컴퓨터구조 Report

Project #1 – MIPS Single Cycle CPU

담당교수: 이성원 교수님

실습분반: 수 7교시

학번: 2019202050

이름: 이강현

[Introduction]

이번 프로젝트는 single cycle cpu를 구현해보는 것으로 다음과 같은 명령어들의 PLA_AND.txt, PLA_OR.txt를 구현해보고 각 명령어들의 동작방식을 이해하고 설명한다.

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 shift 을 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]

PLA_AND

Instruction	Opcode	Func	Regimm
ADDU	000000	100001	xxxxx
OR	000000	100101	xxxxx
ADDIU	001001	xxxxxx	xxxxx
XORI	001110	xxxxxx	xxxxx
SLL	000000	000000	xxxxx
SRAV	000000	000111	xxxxx
SH	101001	xxxxxx	xxxxx
LH	100001	xxxxxx	xxxxx
BLTZ	000001	xxxxxx	00000
JAL	000011	xxxxxx	xxxxx

PLA_AND.txt는 위와 같이 구성하였다. r type은 개별적인 func값을 가지고 I type중 특별한 명령어들은 register immediate값을 추가적으로 가졌다.

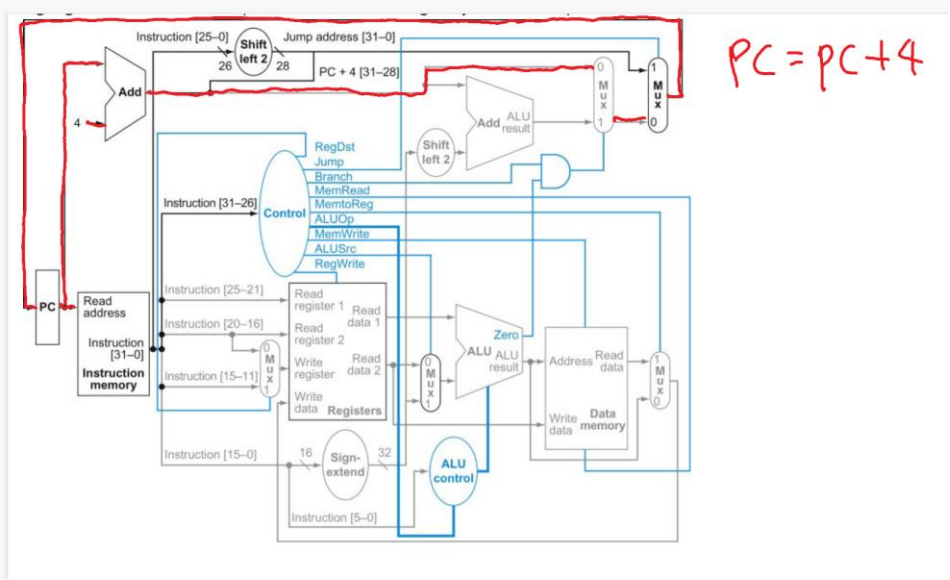
PLA_OR

Instruction	ADDU	OR	ADDIU	XORI	SLL	SRAV	SH	LH	BLTZ	JAL
RegDst	01	01	00	00	01	01	xx	00	xx	10
RegDatSel	00	00	00	00	00	00	xx	00	xx	11
RegWrite	1	1	1	1	1	1	0	1	0	1
EXTmode	x	x	1	0	x	x	1	1	1	x
ALUsrcB	00	00	01	01	00	00	01	01	10	xx
ALUctrl	0x	0x	0x	0x	00	01	0x	0x	0x	xx
ALUop	00101	00001	00101	00011	01101	01111	00100	00100	10000	xxxxx
DataWidth	xxx	xxx	xxx	xxx	xxx	xxx	010	010	xxx	xxx
Memwrite	0	0	0	0	0	0	1	0	0	0
MemtoReg	0	0	0	0	0	0	x	1	x	x
Branch	000	000	000	000	000	000	000	000	101	xxx
Jump	00	00	00	00	00	00	00	00	00	01

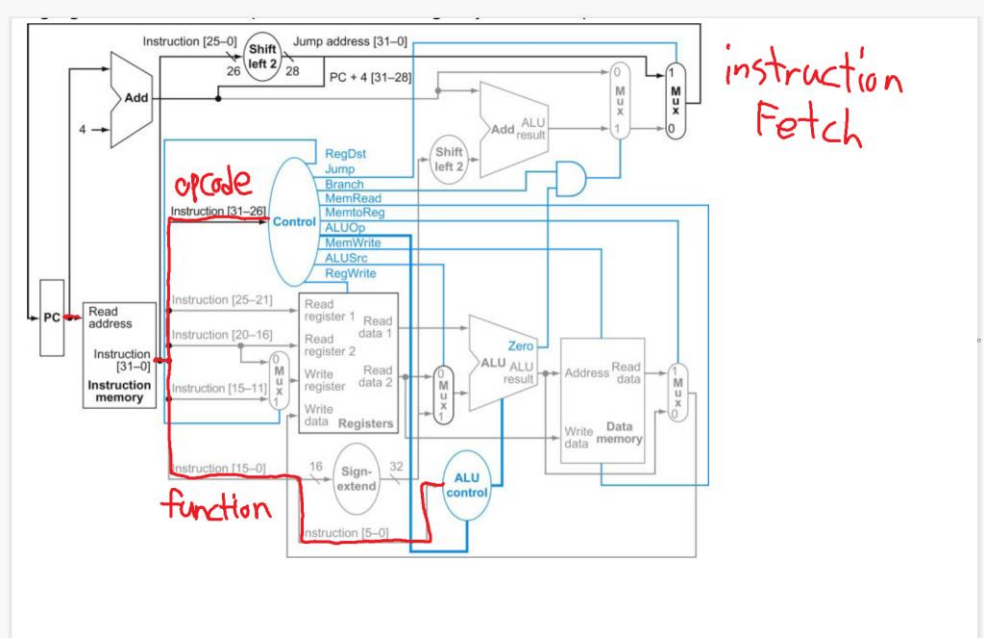
PLA_OR.txt는 위와 같이 구성하였다. 각 명령어마다 동작 시 지녀야 할 control signal을 가지고 있는 것을 알 수 있다.

각 신호에 대한 설명은 아래에서 명령어마다 동작 순서와 함께 설명한다.

우선 일반적인 PC = PC + 4의 모습을 설명한다.

