록 해준다. Instruction Memory로 향한 PC값은 Memory에서 각 bit별로 내용을 꺼내어 [31-26]은 Control, [25-21]은 regitster1=$rs, [20-16]은 register2=$rt와 mux, [15-11]=$rd은 mux, [15-0]은 sign extend와 ALU control로 향한다. XOR의 경우 Control에 의해 Write register는 [15-11]bit 값에 해당하는 register가 된다. RegWrite가 활성화 되므로 결과적으로 레지스터에 결과가 저장될 것이다. 또한 Control에 의해 ALUOP(어떤 ALU의 기능을 쓸지 결정 위에선 00011로 Bitwise XOR)와 ALUSrc(위에선 $rt의 값을 사용한다)가 정해진다. 이를 기반으로 ALU가 실행된다. ALU Control로부터 동작을 받아와 Input데이터를 기반으로 ALU result를 생성한다. MemtoReg가 0이므로 ALU result가 Write data로 이동, 결과적으로 $rs와 $rt가 bitwise xor된 값이 $rd에 저장되고 pc=pc+4가 된다.

XOR의 경우 두 input값을 binary형태로 비교하여 각 bit의 값이 다를 시 해당 bit에 1을, 그렇지 않으면 0을 입력하는 값을 가진다. 다음을 위해 PLA_AND와 PLA_OR에 다음과 같은 값을 넣어주었다. (각 제어신호의 사용 이유는 명령어가 PLA_OR 아래 서술된 조건을 만족해야 하기 때문이다.)

PLA_AND: 000000_100110_xxxxx(R-type이므로 opcode는 000000이 되고 function이 100110이다.)

PLA_OR:01_00_1_x_00_00_00011_xxx_0_0_000_00_xxxxx

$rd/ALU Result/Register Write/No Extension/Register file port B/Normal ALU input&Shift amount(no shift)/Bitwise XOR/Any data width for memory/no write data to memory/write ALU data to Register file/Use PC+4/No Jump

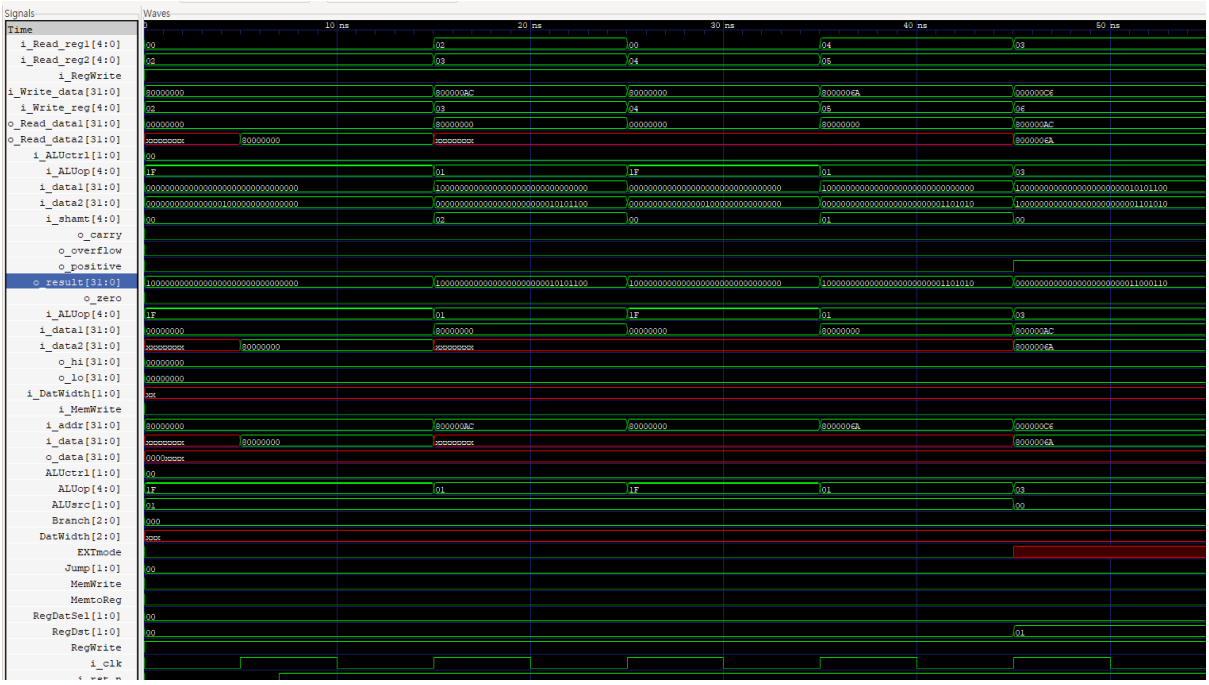
Testbench의 경우에는 두 input의 bit가 각각 (0,0)(1,0)(0,1)(1,1)일 때를 고려해서 input값을 주었다.

이를 위해 두개의 input은 다음과 같이 주었다.

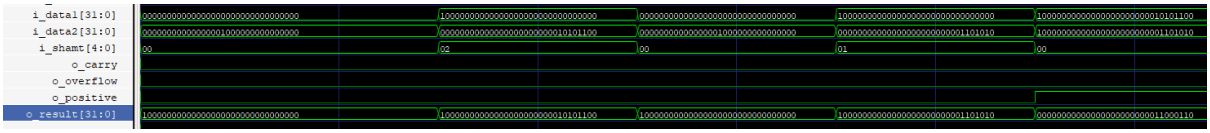
10000000_00000000_00000000_10101100(2) / 100000000_00000000_00000000_01101010(2)

이 둘을 각각 reg3, reg5에 저장 후 XOR하여 reg6에 값을 저장하는 테스트벤치로 진행하였다.

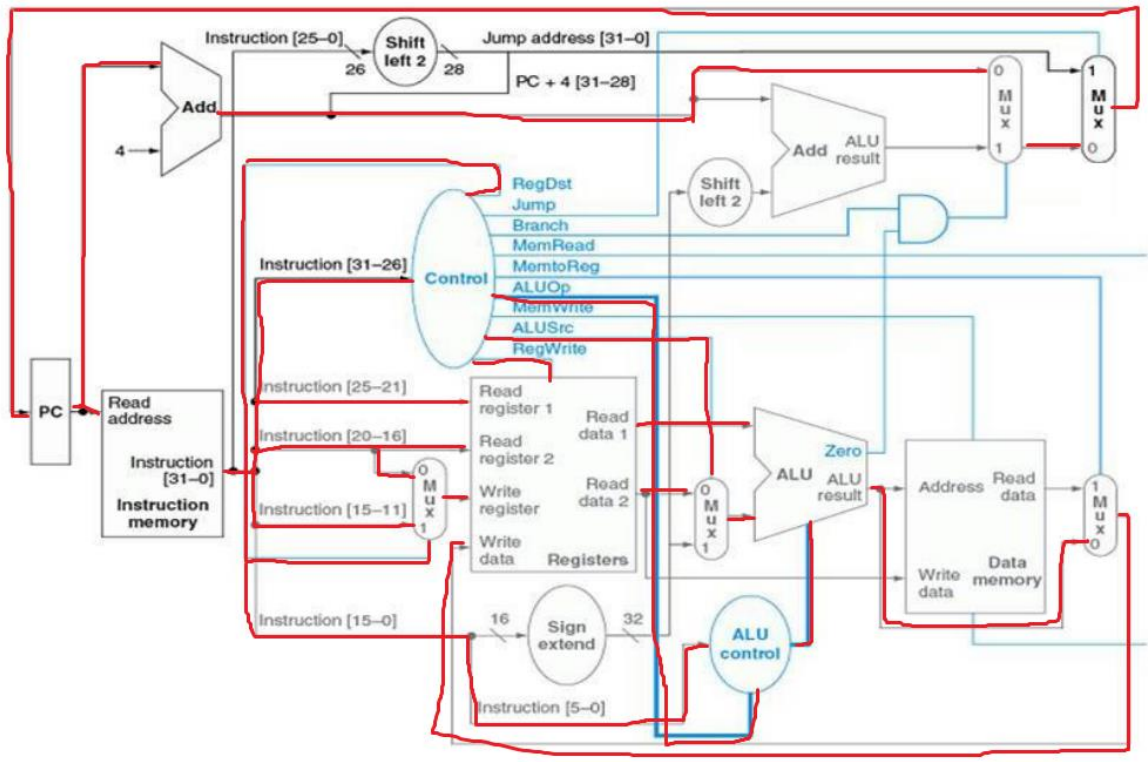
이들은 n번째의 bit가 다를 경우 reg6의 nbit에 1을, 같을 경우 0을 줄것이므로 예측 결과값은 00000000_00000000_00000000_11000110(2)이다.



위와 같은 결과가 나왔다. 결과를 보면 input으로 register3, 5에 0x800000AC, 0x8000006A값이 들어감을 확인했다.(해당 hex값은 위의 input으로 하려했던 binary값과 같다.) 또한 XOR가 사용된 부분을 보면 00110(2)에 해당하는 곳에 잘 저장되고 있음을 알 수 있다. 또한 o_result를 통해 예측된 값과 같은 결과가 나왔다는 것을 확인했다.



-SRA



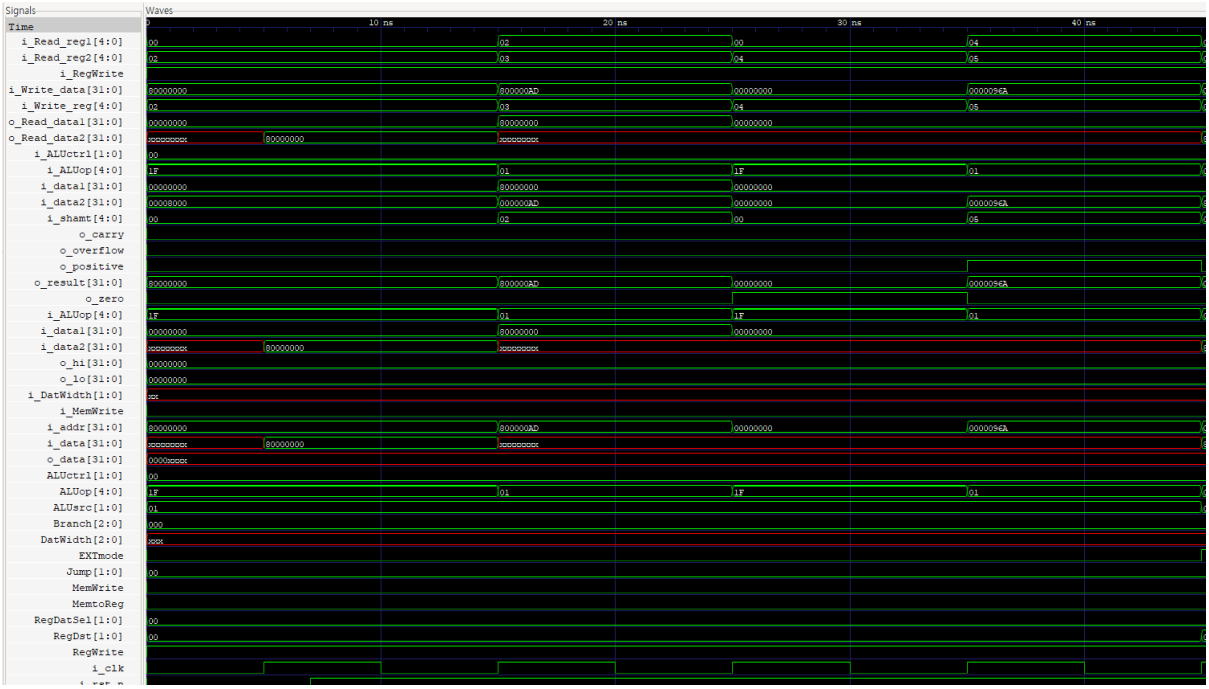
다음은 SRA의 진행을 주어진 회로도 위에 그려 넣은 것이다. SRA가 있는 PC값에 도달 시 PC값은 위의 ADD로 보내 ALU와 MUX를 통해 PC+4를 진행할 수 있도록 해준다. Instruction Memory로 향한 PC값은 Memory에서 각 bit별로 내용을 꺼내어 [31-26]은 Control, [25-21]은 regitster1=$rs, [20-16]은 register2=$rt와 mux, [15-11]=$rd은 mux, [15-0]은 sign extend와 ALU control로 향한다. SRA의 경우 Control에 의해 Write register는 [15-11]bit 값에 해당하는 register가 된다. RegWrite가 활성화 되므로 결과적으로 레지스터에 결과가 저장될 것이다. 또한 Control에 의해 ALUOP(어떤 ALU의 기능을 쓸지 결정 위에선 01111로 b>>>a)와 ALUSrc(위에선 shamt)가 정해진다. 이를 기반으로 ALU가 실행된다. ALU Control로부터 동작을 받아와 Input데이터를 기반으로 ALU result를 생성한다. MemtoReg가 0이므로 ALU result가 Write data로 이동, 결과적으로 $rt가 shamt만큼 right shift된 값이 $rd에 저장되고 pc=pc+4가 된다.

