컴퓨터구조실험

과제 이름: Project #1

담당교수: 이성원

학 과: 컴퓨터정보공학부

학 번: 2019202021

이 름: 정 성 엽

제출일: 2023/4/19

1. Introduction

A. 과제 소개

MIPS 명령어 중에서 LW, SW, ORI, ADD, SUB, J, LUI, BREAK, LLO, LHI는 구현되어 있다. 이때 ADDU, OR, ADDIU, XORI, SLL, SRAV, SH, LH, BLTZ, JAL 명령어를 AND-plane에서는 OPCODE, FUNCTION, REGIMM을 채워 디코딩 배열을 완성하고, OR-plane에서는 RegDst, RegDatSel, RegWrite, SEUmode, ALUsrcB, ALUctrl, ALUop, DataWidth, MemWrite, MemtoReg, Branch, Jump에 대한 제어 신호 배열을 완성한다. 완성 후에는 bat 파일을 실행시켜 gtkwave를 실행하고 파형을 확인한다.

B. 배경 지식

- **R-type**

op	rs	rt	rd	shamt	funct
6-bit [31:26]	5-bit [25:21]	5-bit [20:16]	5-bit [15-11]	5-bit [10:6]	6-bit [5:0]

- **I-type(like Branch)**

op	rs	rt	immediate / offset
6-bit [31:26]	5-bit [25:21]	5-bit [20:16]	16-bit [15:0]

- **J-type**

op	instr index
6-bit [31:26]	26-bit [25:0]

명령은 각 특성에 따라 type을 가지고 있으며 M_TEXT_SEG.txt를 활용하여 gtkwave에서 파형을 얻는다면 위의 형식에 맞춰 명령을 구성해야한다.

- **Instruction decoding configuration 설명**

1. 6-bit Op
   각 명령에 대한 Opcode를 나타낸다. R-type은 000000 Reglm은 000001 이며 다른 명령어도 각 Opcode를 가지고 있다.
2. 6-bit Func
   R-type 명령인 경우 Function에서 재분류 되며 예를 들어 sll 명령의 function은 000000이다.
3. 5-bit Regimm
   Reglmm(I-type)인 경우 해당되며 이곳에서 재분류 되어 명령을 실행한다.

- **Control Signal Configuration 설명**

1. 2-bit RegDst
   ALU 계산 시 B값에 들어갈 레지스터를 Rt, Rd, $31(PC저장) 중 선택한다.

2. 2-bit RegDatSel
   ALU의 결과값,LO, HI,PC 값 중 어느 값을 RF에 write 할지 결정한다.
3. 1-bit RegWrite
   RF에 값을 쓸지 결정하는 제어 신호이다.
4. 1-bit SEUmode
   imm의 값을 sign extention 할지 non-sign extention 할지 결정한다.
5. 2-bit ALUsrcB
   ALU 계산 시 B값에 레지스터값, imm, zero 값을 넣을 지 결정한다.
6. 2-bit ALUctrl
   ALU 를 제어하는 신호로 shift와 input에 대한 결정을 한다.
7. 5-bit ALUop
   ALU 연산 종류를 결정한다.
8. 3-bit DataWidth
   메모리 접근 시 데이터의 width를 32bit, 16bit, 8bit 중 결정하며 sign extend도 가능하다.
9. 1-bit Memwrite
   메모리에 값을 쓸지 결정하는 제어 신호이다.
10. 1-bit MemtoReg
    메모리의 값을 RF로 쓸지, ALU 연산 결과를 RF로 쓸지 결정하는 제어 신호이다.
11. 3-bit Branch
    ALU 연산에 따라 branch 할 종류를 결정한다.
12. 2-bit Jump
    imm26의 값으로 점프할지 Rs의 값으로 점프할지, 또는 점프하지 않을 지 결정한다.

C. My AND-plane(PLA_AND)

inst type	inst	Op	Func	Regimm
R	ADDU	000000	100001	xxxxx
R	OR	000000	100101	xxxxx
I	ADDIU	001001	xxxxxx	xxxxx
I	XORI	001110	xxxxxx	xxxxx
R	SLL	000000	000000	xxxxx
R	SRAV	000000	000111	xxxxx
I	SH	101001	xxxxxx	xxxxx
I	LH	100001	xxxxxx	xxxxx
branch like I	BLTZ	000001	xxxxxx	00000
J	JAL	000011	xxxxxx	xxxxx

이는 MIPS INST Datasheet을 참고하여 제작하였다.