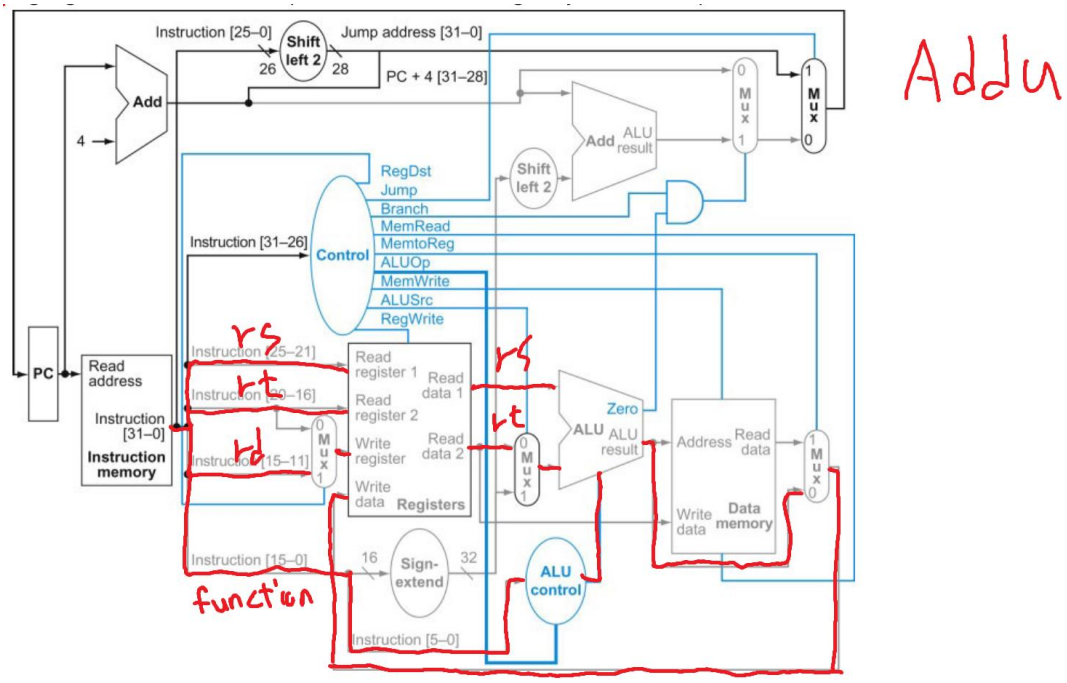


PC의 값을 증가시키는 것은 위의 사진과 같다. PC는 우선 Adder에서 상수 4와 더해진 후 mux를 통해 branch와 jump를 하지 않는 경우에 PC의 입력값으로 전해지게 된다. 이는 branch, jump control를 selection으로 하는 mux에 의해 값이 분기된다.



위는 명령어를 fetch하는 경우이다. opcode는 control로 들어가 명령이 해석되고 function은 opcode가 000000인 rtype명령어일 경우 ALU control에서 추가적인 동작을 결정한다.

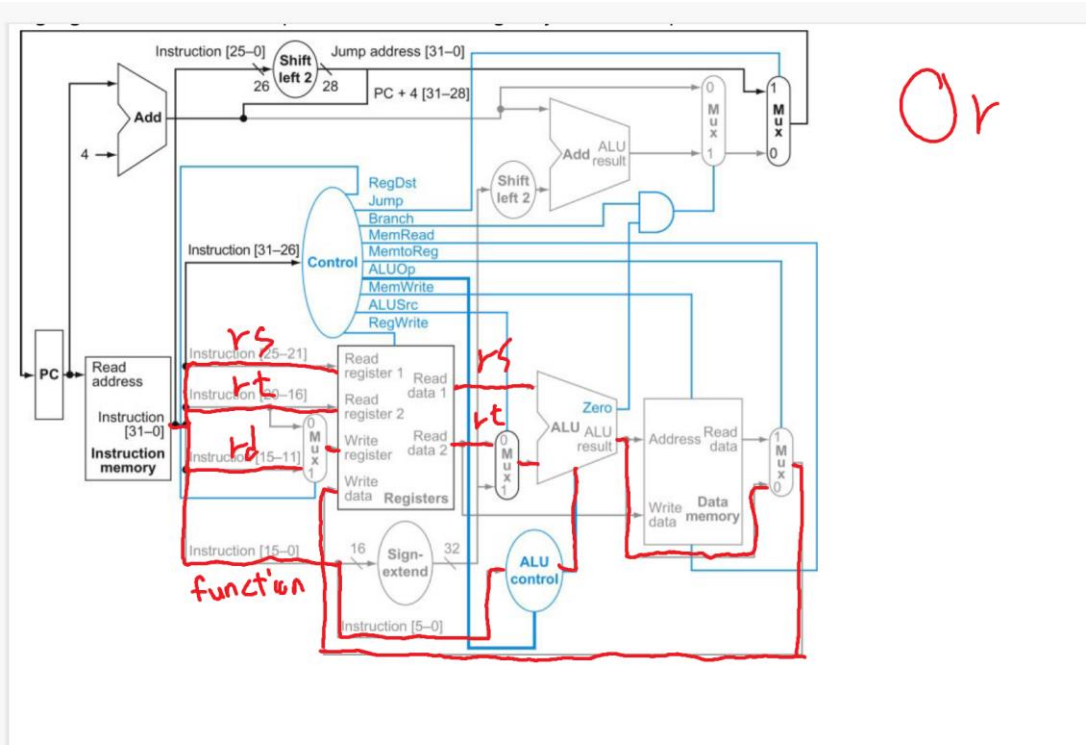
<ADDU>



ADDU의 경우이다. 우선 ALUop는 00101로 unsigned add를 하게된다. 위와 같이 rs와 rt는 그대로 ALU의 입력으로 전해지고 rd는 RegDst가 01이므로 Mux에서 선택된다. 또한 rtype이므로 추가적인 ALU의 동작을 function값을 ALU control에서 해석하여 동작시킨다. ALU의 입력으로 들어가는 값은 rs와 ALUSrcB가 00이기에 상수값이 아닌 rt가 선택되어 ALU의 입력으로 들어가게 된다. 그리고 ALUctrl값은 0x로 각각 register에서 read된 값이 그대로 들어가고 shift는 하지 않으므로 x라는 don't care값을 넣어주었다. 그 후 ALU에서 연산을 마친 값은 또 한번 MemtoReg가 0이기에 MUX에서 ALU의 결과값이 선택되어 register의 write data로 들어가게 된다. 이때 ALU의 결과를 레지스터에 write하게끔 RegDatSel값이 00이다. Register에 write를 하므로 RegWrite는 1이다.

추가적으로 EXTmode나 DataWidth는 해당 control signal들이 결정하는 요소가 이 명령어의 동작과는 상관이 없으므로 x(Don't care)이다. 그리고 Memwrite, Branch, jump control signal은 메모리에 잘못된 값을 쓰거나 명령어 순서에 분기가 나타나지 않게 하기위해 무조건 0으로 유지되어야 한다.

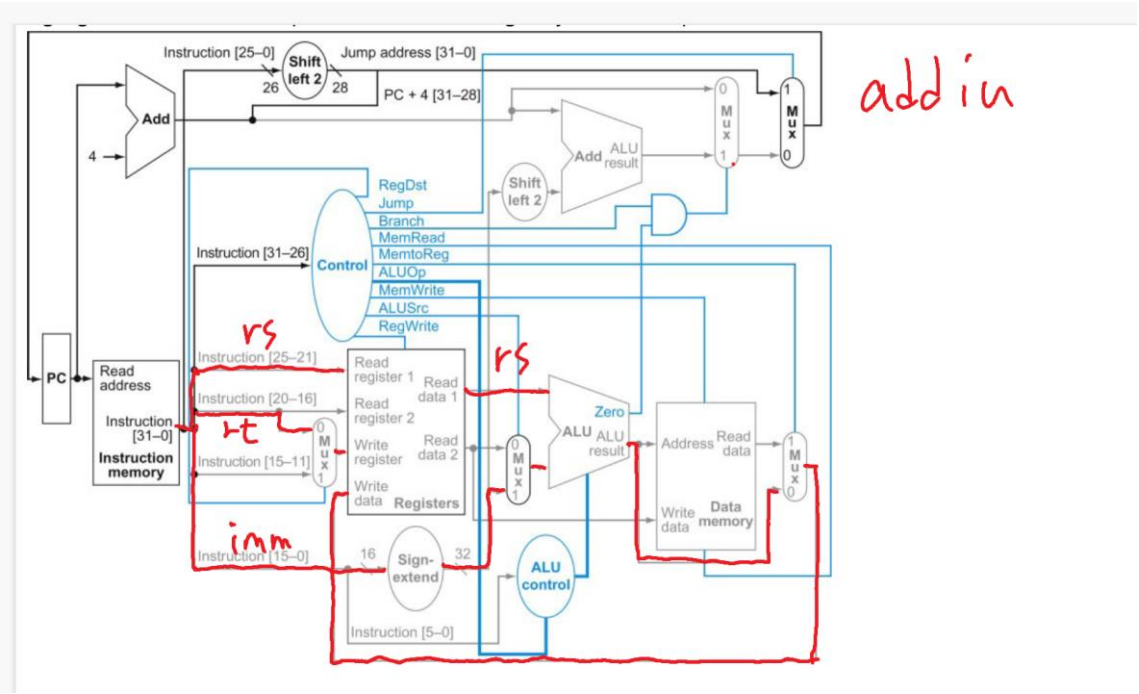
<OR>



다음은 OR의 경우이다.