(각 제어신호의 사용 이유는 명령어가 PLA_OR 아래 서술된 조건을 만족해야 하기 때문이다.)

PLA_AND: 000000_000011_xxxxx(R-type이므로 opcode는 000000이 되고 function이 000011이다.)

PLA_OR: 01_00_1_1_00_00_01111_xxx_0_0_000_00_xxxxx

$rd/ALU Result/Register Write/Sign Extension/Register file port B/Normal ALU input&Shift amount/b>>>a/Any data width for memory/no write data to memory/write ALU data to Register file/Use PC+4/No Jump



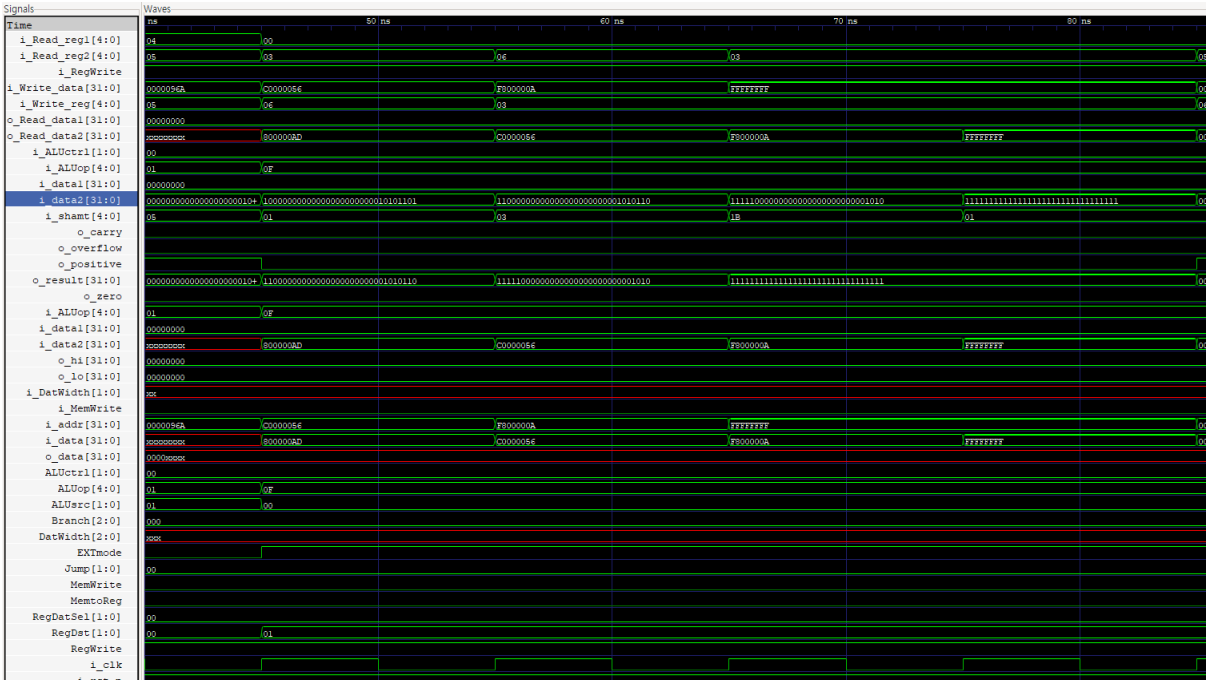
위는 test에 사용할 값 2개를 register3과 register5에 넣어주는 모습이다. 값들은 다음과 같다.

Register 3: 10000000_00000000_00000000_10101101(2)

Register 5: 00000000_00000000_00001001_01101010(2)

위는 각 레지스터가 1,3,27,1만큼 shamt를 갖게 해주었다. Sra가 정상적으로 작동한다면 다음과 같이 작동할 것이다.

Shamt	Value
0	10000000000000000000000010101101(2)
1	11000000000000000000000001010110(2)
3	11111000000000000000000000001010(2)
27	11111111111111111111111111111111(2)
1	11111111111111111111111111111111(2)
0	00000000_00000000_00001001_01101010(2)
1	000000000_00000000_00001001_0110101(2)
3	000000000000_00000000_00001001_0110(2)
27	00000000000000000000000000000000(2)
1	00000000000000000000000000000000(2)

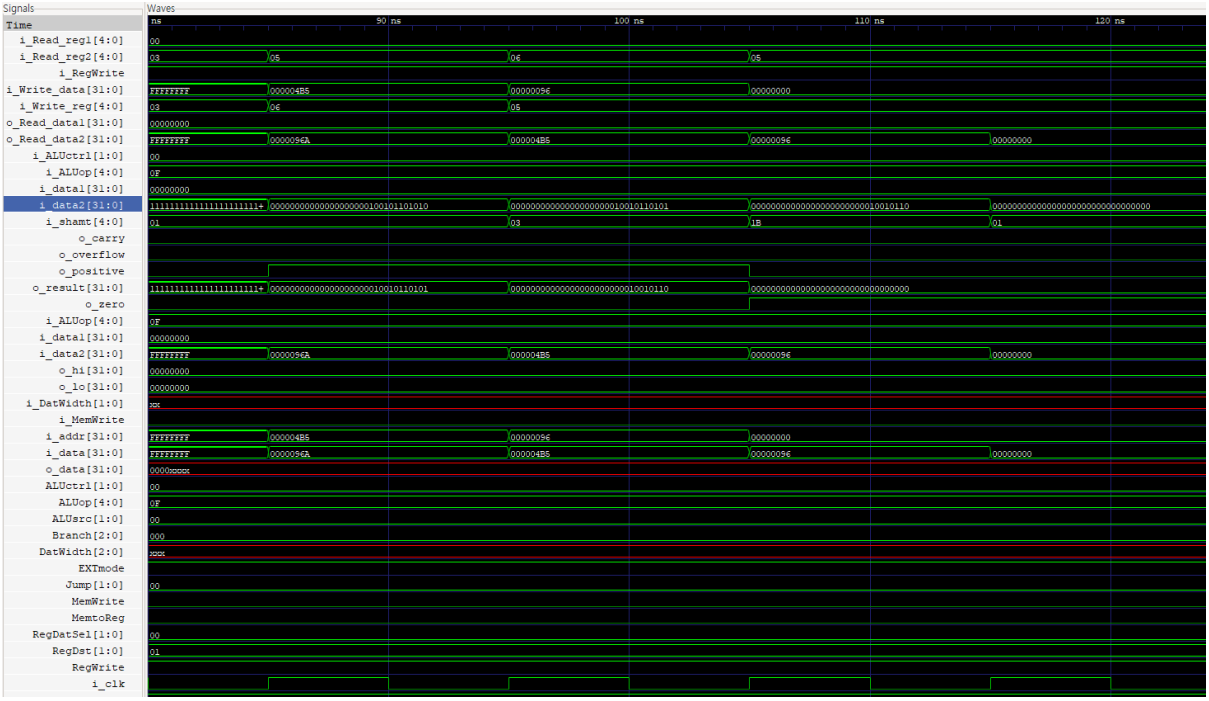


다음은 sra를 통해 shamt만큼 register3의 값이 오른쪽으로 shift되는 모습이다.

Shamt	Value
0	10000000000000000000000010101101(2)
1	11000000000000000000000001010110(2)
3	11111000000000000000000000001010(2)
27	11111111111111111111111111111111(2)
1	11111111111111111111111111111111(2)

위 결과에서 shamt값 만큼 shift right 되고 있음을 알 수 있다.

이때 MSB가 1이고 sra에서는 sign인 상태로 유지되어야 하기 때문에 shift로 인해 공백이 생긴 부분은 MSB의 값에 따라 1로 채워짐을 확인할 수 있다. 다른 register나 본인에게 저장혹은 load 할때도 정상적인 값이 나오고 있음을 확인할 수 있다.



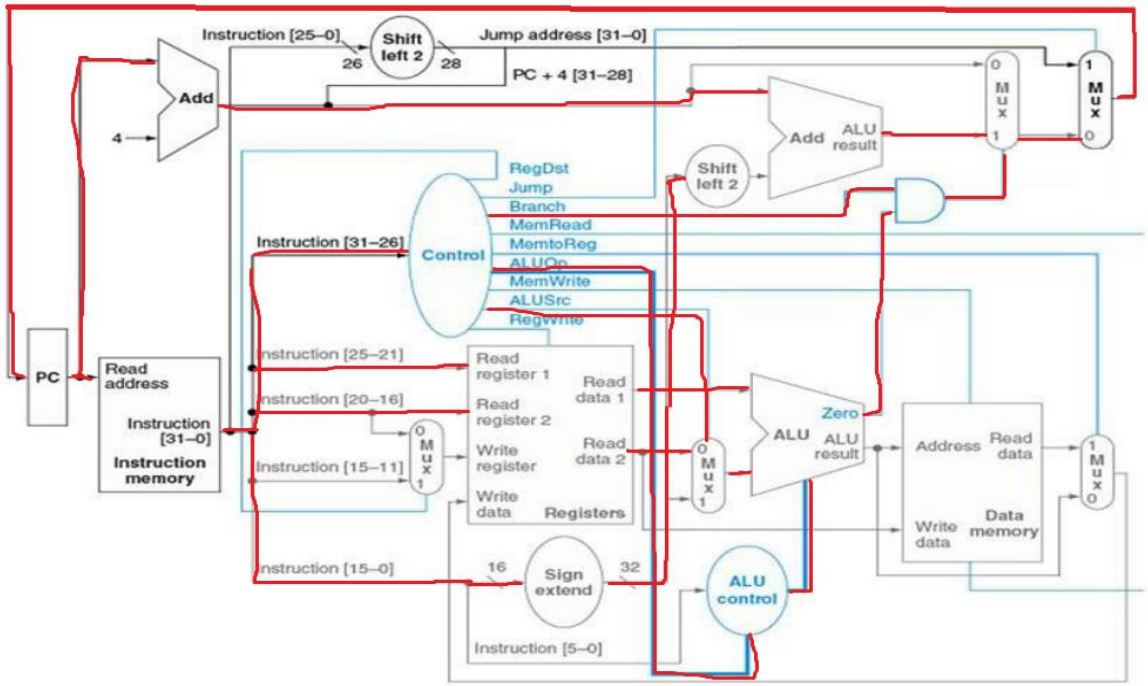
다음은 sra를 통해 shamt만큼 register5의 값이 오른쪽으로 shift되는 모습이다.

Shamt	Value
0	00000000_00000000_00001001_01101010(2)
1	000000000_00000000_00001001_0110101(2)
3	000000000000_00000000_00001001_0110(2)
27	00000000000000000000000000000000(2)
1	00000000000000000000000000000000(2)

위 결과에서 shamt값 만큼 shift right 되고 있음을 알 수 있다.

이때 MSB가 0이고 sra에서는 sign인 상태로 유지되어야 하기 때문에 shift로 인해 공백이 생긴 부분은 MSB의 값에 따라 0로 채워짐을 확인할 수 있다. 다른 register나 본인에게 저장혹은 load 할때도 정상적인 값이 나오고 있음을 확인할 수 있다.