D. My OR-plane(PLA_OR)

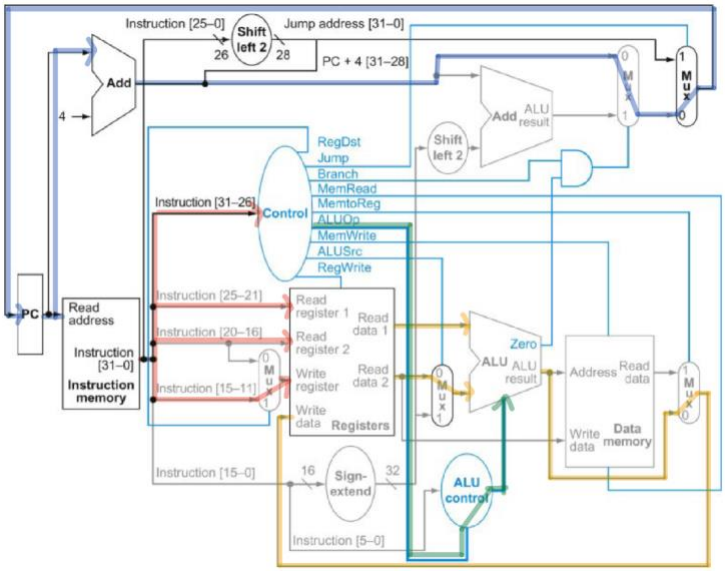
inst type	inst	RegDst 2-bit	RegDatSel 2-bit	RegWrite 1-bit	SEUmode 1-bit	ALUsrcB 2-bit	ALUctrl 2-bit	ALUop 5-bit	DataWidth 3-bit	Memwrite 1-bit	MemtoReg 1-bit	Branch	Jump
R	ADDU	01	00	1	x	00	00	00101	xxx	0	0	000	00
R	OR	01	00	1	x	00	00	00001	xxx	0	0	000	00
I	ADDIU	00	00	1	1	01	00	00101	xxx	0	0	000	00
I	XORI	00	00	1	0	01	00	00011	xxx	0	0	000	00
R	SLL	01	00	1	x	00	00	01101	xxx	0	0	000	00
R	SRAV	01	00	1	x	00	01	01111	xxx	0	0	000	00
I	SH	xx	xx	0	x	01	00	00101	010	1	0	000	00
I	LH	00	00	1	1	01	00	00101	110	0	1	000	00
branch like I	BLTZ	xx	xx	0	0	10	00	10000	xxx	0	0	111	00
J	JAL	10	11	1	1	xx	xx	xxxxx	xxx	0	0	000	01

이는 Control Signals for SCPU를 참고하여 제작하였고 이에 대한 이유는 아래 결과에서 같이 설명한다.

2. 결과화면

A. ADDU – addu $d $s $t – R-type

$d = $s + $t



ADDU는 unsigned number를 더하는 R-type 명령어로 위 그림의 datapath를 통해 진행된다. 레지스터 $s, $t에 저장된 32-bit 불러와 add 연산을 하되 unsigned number로 취급하여 $d에 저장한다. 파란 Path는 PC 값을 4 증가시키는 경로이고 빨간 Path는 $s와 $t에 저장된 값을 불러온다. 불러온 값은 ALUop에 따라 노란색 Path와 같은 경로로 연산을 하고 이를 $d 레지스터에 저장한다. 녹색 path는 ALUop를 ALU에 전달한다.

그러므로 PLA_OR은 01_00_1_x_00_00_00101_xxx_0_0_000_00_xxxxx 이다.