OR도 마찬가지로 rtype이므로 ADDU와 마찬가지로 function을 사용한다. ADDU와 거의 유사하게 작동하며 모든 control신호 또한 비트연산을 하기 때문에 생기는 ALUop 신호를 제외한 나머지 신호들은 동일하다. 그 이유는 일반적으로 pc = pc + 4흐름이고 register에 값을 쓰고 memory에는 접근하거나 읽지 않기 때문이고 rd,rs,rt를 모두 사용하고 상수값의 사용이 없기 때문이다. 따라서 ALUop가 00001로 비트연산을 하는 작업을 제외한 실행 흐름은 동일하다.

<ADDIU>



위는 ADDIU이다.

ADDIU명령은 위의 명령들과 달리 rt에 값을 저장하기 때문에 RegDst신호는 00으로 rt를 선택하게끔 한다. RegDatSel신호는 00으로 ALU 결과값이 rt에 저장된다.

RegWrite신호는 register의 값을 쓰므로 1이다.

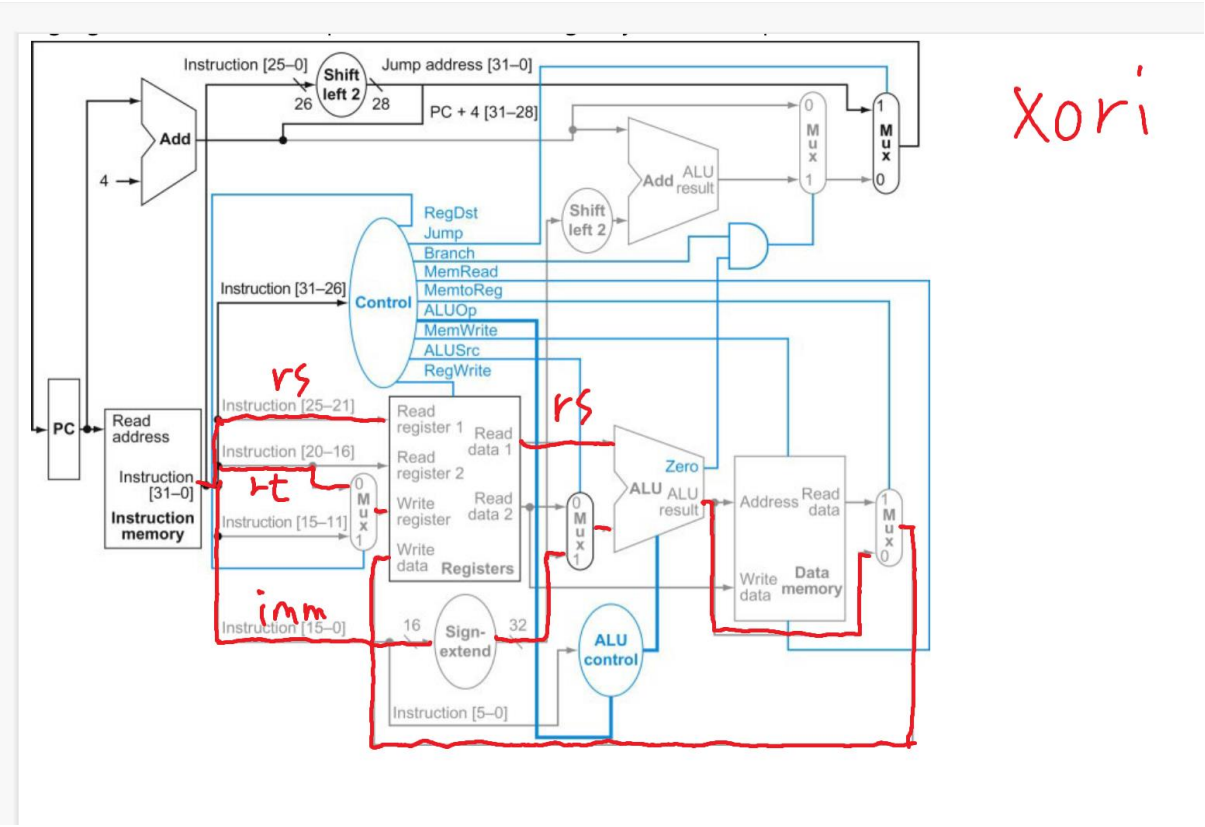
이번 명령에서는 EXTmode신호를 사용한다. 이는 16비트 상수를 비트확장하여 연산하기 위함인데 상수 값은 sign-extension을 사용하므로 1 신호를 주었다.

ALU의 입력값으로는 rs와 rt가 아닌 상수값이 들어가야하므로 ALUsrcB가 01로 이를 결정한다. ALUctrl신호는 값을 바꾸거나 shift하지 않기 때문에 0x로 쉬프트 관련 control부분은 don't care로 주었다. ALUop는 00101로 unsigned add연산을 진행한다.

ADDIU는 레지스터에 값을 쓰므로 DataWidth는 xxx로 don't care로 주었다. 역시나 Memwrite 또한 0으로 메모리에 값을 쓰지 않음을 알려준다.

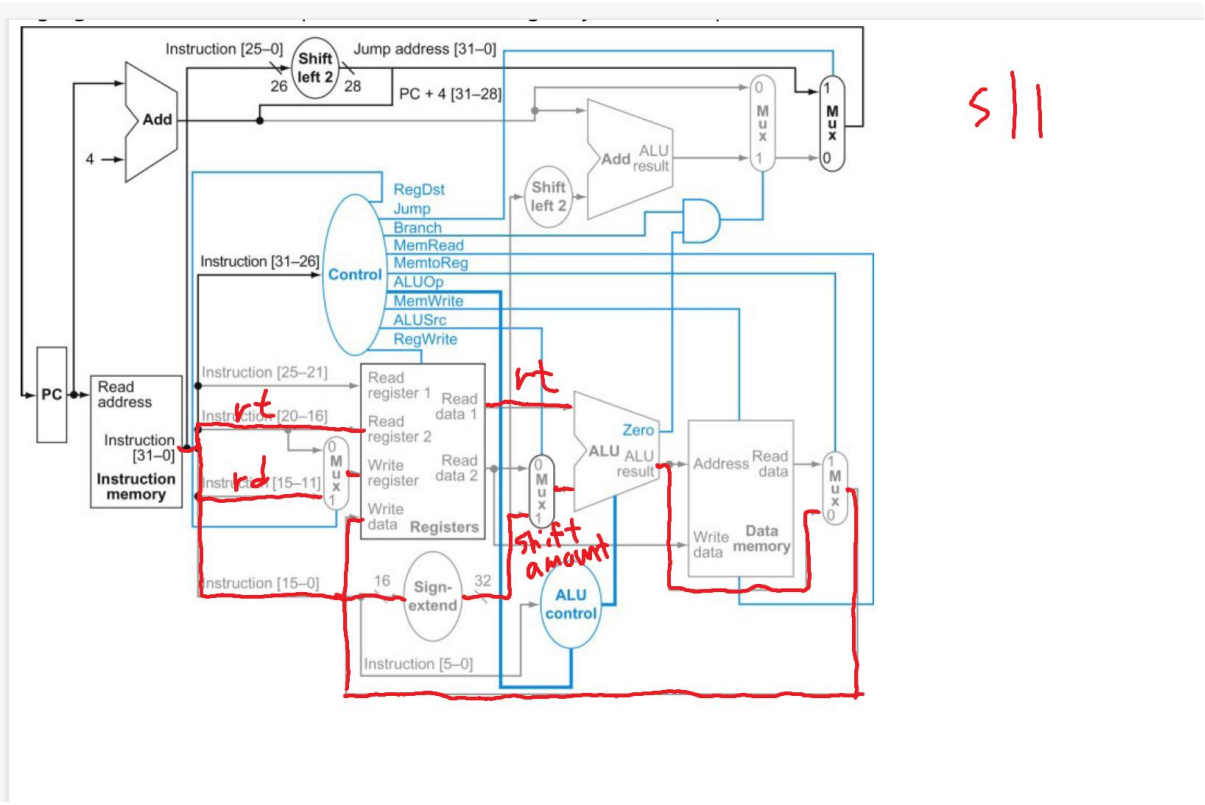
MemtoReg 신호는 ALU결과를 사용하므로 0으로 하였고 나머지 Branch 와 Jump 분기가 아니므로 0으로 하였다.