-BNE



다음은 BNE의 진행을 주어진 회로도 위에 그려 넣은 것이다. BNE가 있는 PC값에 도달 시 PC값은 위의 ADD로 보내 ALU와 MUX를 통해 PC+4+shift left2된 branch값을 진행할 수 있도록 해준다. Instruction Memory로 향한 PC값은 Memory에서 각 bit별로 내용을 꺼내어 [31-26]은 Control, [25-21]은 regitster1=$rs, [25-21]은 regitster2=$rt, [15-0]은 sign extend로 향한다. 이에 Register에서 나오는 Read data1은 $rs, Read data2는 $rt의 값이 된다. BNE의 경우 Control에 의해 ALUOP(어떤 ALU의 기능을 쓸지 결정 위에선 00110로 a-b)와 ALUSrc(위에선 $rt)가 정해진다. 이를 기반으로 ALU가 실행된다. ALU Control로부터 동작을 받아와 Input데이터를 기반으로 ALU zero를 생성한다. 결과를 바탕으로 참(a-b가 0과 같지 않을 경우)일 경우 PC+4+shift left2된 branch값을, 거짓일 경우(a-b가 0과 같을 경우) PC+4를 PC에 보내줄 수 있도록 하고 있다. pc가 ALU 결과가 참일때($rs != $rt) pc=pc+4+shift left2된 branch값, 거짓일때 pc=pc+4가 된다.

([15-0]의 값은 sign extend로 들어가 zero extension을 하여 immediate value를 만들고 이 값을 pc 값을 정하는데 사용한다.)

(각 제어신호의 사용 이유는 명령어가 PLA_OR 아래 서술된 조건을 만족해야 하기 때문이다.)

PLA_AND: 000101_xxxxxx_xxxxx

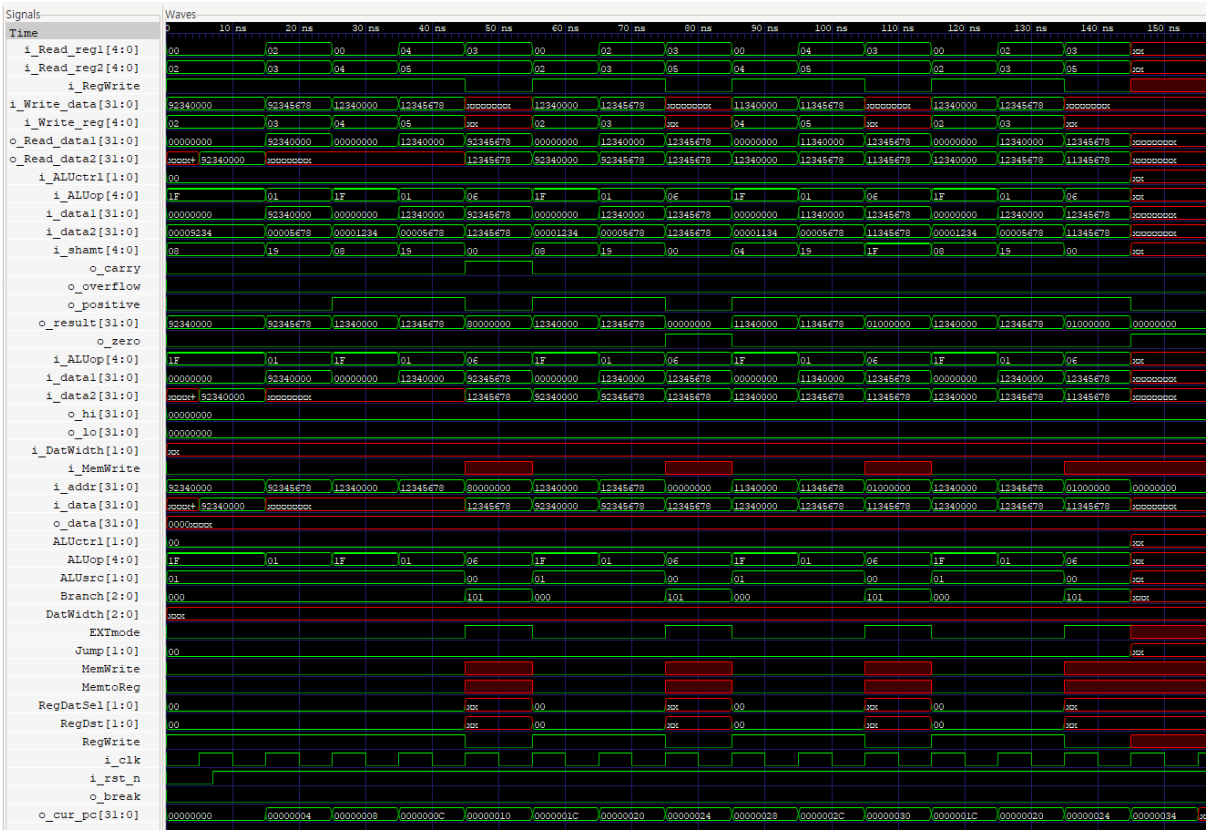
PLA_OR: xx_xx_0_1_00_00_00110_xxx_0_x_101_00_xxxxx

No destination/no destination/do not write Register/Sign Extension/Register file port B/Normal ALU input&Shift amount/a-b/Any data width for memory/no write data to memory/no data to Register file/Branch if not zero/No Jump

위 명령을 테스트 하기 위해 다음과 같은 테스트 벤치를 준비했다.

```
00111100_00000010_10010010_00110100
00110100_01000011_01010110_01111000
00111100_00000100_00010010_00110100
00110100_10000101_01010110_01111000

000101_00011_00101_00000000_00000010
001111_00000_00100_10010010_00100011
001101_00100_00101_01010110_01111000
001111_00000_00010_00010010_00110100

00110100_01000011_01010110_01111000
000101_00011_00101_00000000_00000011
001111_00000_00100_00010001_00110100
00110100_10000101_01010110_01111000

000101_00011_00101_11111111_11111010
```

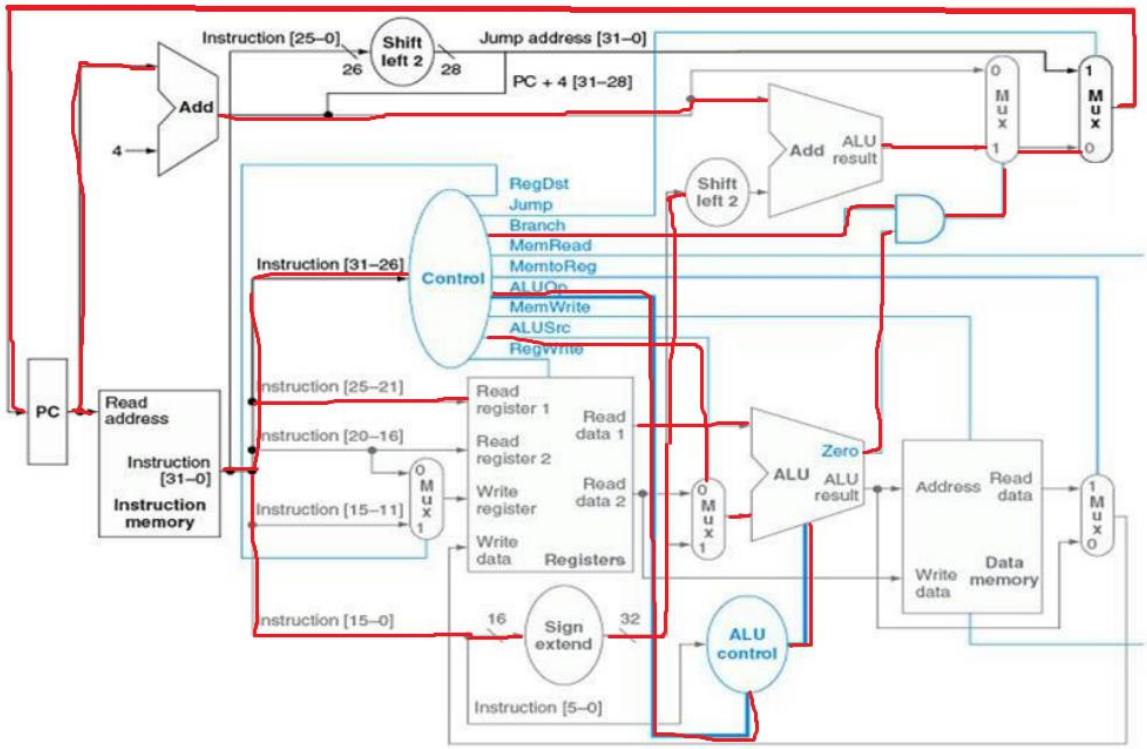
예측대로 라면 register3, 5에 0x92345678, 0x12345678을 input으로 받고, 서로 다르므로 현재 위치에서 3(기존에 주어지는 +1포함)만큼 이동한다. 이동한 곳에서 register3의 값이 0x12345678로 변경되고 이들은 같으므로 다음 BNE를 무시한다. 이후 register5의 값을 0x11345678로 변경한다. 이때문에 다음 BNE에서 -5(기존에 주어지는 +1포함)만큼 이동하게 된다. 이후 register3에 0x12345678값을 넣고 BNE에 걸려 +4(기존에 주어지는 +1포함)만큼 이동하게 된다.



다음은 테스트 벤치를 돌린 결과이다.

위의 값 변화는 예측과 같다는 것을 확인할 수 있다. 실제로 input2개가 다를 때 BNE가 걸린 부분은 입력된 값만큼 이동한 것을 확인할 수 있었다.(0x10->0x1C, 0x30->0x1C,0x24->0x34) 또한 두 input값이 같을 때는 작동하지 않는 것을 확인할 수 있었다.(0x24->0x28)

-BLEZ



다음은 BLEZ의 진행을 주어진 회로도 위에 그려 넣은 것이다. BLEZ가 있는 PC값에 도달 시 PC값은 위의 ADD로 보내 ALU와 MUX를 통해 PC+4+shift left2된 branch값을 진행할 수 있도록 해준다. Instruction Memory로 향한 PC값은 Memory에서 각 bit별로 내용을 꺼내어 [31-26]은 Control, [25-21]은 regitster1=$rs, [15-0]은 sign extend로 향한다. 이에 Register에서 나오는 Read data1은 $rs의 값이 된다. BLEZ의 경우 Control에 의해 ALUOP(어떤 ALU의 기능을 쓸지 결정 위에선 00100로 a+b)와 ALUSrc(위에선 zero)가 정해진다. 이를 기반으로 ALU가 실행된다. ALU Control로부터 동작을 받아와 Input데이터를 기반으로 ALU zero를 생성한다. ALU 결과를 바탕으로 참일 경우 PC+4+shift left2된 branch값을, 거짓일 경우 PC+4를 PC에 보내줄 수 있도록 하고 있다. pc가 ALU 결과가 참일때($rs<0) pc=pc+4+shift left2된 branch값, 거짓일때 pc=pc+4가 된다.

([15-0]의 값은 sign extend로 들어가 zero extension을 하여 immediate value를 만들고 이 값을 pc값을 정하는데 사용한다.)

(각 제어신호의 사용 이유는 명령어가 PLA_OR 아래 서술된 조건을 만족해야 하기 때문이다.)