<XORI>



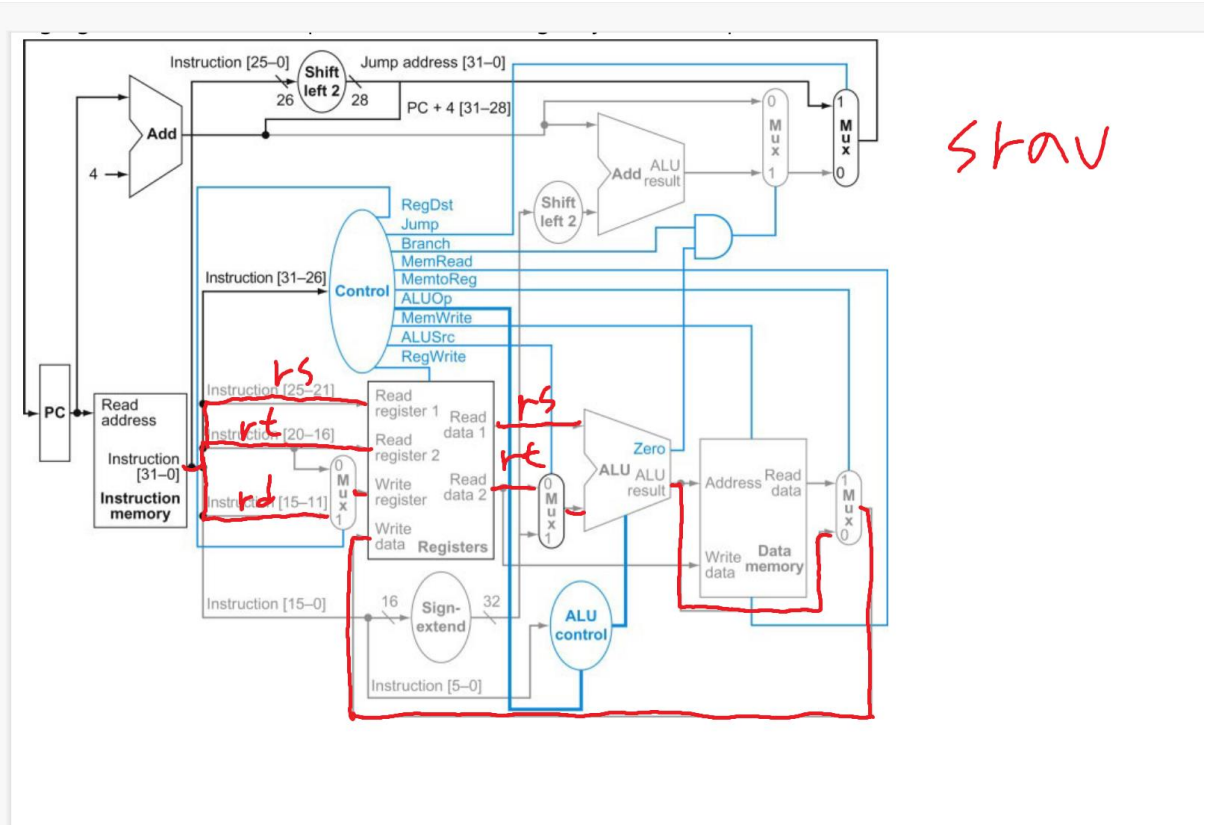
위는 XORI의 경우이다. XORI 또한 위의 ADDIU명령어와 ALUop신호를 제외한 나머지 control신호가 동일하다. 그 이유도 상수와 레지스터의 연산을 진행한다는 점, 연산이 끝나면 레지스터에 값을 쓴다는 점, 메모리를 사용하지 않는다는 것들이 동일하기 때문이다. 하지만 ALU에서 xor연산을 한다는 것과 상수비트확장에 있어서 ADDIU와 다르게 zero-extension을 한다는 점이 다르다.

<SLL>



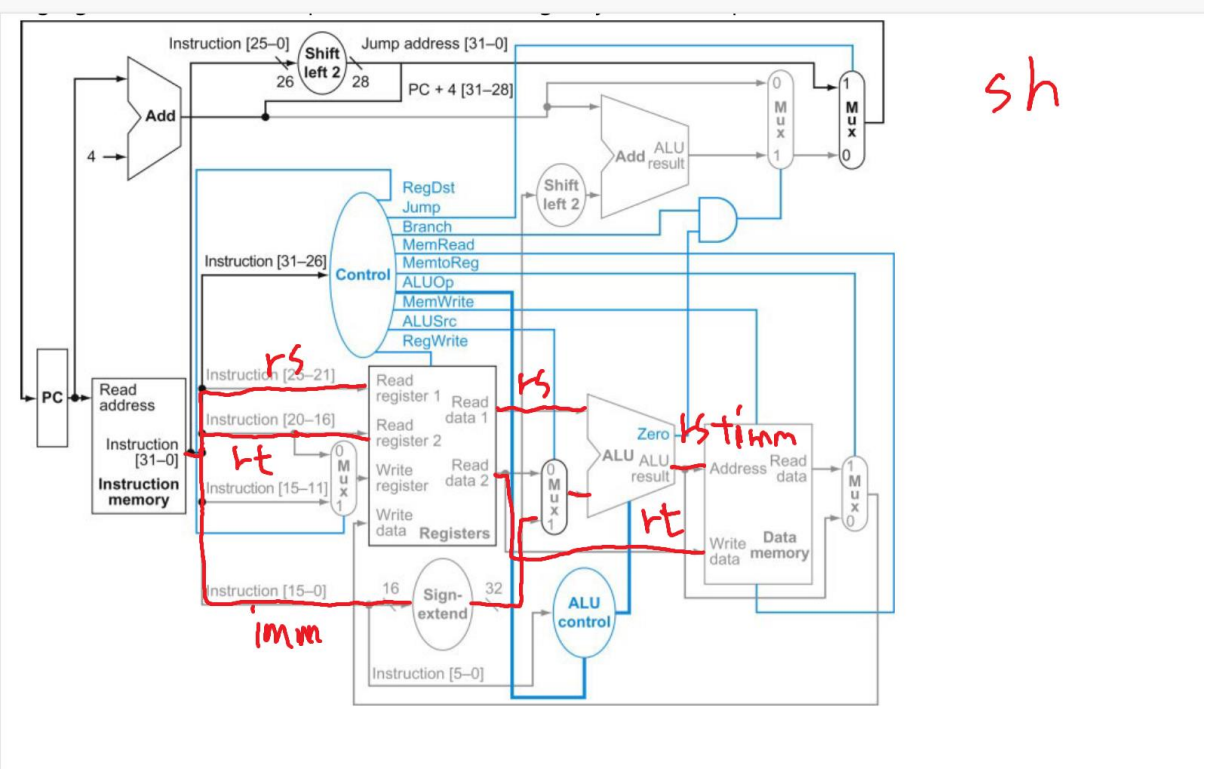
SLL명령어는 shift amount값만큼 레지스터를 logical Left Shift하는 명령어이다. 먼저 RegDst는 00으로 rd에 저장하게 되며 RegDatSel은 00으로 ALU의 값을 저장한다. 레지스터에 값을 써야하기 때문에 RegWrite는 1이며 immediate값이 아닌 5비트 shift amount값을 사용하기 때문에 비트확장에 관련한 EXTmode값은 don't care이다. ALUsrcB는 00으로 immediate값이 아닌 B port를 사용한다. shift amount값은 비트 기준으로 상수에 있지만 상수값을 사용하지 않고 B port를 사용하는 이유는 ALUctrl에 있다. ALUctrl에서 shift할 값을 어떤 것을 사용할지를 결정할 때 shift amount를 사용한다는 00이라는 control을 하게되면 정상적으로 작동된다. sll은 shift amount만 사용해야 하는데 뒷 function값과 함께 ALU에 들어가면 제대로 된 shift를 할 수 없기에 ALUctrl을 사용해서 제어한다. ALUop값은 01101로 logical left shift를 하게된다. 메모리에 접근하지 않기에 DataWidth는 xxx로 don't care이고 Memwrite는 메모리에 잘못된 값이 들어가지 않도록 0, MemtoReg는 ALU값을 레지스터로 보내야 하므로 0 branch와 jump를 하지 않기에 나머지 두 신호도 0이 된다.

<SRAV>



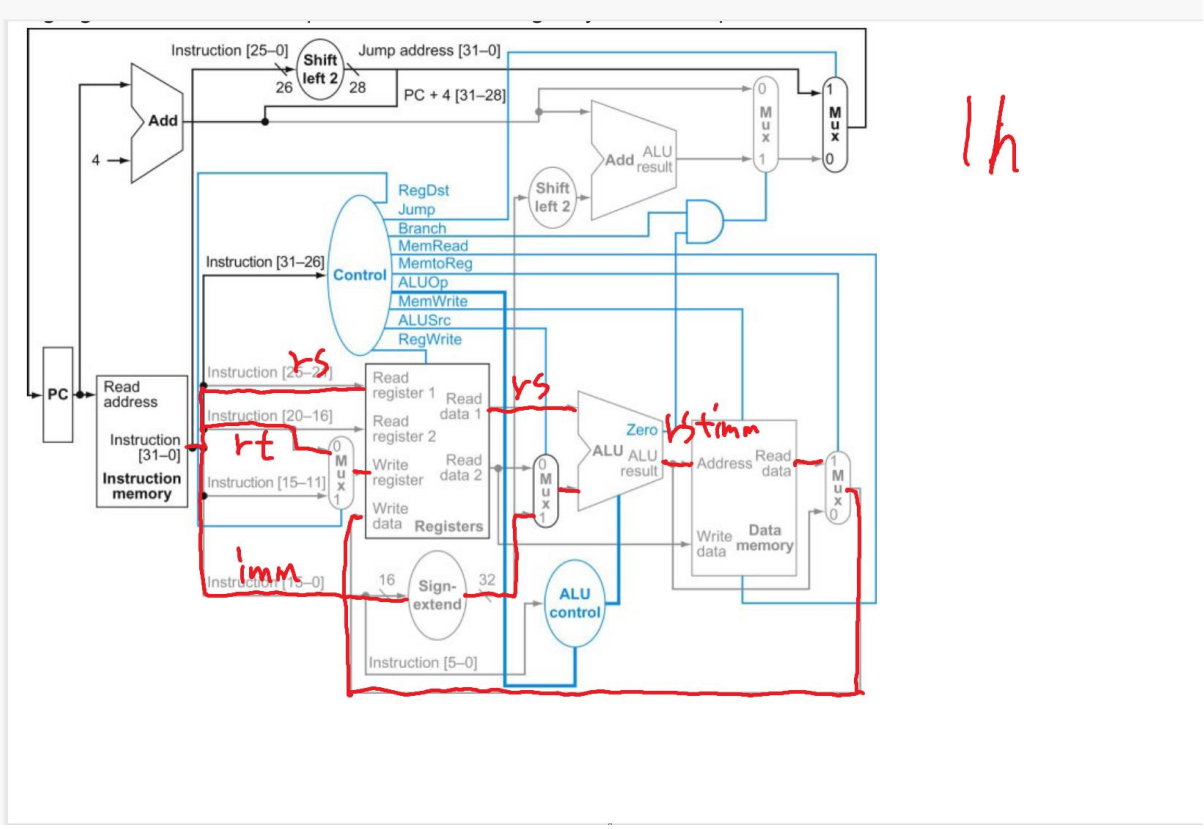
srav명령어는 sll명령어와 다르게 shift 값으로 shift amount값이 아닌 rs를 사용한다. 따라서 sll명령어와의 차이는 ALUctrl로 shift amount값이 아닌 rs를 사용하도록 01을 주고 ALUop를 01111로 arithmetic right shift를 하게끔 하면 그외의 나머지 control신호들을 동일하게 하여도 정상 작동한다.

<SH>



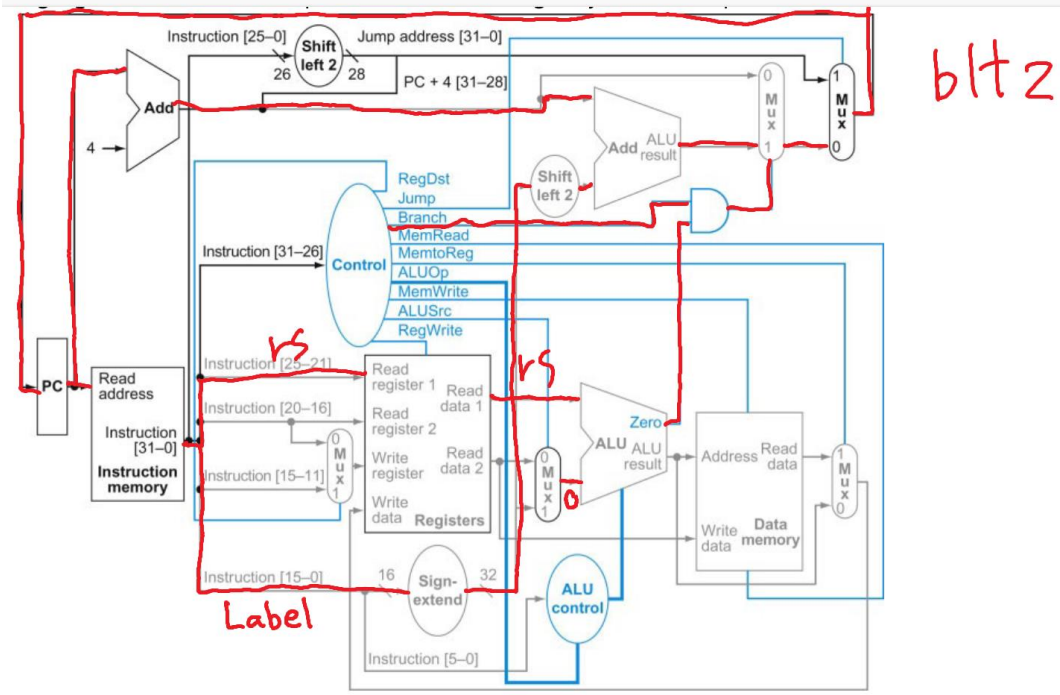
SH명령어는 store halfword이므로 메모리에 접근하는 명령어이다. 따라서 레지스터에 값을 쓰지 않기 때문에 RegDst,RegDatSel은 모두 don't care이고 Regwrite는 무조건 0이어야 한다. EXTmode는 1로 메모리 주소를 만들어내기 위해 immediate 값과 연산을 할 것이므로 sign-extension을 하도록 한다. ALUsrcB는 rs에 더해질 immediate값을 선택하기 위해 01로 하고 ALUctrl은 shift에 관한 동작이 없기에 0x로 두었다. ALUop는 더하기 연산을 위해 00100으로 하고 이번엔 메모리에 값을 써 넣기 때문에 DataWidth 신호를 사용한다. 이때 SH명령어는 halfword를 주소에 저장하기 때문에 010으로 하였고 메모리에 값을 쓰므로 Memwrite는 1이다. 레지스터에 값을 쓰지 않기 때문에 MemtoReg는 don't care이다. 마찬가지로 분기가 일어나지 않기에 Branch, jump또한 0이다.