PLA_AND: 000110_xxxxxx_xxxxx

PLA_OR: xx_xx_0_1_10_00_00100_xxx_0_x_110_00_xxxxx

No destination/no destination/do not write Register/Sign Extension/Zero/Normal ALU input&Shift amount/a+b/Any data width for memory/no write data to memory/no data to Register file/Branch if not positive/No Jump

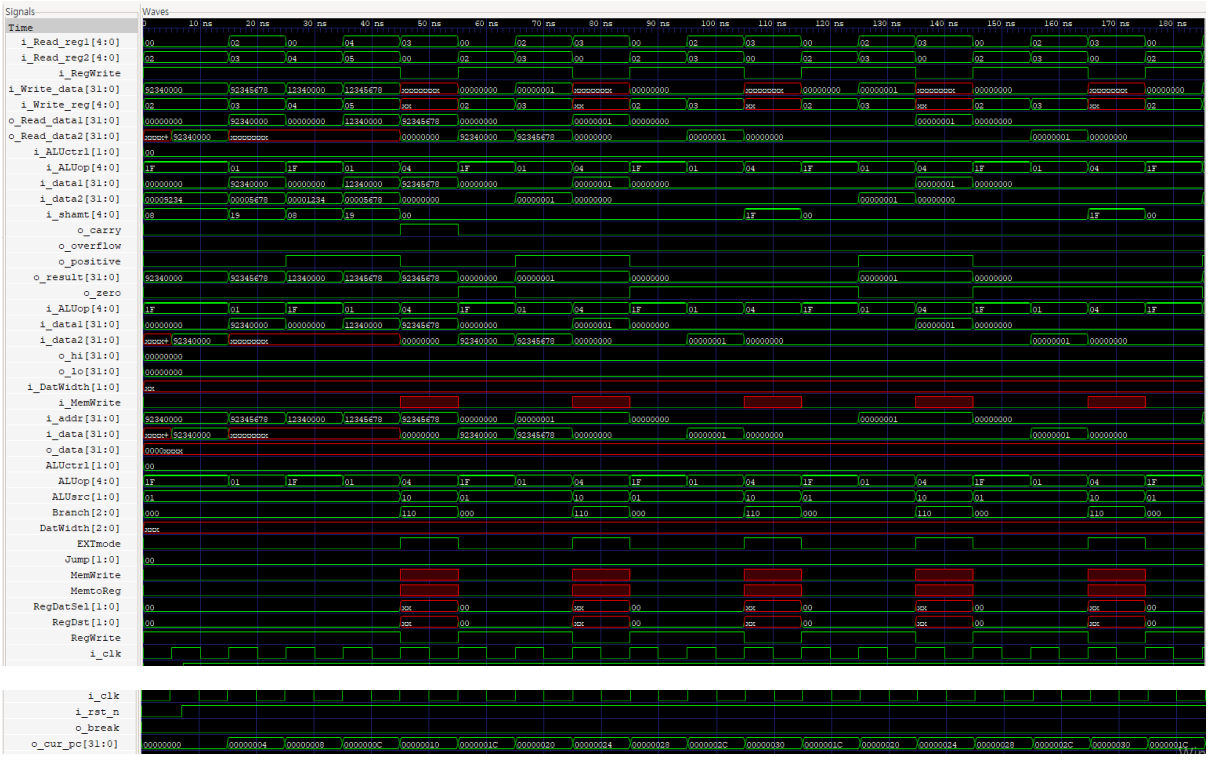
00111100_00000010_10010010_00110100
00110100_01000011_01010110_01111000
00111100_00000100_00010010_00110100
00110100_10000101_01010110_01111000

000110_00011_00000_00000000_00000010
001111_00000_00100_10010010_00100011
001101_00100_00101_01010110_01111000
001111_00000_00010_00000000_00000000

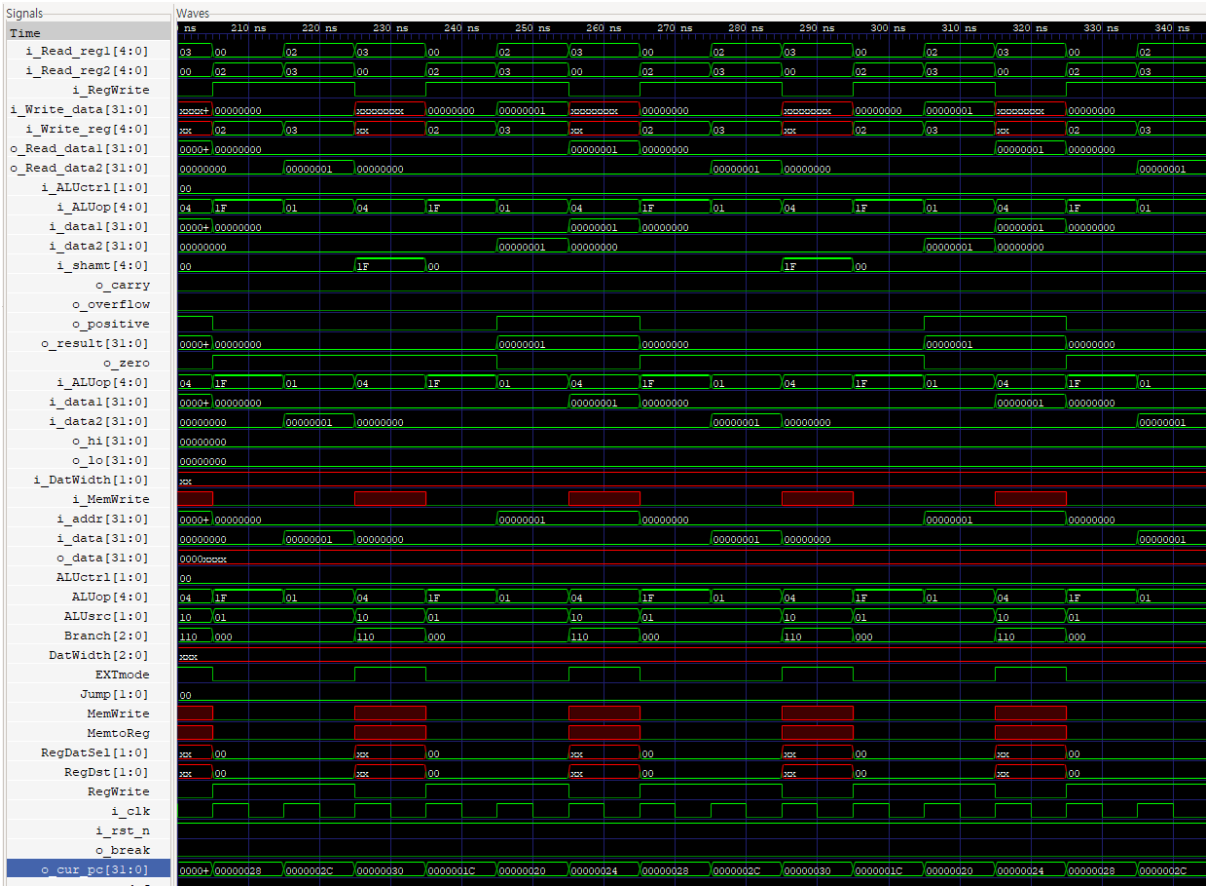
001101_00010_00011_00000000_00000001
000110_00011_00000_00000000_00000011
001111_00000_00010_00000000_00000000
001101_00010_00011_00000000_00000000

000110_00011_00000_11111111_11111010

예측대로 라면 register3에 0x92345678을 input으로 받고, register3에 MSB가 1인 음수 값이 들어갔으므로 현재 위치에서 3(기존에 주어지는 +1포함)만큼 이동한다. 이동한 곳에서 register3의 값이 0x00000001로 변경되고 이는 0보다 크므로 다음 BNE를 무시한다. 이후 register3의 값을 0x00000000로 변경한다. 이때문에 다음 BNE에서 -5(기존에 주어지는 +1포함)만큼 이동하게 된다. 이후 register3에 0x92235678값을 넣고 이후 0x00000001값을 넣어준다. 이 때문에 0x30에서 0x1C로 계속해서 무한루프가 발생하게 될 것이다.

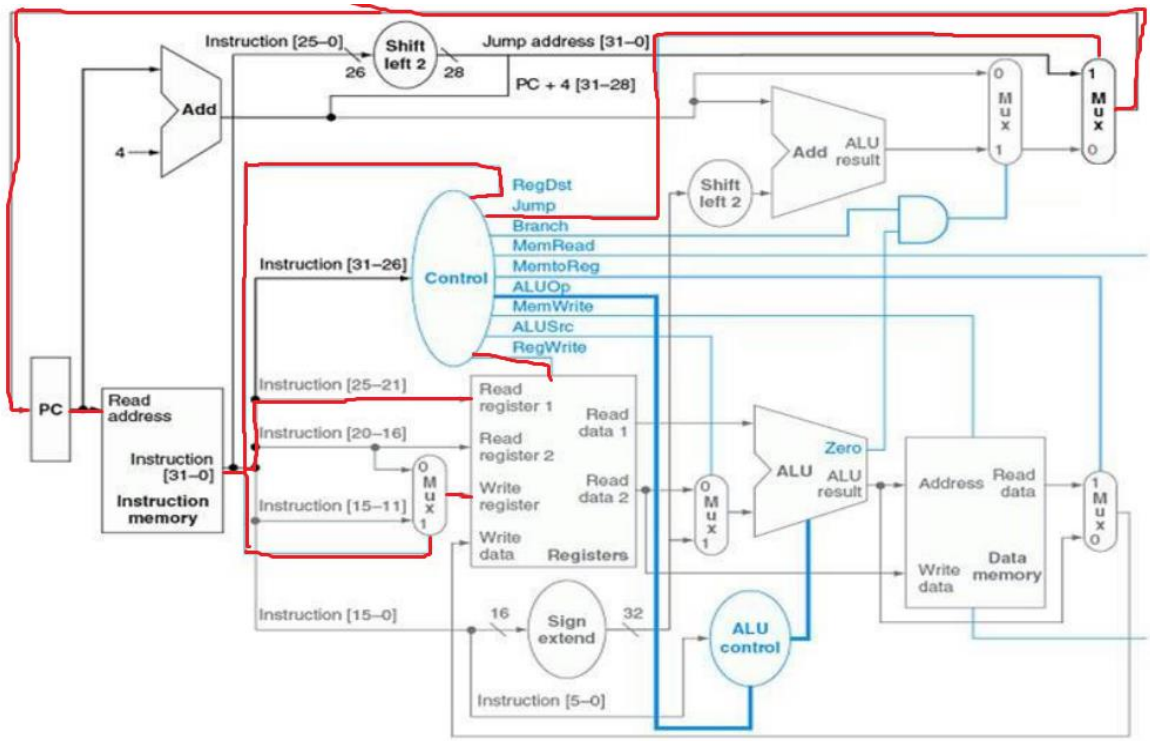


해당 테스트벤치를 돌려본 결과는 다음과 같다. 위의 값 변화는 예측과 같다는 것을 확인할 수 있다. 실제로 input의 값이 음수일 때와 0일 때 BLEZ가 걸린 부분은 입력된 값만큼 이동한 것을 확인할 수 있었다.(0x10->0x1C, 0x30->0x1C) 또한 두 input값이 같을 때는 작동하지 않는 것을 확인할 수 있었다.(0x24->0x28) 끝으로 의도된 대로 0x30->0x1C가 계속해 발생하게 되어 무한루프가 생겨남을 확인할 수 있었다.



(200ns이후의 모습. 무한루프가 발생함을 pc값을 통해 확인할 수 있다.)

-JALR



다음은 JALR의 진행을 회로 위에 표시한 것이다. PC를 통해 받은 값이 Instruction에 의해 구체화 및 분배되어 Control, Register에 값이 들어간다. 이때문에 Control의 RegDst, Jump, RegWrite가 활성화 된다. 이 Jump의 활성화로 PC에 들어가는 값이 Jump에 사용될 $rs값이 된다.(단, 위 회로에서는 이가 구현될 수 없다. 이를 맞게 구현하려면 Read data1에서 값을 받아 PC에 넣어줄 수 있어야 한다.) 또한 RegWrite에 의해 Register에 값을 쓸 수 있게 된다. 이때 RegDst에 의해 Write register에 $31의 값이 들어가게 된다. 고로 RegWrite인 $31에 Write Data가 입력될 것이다.(이때 Write data로는 ALU result만을 받아올 수 있으므로 pc의 값을 받아올 수 없다. 이 때문에 위 회로에서 $31=pc는 구현될 수 없다.)

(각 제어신호의 사용 이유는 명령어가 PLA_OR 아래 서술된 조건을 만족해야 하기 때문이다.)

PLA_AND:000000_001001_xxxxx(R-type이므로 opcode는 000000이 되고 function이 001001이다.)