<LH>



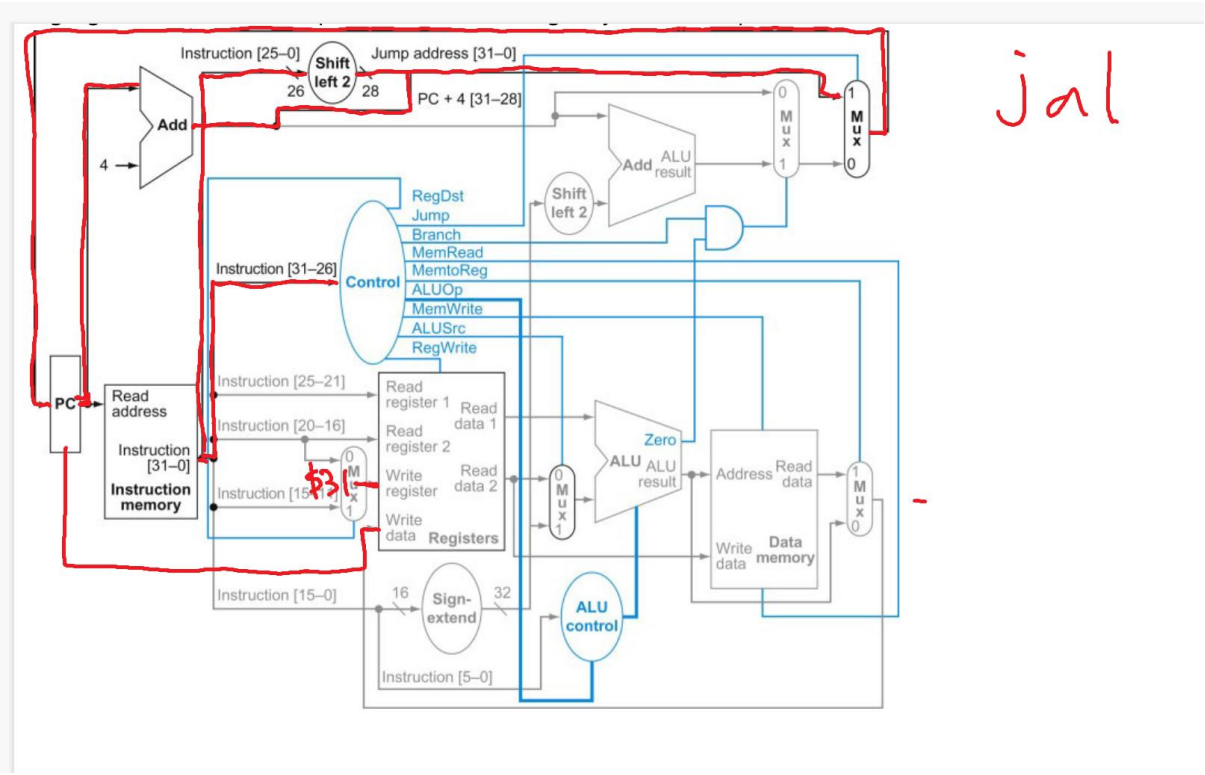
LH명령어는 SH와 반대로 메모리에서 값을 읽어와 레지스터에 로드하는 명령어이다. 그렇기에 레지스터에 관한 control들이 동작한다. 먼저 rt에 rs와 immediate값을 더한 주소에 해당하는 값을 저장하므로 RegDst는 00이 된다. 또한 Reg에 read된 MEM값을 저장하므로 RegDatSel은 00이 된다. Regwrite는 레지스터에 값을 쓰므로 1이되며 SH명령어와 마찬가지로 rs와 immediate값을 더해야하므로 immediate의 비트확장은 sign-extension이다. 따라서 EXTmode는 1이다. ALUsrcB는 immediate값을 선택해야 하므로 01이고 ALUctrl은 shift와 같은 추가적인 ALU동작을 하지 않기에 0x로 두었다. SH와 마찬가지로 더하기 연산을 통해 주소값을 얻어내기 때문에 ALUop는 동일한 00100이고 halfword를 load하기 때문에 DataWidth또한 010이다. Memwrite는 메모리를 읽기만 하고 쓰진 않으므로 0이며 MemtoReg는 ALU가 아닌 Mem값을 Reg의 writedata로 사용할 예정이므로 1이다. Branch와 jump는 실행분기가 일어나지 않으므로 0이다.

<BLTZ>



BTLZ명령어는 rs와 0을 비교하여 branch할지 말지를 결정한다. 따라서 register나 memory에 접근하지 않는다. 따라서 RegDst,RegDatSel,DataWidth,MemtoReg는 don't care이고 Regwrite와 Memwrite는 0으로 잘못된 값이 쓰여지지 않게끔 해야한다. ALU에서는 rs와 0이 서로 Set less than 연산을 진행해야하기 때문에 ALUsrcB는 zero를 선택하게끔 10으로 두었다. ALUctrl은 쉬프트 관련 추가 연산이 없으므로 0x로 두었고 ALUop는 set less than에 해당하는 10000값을 주었다. Branch는 101로 branch if not zero로 하였는데 이 이유는 rs<0이 만족되면 ALU의 결과로 1을 반환하게 될 것이다. 그러면 zero가 아니므로 branch하게 된다. jump는 하지 않으므로 00이다. 추가적으로 Label은 위의 ALU에서 주소의 맨 오른쪽 두 비트는 00이 되게끔 shift left 2를 한 후 pc = pc+4+Label연산을 진행한다. 따라서 EXTmode도 1로 sign-extension을 진행한다.

<JAL>



JAL명령어는 jump and link로 무조건 jump를 진행하지만 pc+4의 값을 $31에 저장해 두고 나중에 다시 돌아올 수 있게끔 한다. 우선 RegDst는 $31에 PC값을 저장해야하므로 10, RegDatSel은 PC값을 Register file에 쓰게끔 11, Regwrite는 1로 설정했다.