PLA_OR:10_11_1_x_xx_xx_xxxxx_xxx_0_x_xxx_10_xxxxx

$31/Write PC to Register file/write to Register file/No Extension/No ALU/ No ALU / No ALU /Any data width for memory/no write data to memory/no data to Register file/No branch/Use $rs

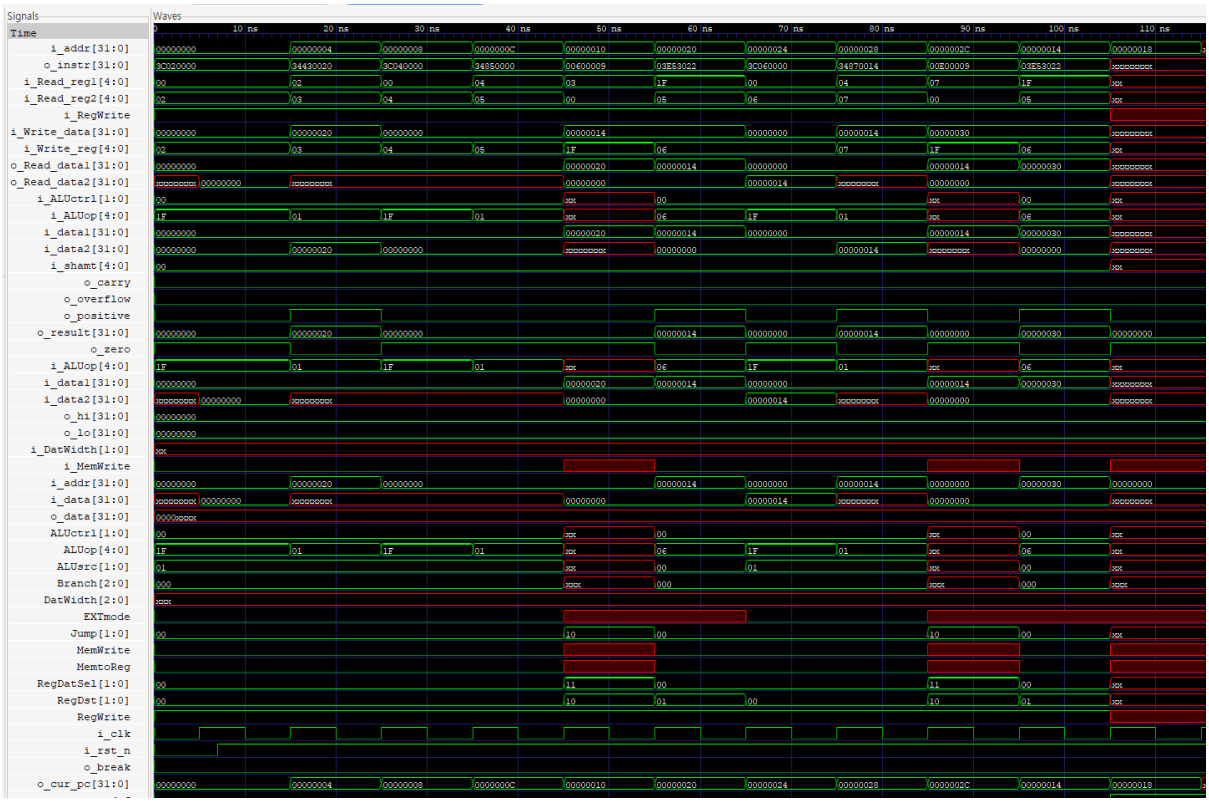
JALR의 테스트 벤치는 아래/위로의 JUMP를 확인함과 동시에 미리 구현된 SUB를 이용해 $31에 값이 저장되었는지 확인하는 것을 목적으로 하고 있다.

00111100_00000010_00000000_00000000
00110100_01000011_00000000_00100000
00111100_00000100_00000000_00000000
00110100_10000101_00000000_00000000

00000000_01100000_00000000_00001001
00000011_11100101_00110000_00100010
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx

00000011_11100101_00110000_00100010
00111100_00000110_00000000_00000000
00110100_10000111_00000000_00010100
00000000_11100000_00000000_00001001

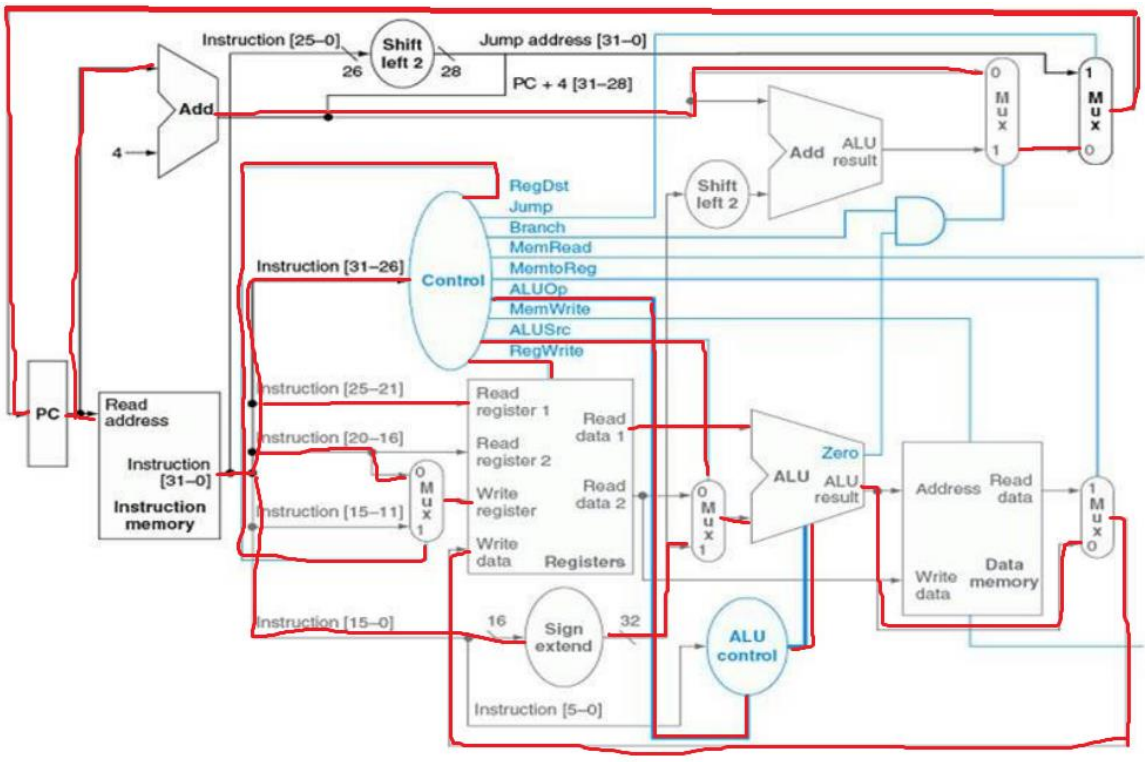
예상대로라면 넣어진 input값들을 바탕으로 register3에 저장된 0x00000020으로 jump 후 $31에 저장된 값에서 0을 뺀 값을 출력해 $31에 저장된 값(0x00000014)을 확인할 것이다. 이후 register7에 넣어진 0x00000014으로 jump 후 $31에 저장된 값에서 0을 뺀 값을 출력해 $31에 저장된 값(0x00000030)을 확인하고 프로그램이 종료될 것이다.



위의 결과를 보면 예측한 대로 값이 나온 것을 확인할 수 있다.

JALR에 의해 동작하는 부분의 o_cur_pc값을 보면 0x00000010->0x00000020으로, 0x0000002C->0x00000014로 잘 JUMP함을 확인할 수 있으며 JUMP한 곳에서 확인한 $31-0의 값은 JUMP했던 위치의 바로 다음인 0x00000014, 0x00000030이 알맞게 들어가 있다.

-ANDI



다음은 ANDI의 진행을 주어진 회로도 위에 그려 넣은 것이다. ANDI가 있는 PC값에 도달 시 PC 값은 위의 ADD로 보내 ALU와 MUX를 통해 PC+4를 진행할 수 있도록 해준다. Instruction Memory로 향한 PC값은 Memory에서 각 bit별로 내용을 꺼내어 [31-26]은 Control, [25-21]은 regitster1=$rs, [20-16]은 register2=$rt와 mux, [15-11]=$rd은 mux, [15-0]은 sign extend와 ALU control로 향한다. ANDI의 경우 Control에 의해 Write register는 [20-16]bit 값에 해당하는 register가 된다. RegWrite가 활성화 되므로 결과적으로 레지스터에 결과가 저장될 것이다. 또한 Control에 의해 ALUOP(어떤 ALU의 기능을 쓸지 결정 위에선 00000로 Bitwise AND)와 ALUSrc(위에선 immediate value의 값을 사용한다)가 정해진다. [15-0]bit였던 값은 sign extend를 통해 32bit value가 되어 ALU 직전의 Mux에 입력된다.(이를 immediate value이라 하며 zero extension을 수행한다.) 이를 기반으로 ALU가 실행된다. ALU Control로부터 동작을 받아와 Input데이터를 기반으로 ALU result를 생성한다. MemtoReg가 0이므로 ALU result가 Write data로 이동, 결과적으로 $rs와 immediate value가 bitwise and된 값이 $rt에 저장되고 pc=pc+4가 된다.

(각 제어신호의 사용 이유는 명령어가 PLA_OR 아래 서술된 조건을 만족해야 하기 때문이다.)

PLA_AND: 001100_xxxxxx_xxxxx

PLA_OR:00_00_1_0_01_00_00000_xxx_0_0_000_00_xxxxx

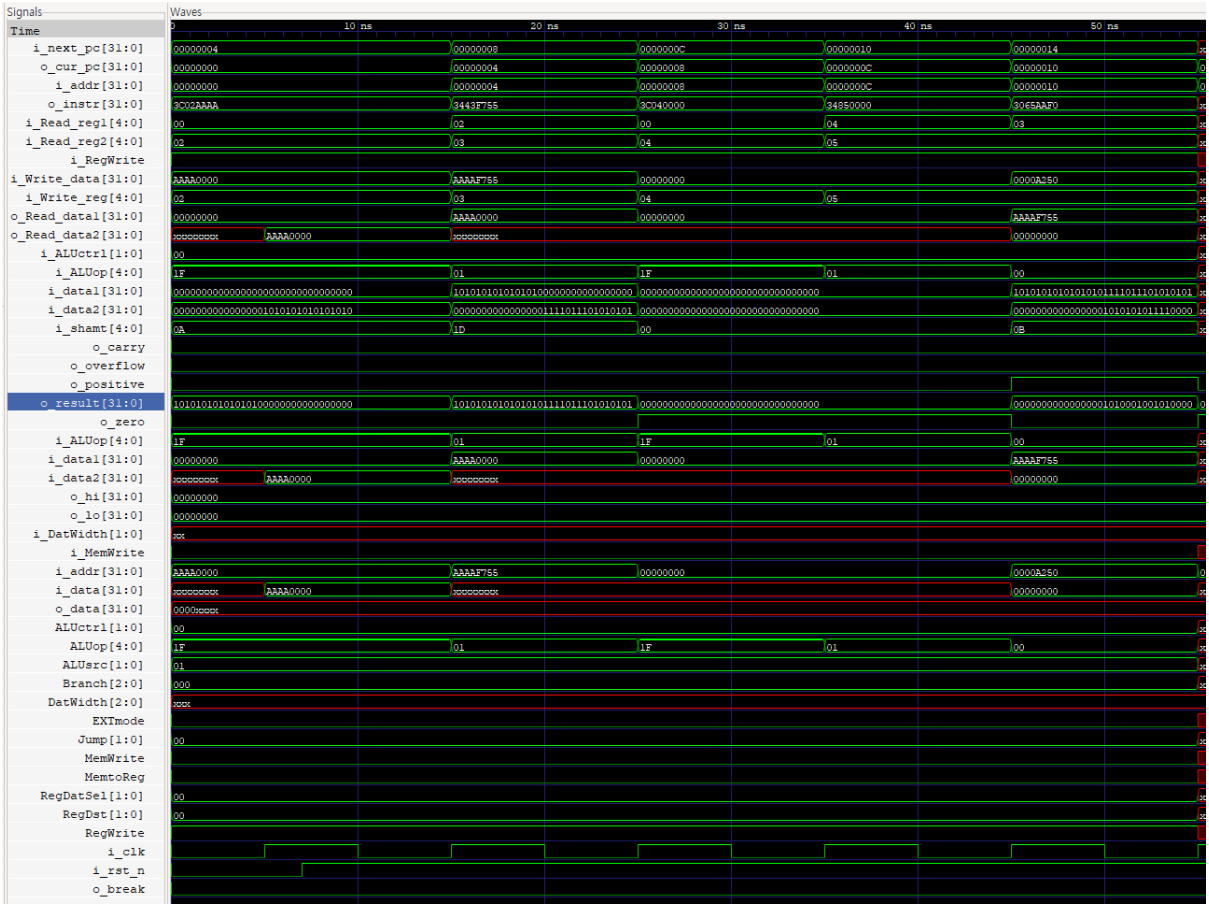
$rt/Write ALU to Register file/write to Register file/zero Extension/Immediate/ No ALU / No ALU /Any data width for memory/no write data to memory/no data to Register file/No branch/Use $rs

각 bit가 (0,0) (0,1) (1,0) (1,1)인지를 확인해서 (1,1)일 때만 해당 bit를 1로 바꿔 줌을 확인해야한다. 이때 ANDI는 zero extension이므로 31-16bit는 0이 될 것이다. 이를 확인하기 위해 다음 testbench를 넣어주었다.

Register3: 10101010_10101010_11110111_01010101(2)

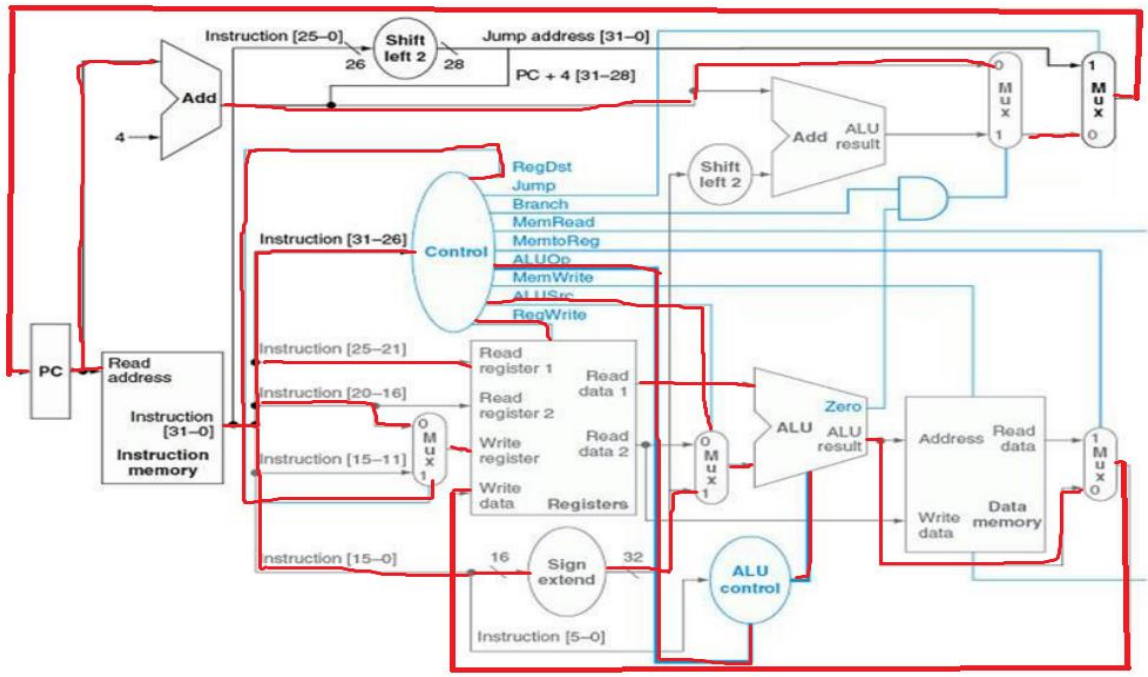
Immediate value:10101010_11110000(2)

Zero extension이므로 immediate value가 00000000_00000000_10101010_11110000으로 판정되어 예상되는 값은 00000000_00000000_10100010_01010000(2)이다.



위는 해당 test bench를 넣어본 결과이다. 예측된 값과 같이 immediate value가 00000000_00000000_10101010_11110000으로 판정되어 계산되었다. 예상한 대로 결과 값은 00000000_00000000_10100010_01010000(2)으로 i_Write_reg를 바탕으로 register 5에 이 값이 들어갔음을 확인할 수 있다.

-SLTIU



다음은 SLTIU의 진행을 주어진 회로도 위에 그려 넣은 것이다. SLTIU가 있는 PC값에 도달 시 PC 값은 위의 ADD로 보내 ALU와 MUX를 통해 PC+4를 진행할 수 있도록 해준다. Instruction Memory로 향한 PC값은 Memory에서 각 bit별로 내용을 꺼내어 [31-26]은 Control, [25-21]은 regitster1=$rs, [20-16]은 register2=$rt와 mux, [15-11]=$rd은 mux, [15-0]은 sign extend와 ALU control로 향한다. SLTIU의 경우 Control에 의해 Write register는 [20-16]bit 값에 해당하는 register가 된다. RegWrite가 활성화 되므로 결과적으로 레지스터에 결과가 저장될 것이다. 또한 Control에 의해 ALUOP(어떤 ALU의 기능을 쓸지 결정 위에선 10001로 Unsigned SLT)와 ALUSrc(위에선 immediate value의 값을 사용한다)가 정해진다. [15-0]bit였던 값은 sign extend를 통해 32bit value가 되어 ALU 직전의 Mux에 입력된다.(이 값을 immediate value라고 하며 이때 zero extension을 한다.) 이를 기반으로 ALU가 실행된다. ALU Control로부터 동작을 받아와 Input데이터를 기반으로 ALU result를 생성한다. 이때 ALU의 결과로 $rs의 값이 immediate의 값보다 작으면 1, 크면 0이 출력된다. MemtoReg가 0이므로 ALU result가 Write data로 이동, 결과적으로 $rs의 값이 immediate의 값보다 작으면 1, 크면 0이 $rt에 저장된다.

(각 제어신호의 사용 이유는 명령어가 PLA_OR 아래 서술된 조건을 만족해야 하기 때문이다.)

PLA_AND:001011_xxxxxx_xxxxx

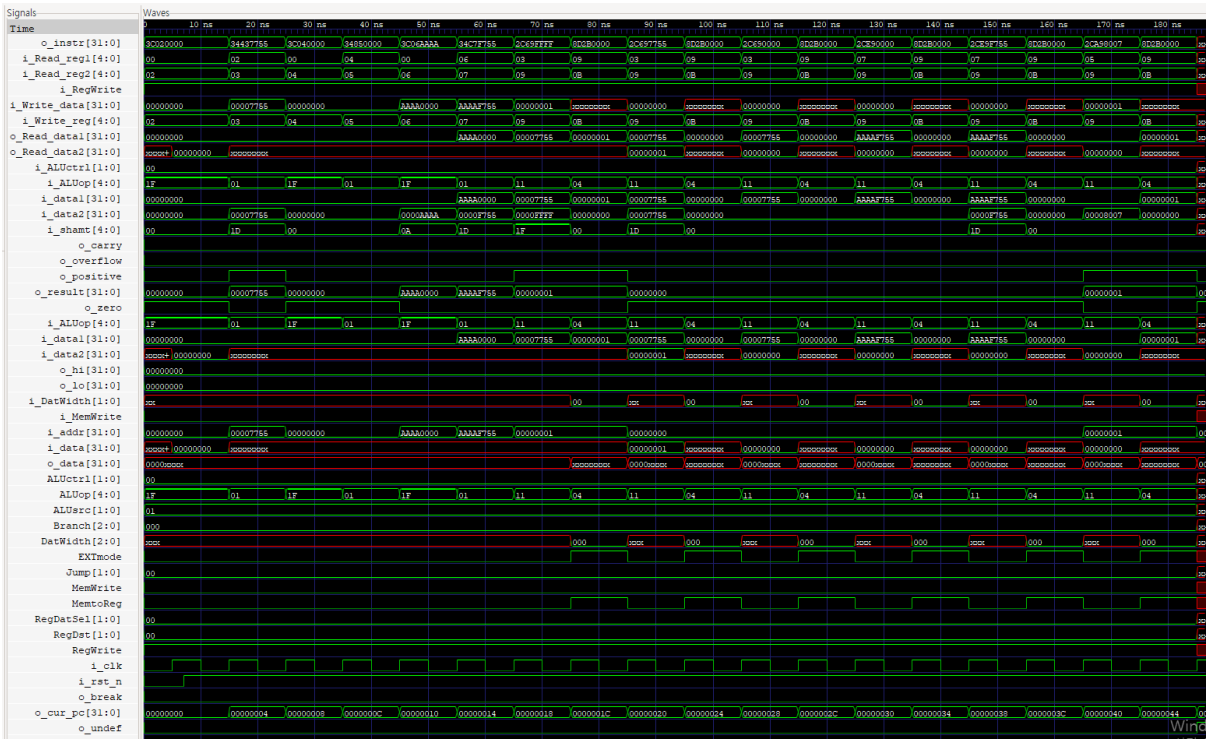
PLA_OR:00_00_1_0_01_00_10001_xxx_0_0_000_00_xxxxx

$rt/Write ALU to Register file/write to Register file/zero Extension/Immediate/ Normal ALU input, shift Amount(no shift) / Unsigned SLT /Any data width for memory/no write data to memory/ALU data to Register file/Use PC+4/Do not use Jump address

테스트 벤치는 register값의 MSB가 0일경우 Immediate값이 더 클 경우, 0일 경우, 더 작을 경우와 register값의 MSB가 1일 경우에서 Immediate값이 0일 경우, MSB가 1일 경우, register값이 0, immediate값이 양수일 경우를 비교해 보았다. 해당 테스트를 위해 register들에는 다음과 같은 값들을 넣었다.

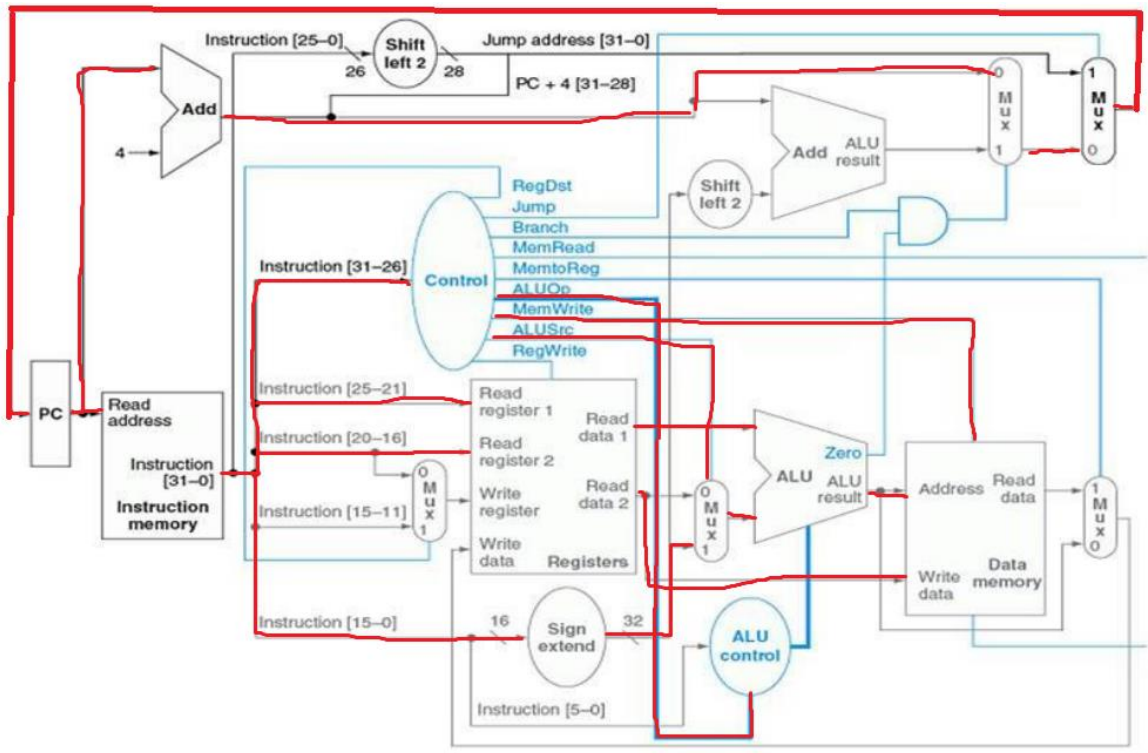
Register3:0x00007755 / Register5:0x00000000 / Register7:0xAAAAF755

Register3에 대해 0xFFFF, 0x7755, 0x0000를, Register7에 대해 0x0000,0xF755, Register5에 대해 0x8007을 대입해 주었다. 이 때 해당 명령어는 unsigned에서 작동하기 때문에 0x00000001, 0x00000000,0x00000000,0x00000000,0x00000000,0x00000001 순으로 register에 저장될 것이다. 이 때 저장되는 register는 9이다.