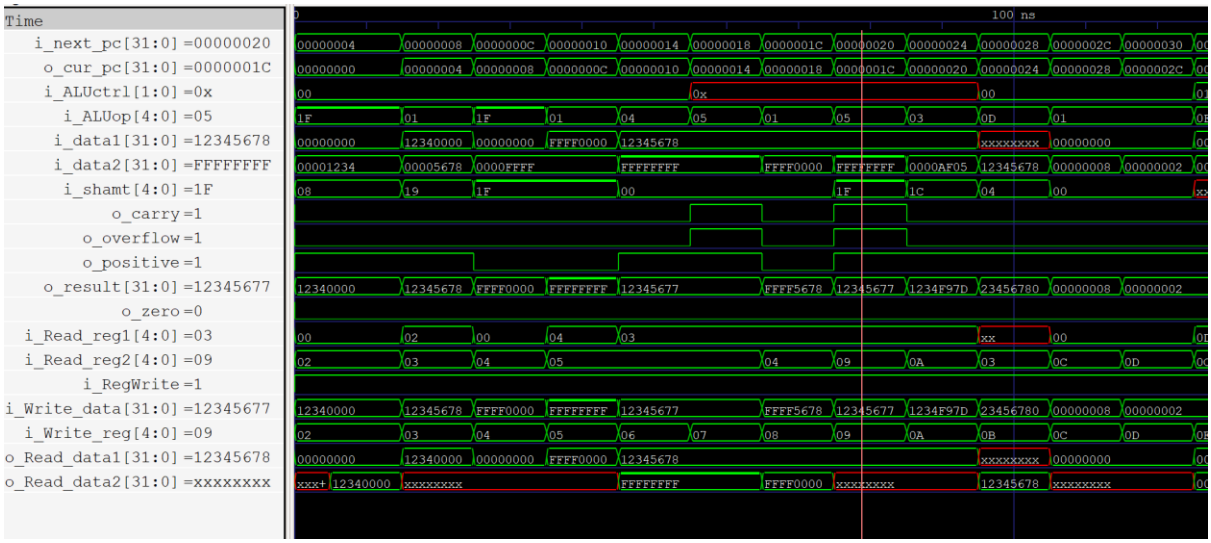
상수는 사용되지 않으므로 비트확장에 관련된 EXTmode는 x이고 ALU관련 모든 신호들은 x이다. Memwrite는 0으로 잘못된값이 쓰여지는 것을 방지하고 Branch는 하지 않으므로 xxx, jump는 Pseudo Jump address format을 사용하도록 01로 두었다.

따라서 PC = PC+4+(shift left 2)Label이 된다.

```
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 = 0x12355677 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_0000_0000_0000_1000//ori $0 $12 0x0008    $12 = 0x00000008 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 = 0x00000002 address:0x00000030
101001_01101_00011_0000_0000_0000_0010//sh $13 $3 0x0002     MEM[$13 + 2] = $3 address:0x00000034
100001_01101_01111_0000_0000_0000_0010//lh $13 $15 0x0002    $15 = MEM[$13 + 2] address:0x00000038
000001_00011_00000_0000_0000_0000_0011//bltz $3 0x0003                          address:0x0000003C
000001_00100_00000_0000_0000_0000_0011//bltz $4 0x0003                          address:0x00000040
000011_0000_0000_0000_0000_0000_0000_00 //jal                                   address:0x00000044
001111_00000_00010_0001_0010_0011_0100//lui $2 0x1234        $2 = 0x12340000    address:0x00000048
001101_00010_00011_0101_0110_0111_1000//ori $2 $3 0x5678     $3 = 0x12345678    address:0x0000004C
001111_00000_00100_1111_1111_1111_1111//lui $4 0x8765        $4 = 0xffff0000    address:0x00000050
001101_00100_00101_1111_1111_1111_1111//ori $4 $5 0x4321     $5 = 0xffffffff    address:0x00000054
000000_00011_00101_00110_00000_100000//add $3 $5 $6          $6 = 0x12345677    address:0x00000058
000000_00011_00101_00111_00000_100001//addu $3 $5 $7         $7 = 0x12345677    address:0x0000005C
000011_0000_0000_0000_0000_0000_0000_00 //jal                                   address:0x00000060
```

위는 사용한 instruction memory이다. 모든 명령어가 작동되는지 확인할 수 있게 하였고 분기를 나눌 때 조건에 맞게 작동되는지 unsigned명령어는 아닌 명령어와 무엇이 다른지 확인할 수 있게 하였다.



위는 명령어의 결과이다.

0x00000000일때는 lui명령어로 16비트 상수를 $2의 상위비트 16비트에 넣는 명령인데 ALU의 i_data2에서 0x00001234였던 상수가 ALU의 o_result에서 0x12340000으로 변경되어 register의 i_write_data에서 $2에 0x12340000으로 쓰여지는 것을 알 수 있다.

0x00000004일때는 ori명령어로 상수를 레지스터값과 or연산을 진행한다. 따라서 register의 o_read_data1에 0x12340000으로 $2값이 읽혔고 이는 ALU의 i_data1으로 들어가고 i_data2는 상수값 0x00005678이 들어간다. 두 입력은 ALU에서 or연산을 진행하고 결과인 o_result가 register에 i_Write_data로 전달되어 $3에 쓰여진다.

0x00000008와 **0x0000000C**는 동일한 방법으로 $4 $5에 0xffff0000과 0xffffffff가 저장된다.

0x00000010에는 $3 $5이 서로 add연산을 한 후 $6에 저장되는데 이는 add연산과 addu연산의 비교를 위한 명령어이다. 바로 다음 명령어 **0x00000014**는 $3과 $5를 addu연산하여 $7에 저장되는데 둘은 같은 흐름으로 ALUop만 다를뿐 동일한 흐름으로 레지스터에 값을 쓰는데 $6과 $7은 결과적으로 같은 값을 저장하게 된다. 하지만 add연산은 0xffffffff를 부호가 있는 수로 판단하여 연산을 하여 carry나 overflow가 발생하지 않았지만 addu연산은 부호가 없는 수로 판단하여 carry와 overflow의 flag가 1이된 것을 확인할 수 있다. 전자는 0x12345678과 -1를 더했다면 후자는 0x12345678과 0xffffffff를 더했다는 것을 알 수 있다.

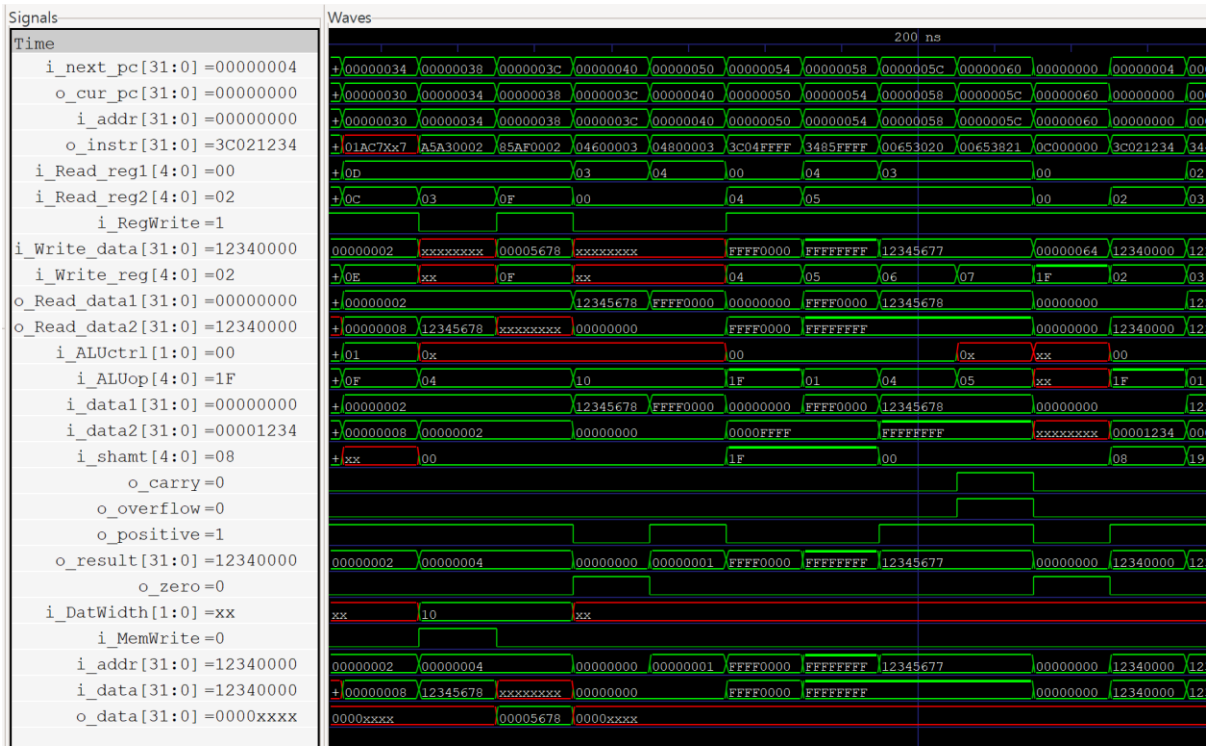
0x00000018에는 $3과 $4가 서로 or연산을 진행한다. register의 o_read_data1에 0x12345678, o_read_data2에 0xffff0000이 or연산을 진행하고 ALU의 o_result에 0xffff5678이 나온 후 레지스터의 i_Write_data에 전달되고 i_Write_reg에 08 즉 $8에 저장됨을 알 수 있다.