위 testbench로 결과를 본 결과 다음과 같이 나오게 되었다. 예측된 대로 register 9에 0x00000001, 0x00000000,0x00000000,0x00000000,0x00000000,0x00000001 순으로 저장된 것을 확인할 수 있었다.

-SB



다음은 SB의 진행을 주어진 회로도 위에 그려 넣은 것이다. SB가 있는 PC값에 도달 시 PC값은 위의 ADD로 보내 ALU와 MUX를 통해 PC+4값을 진행할 수 있도록 해준다. Instruction Memory로 향한 PC값은 Memory에서 각 bit별로 내용을 꺼내어 [31-26]은 Control, [25-21]은 regitster1=$rs, [20-16]은 register2=$rt, [15-0]은 sign extend로 향한다. 이에 Register에서 나오는 Read data1은 $rs의 값이 된다. 또한 [15-0]이 향한 sign extend에선 sign extend가 발생, immediate value가 만들어진다. SB의 경우 Control에 의해 ALUOP(어떤 ALU의 기능을 쓸지 결정 위에선 00100로 a+b)와 ALUSrc(위에선 immediate value)가 정해진다. 이를 기반으로 ALU가 실행된다. ALU Control로부터 동작을 받아와 Input데이터를 기반으로 ALU result를 생성한다. ALU 결과를 바탕으로 Data memory에서 저장할 주소를 정한다. 또한 이를 Read data2에서 나오는 $rt의 값을 넣어준다. pc=pc+4가 된다.

(각 제어신호의 사용 이유는 명령어가 PLA_OR 아래 서술된 조건을 만족해야 하기 때문이다.)

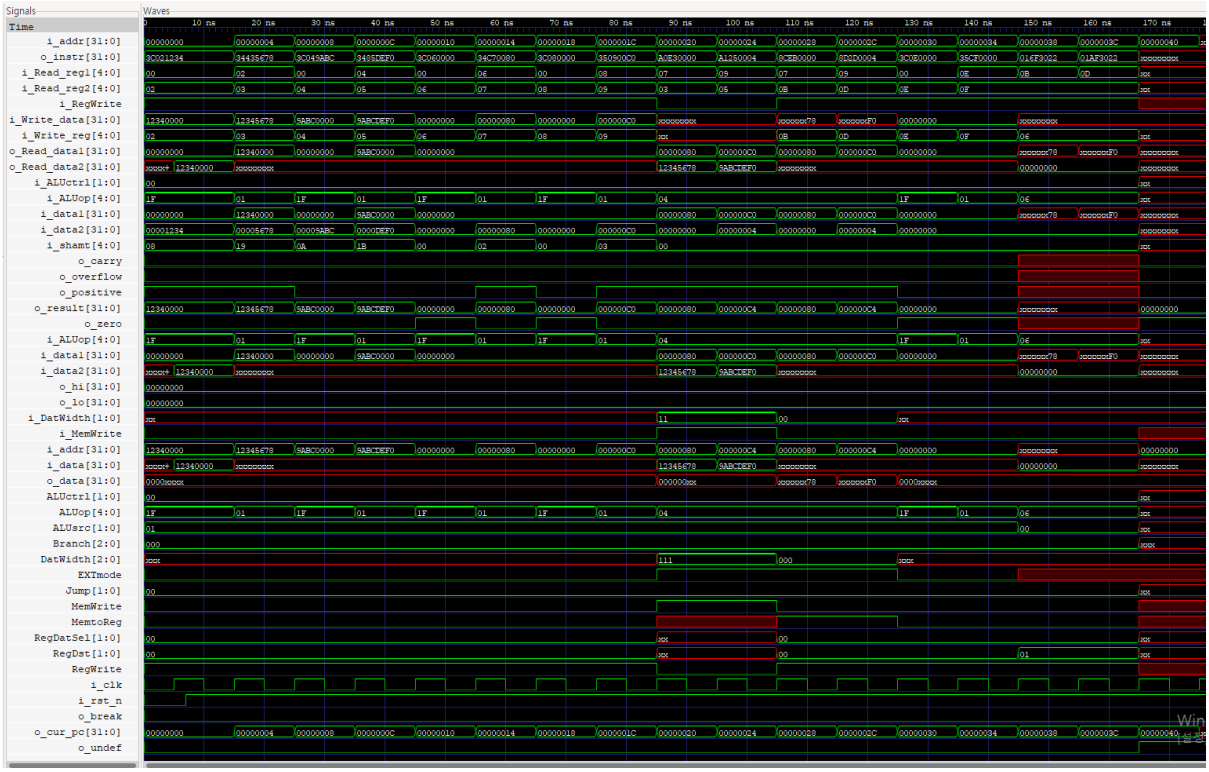
PLA_AND: 101000_xxxxxx_xxxxx

PLA_OR: xx_xx_0_1_01_00_00100_011_1_x_000_00_xxxxx

No register/No Register/No write to Register file/sign Extension/Immediate/ Normal ALU input, shift Amount/ A+B /8-bit byte/write data to memory/No write to Register/Use PC+4/Do not use Jump address

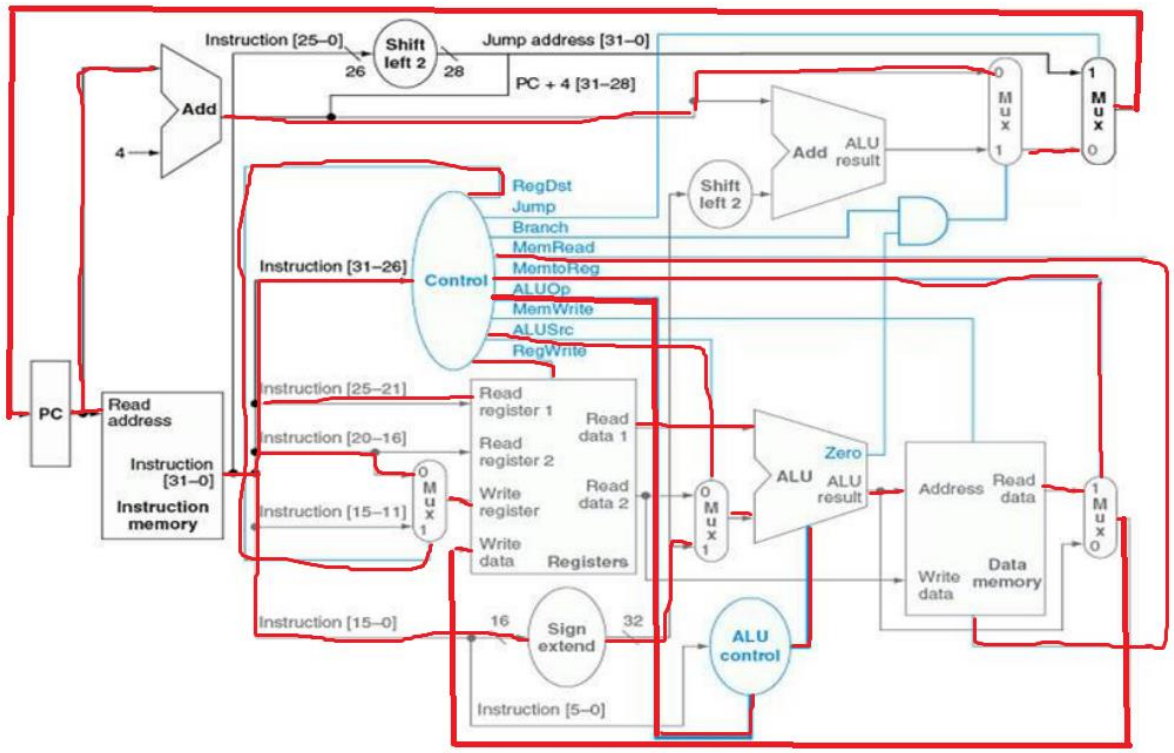
SB를 테스트하기 위해 다음과 같은 테스트를 진행하였다.

해당 명령어는 Memory의 지정된 위치에 $rs에 있는 값을 8bit 값으로 저장하는 역할을 한다. 이를 위해 Register3=0x12345678 Register4=0x9ABCDEF0을 주고 해당 값들을 MEM($s+i):1에 sb를 통해 저장하게 해주었다. 이후 해당 값을 잘 가져오며 메모리가 주변에 영향을 주지 않았는지 확인하기 위해 Lw를 call하였다. 이때 예측된 값은 하위 8bit만을 남긴 0xxxxxxx78, 0xxxxxxxF0이 출력될 것이라 예측했다. (해당 프로그램은 없는 값을 x로 감지해주기 때문에 lw로 받으면 하위 8bit는 받아져 보이지만 나머지 memory는 비어서 x로 출력될 것이다.)이는 돌아온 값은 sub를 통해 i_data1를 확인했다. (해당 SB는 하위 8bit를 저장한다.)



예상대로 SB를 통해 하위 8bit가 메모리에 저장됨을 확인할 수 있었다. 또한 Lw를 통해 값을 받아 초기화되지 않은 부분을 x로 표시하고 하위 8bit에 sb를 통해 저장된 값들을 표현해주고 있었다.

-LBU



다음은 LBU의 진행을 주어진 회로도 위에 그려 넣은 것이다. LBU가 있는 PC값에 도달 시 PC값은 위의 ADD로 보내 ALU와 MUX를 통해 PC+4값을 진행할 수 있도록 해준다. Instruction Memory로 향한 PC값은 Memory에서 각 bit별로 내용을 꺼내어 [31-26]은 Control, [25-21]은 regitster1=$rs, [20-16]은 mux를 통해 Write register로, [15-0]은 sign extend로 향한다. 이에 Register에서 나오는 Read data1은 $rs의 값이 된다. 또한 [15-0]이 향한 sign extend에선 sign extend가 발생, immediate value가 만들어진다. LBU의 경우 Control에 의해 ALUOP(어떤 ALU의 기능을 쓸지 결정 위에선 00100로 a+b)와 ALUSrc(위에선 immediate value)가 정해진다. 이를 기반으로 ALU가 실행된다. ALU Control로부터 동작을 받아와 Input데이터를 기반으로 ALU result를 생성한다. ALU 결과를 바탕으로 Data memory에서 불러올 주소를 정한다. 또한 MemtoRead와 MemtoREG가 활성화 되어 있으므로 ALU result에 해당하는 주소에 존재하는 Memory의 값을 Write data로 넘겨주게 될 것이다. 이를 Write register로 지정된 $rt에 넣어준다. pc=pc+4가 된다.