0x0000001C와 **0x00000020**은 각각 addiu,xori이다. addiu는 위의 **0x00000014**에 해당하는 addu연산과 동일한 연산을 진행하되 연산에 상수값을 사용한다.

두 연산 모두 ALUop만 다르게 연산을 진행하지만 ALU에서 연산 후 Register에 값을 쓴다는 것은 동일하다. 그 결과 $9에는 addu의 연산 결과인 0x12345677이 저장되고 $10에는 0x1234f97d가 저장된다.

0x00000024는 sll명령어이다. $10에 $3을 amount=4만큼 shift logical left한 결과가 저장되는데 결과적으로 $11에 0x23456780이 저장된 것을 알 수 있다.

0x00000028과 0x0000002C는 아래 명령어에서 사용할 적당한 상수값을 만들어서 레지스터에 저장하는 과정이다.



0x00000030은 srav명령으로 $14에 $12를 $13의 값만큼 shift right arithmetic하는 것이다. 그 결과 $12의 0x00000008이 2번 arithmetic right shift되어 0x00000002가 되어 $14에 저장되었다.

0x00000034는 sh명령으로 $13에 0x00000002라는 상수를 더해 메모리에 접근할 주소를 만들고 이곳에 $3의 값을 store하는데 register의 o_Read_data1에 $13, o_Read_data2에 $3값이 나오고 $13은 상수와 ALU에서 add연산을 마쳐 o_result 0x00000004가 되었고 이것이 memory의 i_addr에 전달되어 메모리 주소를 가리키고 있으면 $3의 값이 i_data로 전달되어 메모리의 값이 적힌다.

그 결과는 다음과 같다.

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
xxxxxxxxxxxxxxxx0101011001111000 xxxx5678
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
```

4라는 주소에 32비트가 저장되나 하위 16비트만 저장되어 저 자리에 $3의 하위 16비트만 저장된 것을 확인할 수 있다.

0x00000038은 lh명령으로 이번엔 메모리에서 값을 가져와 레지스터에 저장한다. 이번엔 $13과 상수 0x00000002를 더한 곳에서 전 명령어에서 저장한 값을 불러왔다.

0x00000004의 i_addr에 접근하니 0x00005678이 메모리에서 read되었고 이를 register

의 i_Write_data에 전달하여 $15에 저장하는 것을 볼 수 있다.

0x0000003C와 0x00000040은 bltz명령어를 실행하며 각각 양수와 음수의 값을 조건에 넣어주었을 때 어떻게 동작하는지를 살펴본다.

우선 **0x0000003C**에서 0x12345678이란 값은 0보다 크므로 branch의 조건에 맞지않아 pc = pc + 4으로 **0x00000040**으로 넘어가서 실행하게 되고 **0x00000040**에서 0xffff0000이라는 값은 음수이므로 조건을 만족하게 되어 16만큼 건너뛴 **0x00000050**을 실행한다. 그 이유는 상수 값으로 0x0003을 주었는데 이는 left shift 2를 하여 0x000C가 되고 이것이 PC + 4와 더해져 결과적으로 PC = PC + 0x00000010이 실행되게 되는 것이다.

마지막으로 **0x00000060**이다. 0x00000040과 0x00000060사이의 명령어는 bltz명령어가 잘 실행되는지 확인하기 위해 넣어둔 중복된 명령들이므로 설명을 생략한다.

0x00000060은 jal명령어로 PC+4를 $31에 적어둔다. i_Write_Reg부분을 확인하면 1F로 $31에 i_Write_Data **0x00000064**를 적는다. 그 후 instruction[25:0]을 left shift하게 되면 현재 label이 0이므로 28비트가 모두 0인 값이 만들어지고 이값의 앞에 PC + 4의 4비트를 이어붙인다. **0x00000064**의 앞 4비트또한 0이므로 결과적으로 PC는 **0x00000000**으로 jump하게된다.

```
00000000 00000000 00000000 00000000 : 00000000
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
00010010 00110100 00000000 00000000 : 12340000
00010010 00110100 01010110 01111000 : 12345678
11111111 11111111 00000000 00000000 : ffff0000
11111111 11111111 11111111 11111111 : ffffffff
00010010 00110100 01010110 01110111 : 12345677
00010010 00110100 01010110 01110111 : 12345677
11111111 11111111 01010110 01111000 : ffff5678
00010010 00110100 01010110 01110111 : 12345677
00010010 00110100 11111001 01111101 : 1234f97d
00100011 01000101 01100111 10000000 : 23456780
00000000 00000000 00000000 00001000 : 00000008
00000000 00000000 00000000 00000010 : 00000002
00000000 00000000 00000000 00000010 : 00000002
00000000 00000000 01010110 01111000 : 00005678
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
00000000 00000000 00000000 01100100 : 00000064
```

위에서부터 $0 ~ $31이다. 각 명령어에 맞게 레지스터에 값이 저장되었음을 알 수 있다.

<Conclusion>

이번에 구현된 single cycle CPU를 Programmable Logic Array를 이용하여 구현해보았는데 단순히 텍스트파일에 값을 써넣어서 작동시키는 방식이 신기했다. sum of products는 모든 논리를 나타낼 수 있기 때문에 이러한 방식이 가능하다는 것을 알았다. 각 명령어마다 어떤 흐름으로 동작하는지 선으로 따라가보고 이러한 동작을 하기 위해서는 어떤 신호를 사용하면 좋을지 고민해보는 시간이 이번 CPU를 이해하는데 도움이 되었다. 특히나 어려웠던 것은 bltz명령어였는데 우선 ALU에서 set less than을 사용하는데 조건을 만족하면 결과는 1이되므로 zero flag는 0이 되는데 둘중 어떤 값을 branch조건에 사용해야하는지가 헷갈렸고 또한 read register 1은 read data1으로 read register 2은 read data2로만 이동하는 줄 알았는데 gtkwave를 통해 꼭 그렇지만은 않은 것을 알게되었다. 과제를 진행하면서 다양한 명령어들의 동작방식과 명령어마다 다른 instruction format을 직접 쳐보고 이해할 수 있었다.