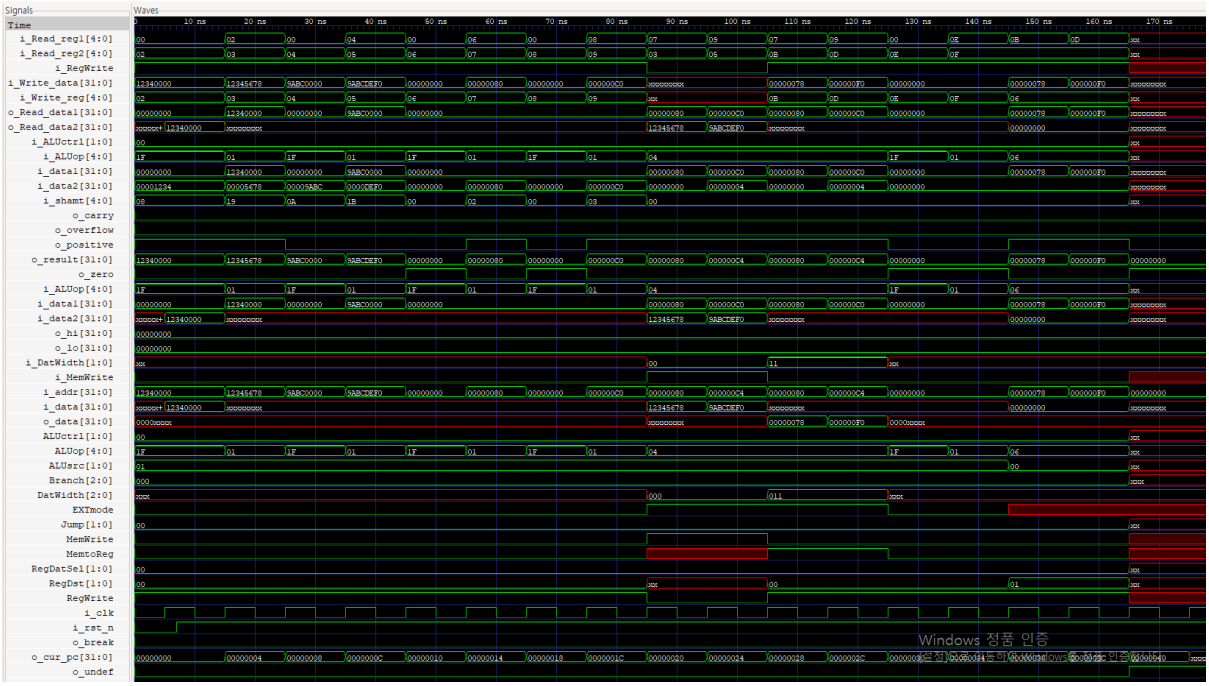
(각 제어신호의 사용 이유는 명령어가 PLA_OR 아래 서술된 조건을 만족해야 하기 때문이다.)

PLA_AND: 100100_xxxxxx_xxxxx

PLA_OR: 00_00_1_1_01_00_00100_011_0_1_000_00_xxxxx

$rt/Write MEM to Register file/write to Register file/sign Extension/Immediate/Normal ALU input, shift Amount / a+b /8-bit byte/no write data to memory/Memory data to Register file/Use PC+4/Do not use Jump address

해당 명령어는 Memory의 지정된 위치의 값을 unsigned형태의 8bit값으로 불러오는 역할을 한다. 이를 위해 Register3=0x12345678 Register4=0x9ABCDEF0을 주고 해당 값들을 MEM($s+i):1에 sw를 통해 저장하게 해주었다. 이후 해당 값을 lbu가 잘 가져오는 지 확인하였다. 이때 예측된 값은 하위 8bit만을 남긴 0x00000078, 0x000000F0이 출력될 것이라 예측했다. 돌아온 값은 sub를 통해 0을 빼어 확인했다. (해당 MIPS는 Big Endian으로 돌아가므로 하위 8bit가 남는다.)



예상대로 input값으로 register3와 5으로부터 0x12345678과 0x9ABCDEF0을 받았으며 lbu로 레지스터에 저장하였을 때 하위 8bit인 0x78과 0xF0가 저장됨을 확인할 수 있었다.

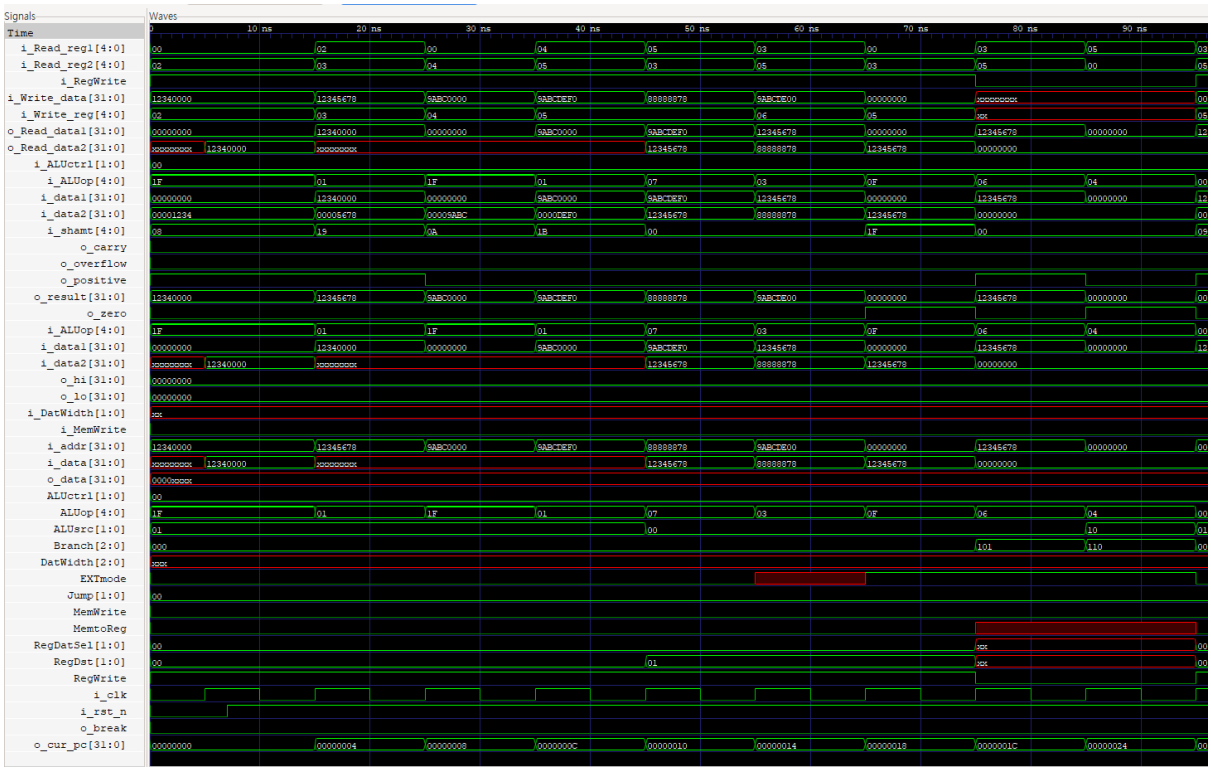
-전체 명령어 테스트

다음 명령어들을 조합했을 때 작동이 잘 되는지 확인하기위해 다음과 같은 테스트를 진행하였다.

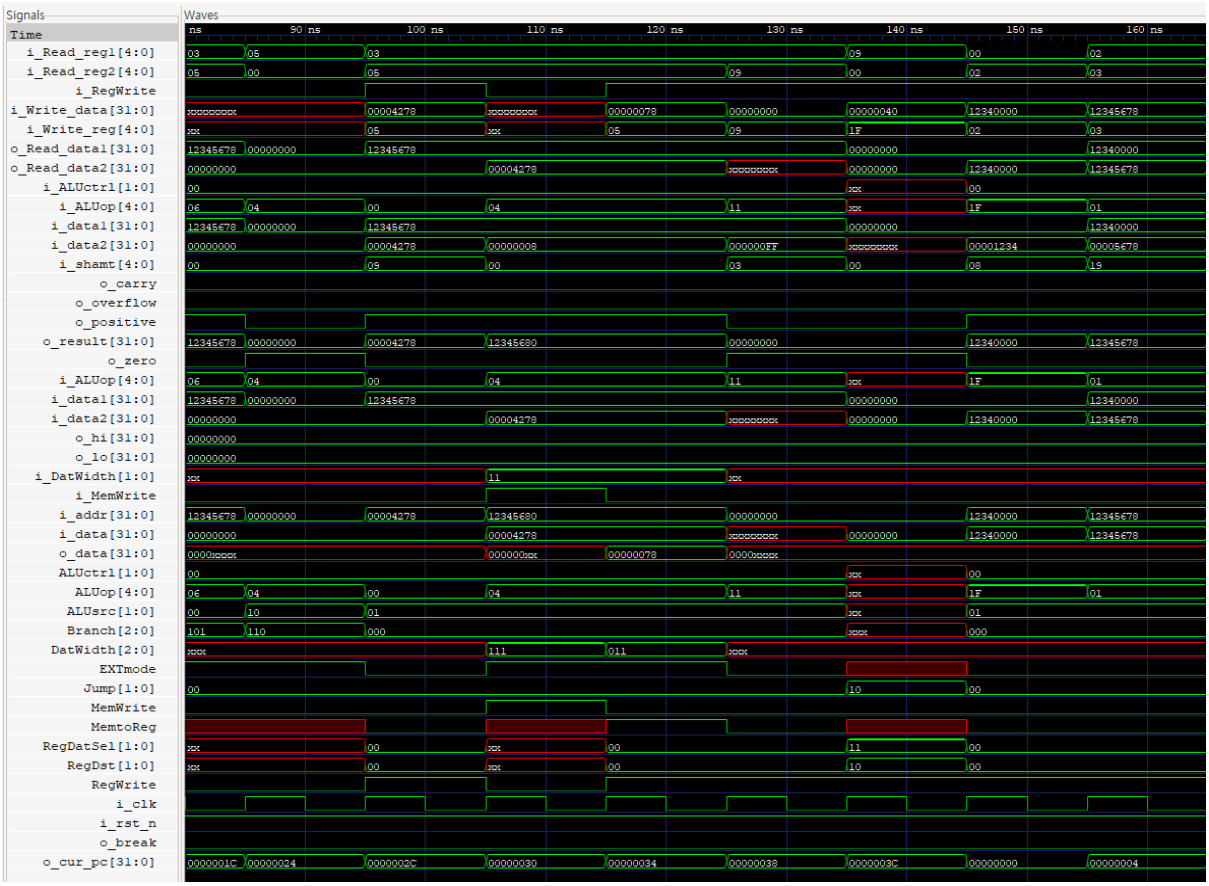
```
00111100_00000010_00010010_00110100
00110100_01000011_01010110_01111000
00111100_00000100_10011010_10111100
00110100_10000101_11011110_11110000

000000_00101_00011_00101_00000_100011    //sub r5-r3              result:r5=0x88888878
000000_00011_00101_00110_00000100110     //xor r3,r5              result:r6=0x9ABCDE00
000000_00000_00011_00101_11111_000011    //sra  r3 >>31 =>r5  result:r5=0x00000000
000101_00011_00101_00000000_00000001     //bne r3!=r5             result:next pc=0x00000024

xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
000110_00101_00000_00000000_00000001     //blez r5<=0         result:next pc=0x0000002C
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
001100_00011_00101_01000010_01111000     //andi r3,imme       result:e5=0x00004278

101000_00011_00101_0000000000000000      //sb r5 in memory[r3]
100100_00011_00101_0000000000000000      //lbu r5 from memory[r3]        result:r5=0x00000078
001011_00011_01001_00000000_11111111     //sltiu if r3<0x00FF put 1 to r9  result:r9=0x00000000
000000_01001_00000_00000000_00001001     //jalr //save pc and jump to r9  result:next pc=0x00000000
```

*입력된 값/동작/예상 결과이다. 마지막 jalr이 0으로 jump하므로 이는 무한히 반복한다.

실행결과는 다음과 같다.



Pc 0x00000000~0x00000024구간에서 예측과 같은 결과가 보여진다.



0x00000024~0x0000003C구간도 예측과 같은 결과를 보이고 있음을 알 수 있다.

이후 0으로 Jump해 무한loop가 발생하게 된다.

결과 및 고찰

결과적으로 모든 명령어의 testbench를 통한 test및 동작을 확인했다. 해당 회로를 구현 및 동작시켜보면서 MIPS에 대한 개념을 더 확립할 수 있었고, MIPS에서 사용되는 format에 대한 이해도 또한 상승했다. 이를 메모장을 통해 0과 1로 코드를 만드는 것도 새로운 경험이었다. 다만 회로상에서 mux에 대한 오류나 jalr명령어에 대한 회로를 통한 datapath 묘사 불가와 같은 아쉬운 점들이 있었다. 그러나 새로운 방법을 찾아내고 오류를 간파해내며 회로 수정 등을 하여 이해도를 더욱 높일 수 있었다.

Reference

MIPS Inst Datasheet.pdf

Control Signals for SCPU.pdf

프로젝트1 제안서 2022.pdf