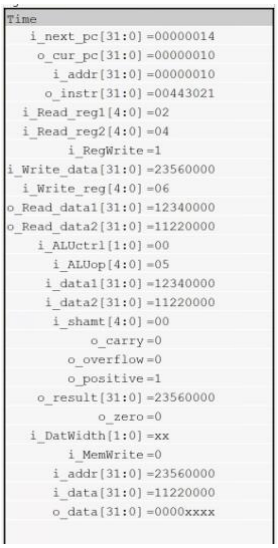
- Test_bench

  00111100_00000010_00010010_00110100 //lui $2 0x1234

  00111100_00000100_00010001_00100010 //lui $4 0x1122

  000000_00010_00100_00110_00000_100001 //addu $6 $2 $4

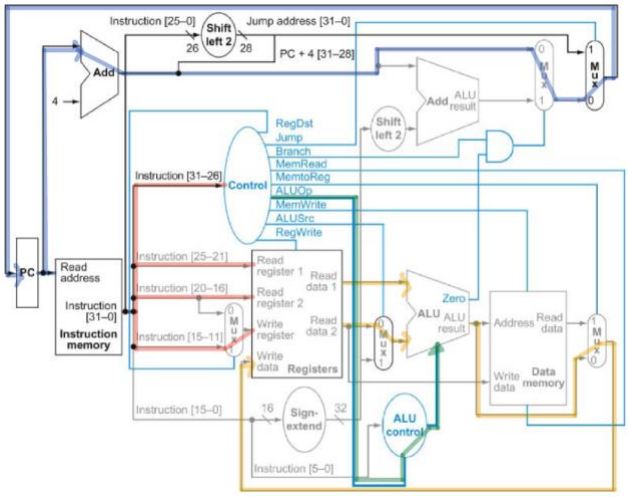
  $2의 값은 0x12340000 $4의 값은 0x11220000이다. 이를 unsigned add를 진행하면 0x235600000 값이다. 이는 아래 파형을 통해 확인할 수 있다.



ALUop에 원하는 명령값이 들어갔고 input도 올바르게 들어감을 확인할 수 있다.

B. OR -or $d $s $t - R-type

$d = $s | $t



OR 명령어는 R-type 명령어로 $s와 $t를 bitwise OR 연산하여 $d에 저장한다. 파란 path는 PC+4의 경로이고, 빨간 path는 Control unit의 출력과 RF의 Readdata와 Writedata를 관리하도록 한다. 노란 path는 ALU에서 연산하여 결과를 RF에 저장한다. 녹색 path는 ALUop를 ALU에 전달한다.

그러므로 PLA_OR은 01_00_1_x_00_00_00001_xxx_0_0_000_00_xxxxx 이다.

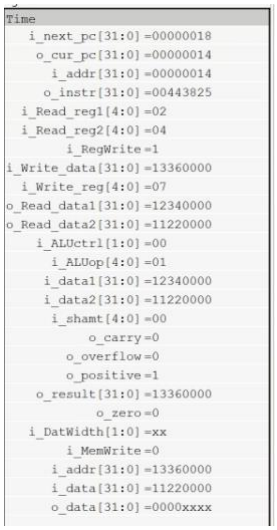
- Test_bench

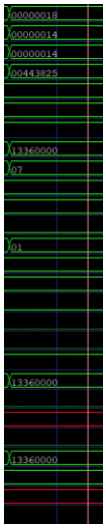
  00111100_00000010_00010010_00110100 //lui $2 0x1234

  00111100_00000100_00010001_00100010 //lui $4 0x1122

  000000_00010_00100_00111_00000_100101 //or $7 $2 $4

  $2 값은 0x12340000 $4 값은 0x11220000 일 때 or 연산을 진행하면 0x13360000이다. 이는 아래 파형을 통해 확인할 수 있다.

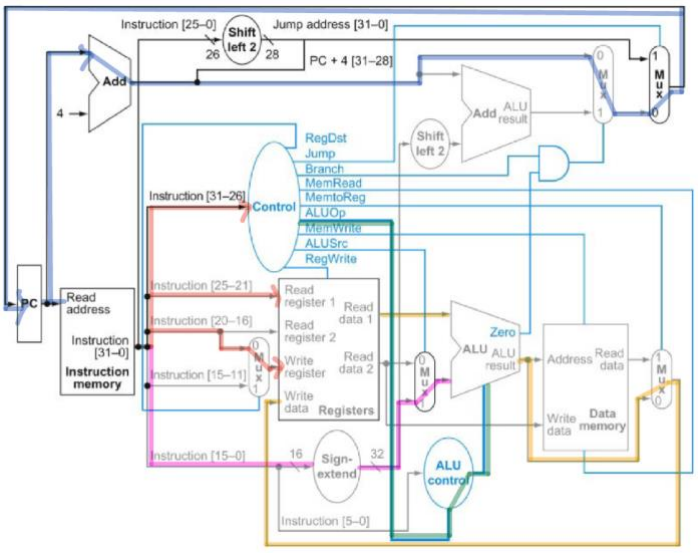




ALUop에 원하는 명령값이 들어갔고 input도 올바르게 들어감을 확인할 수 있다.

C. ADDIU – addiu $t $s imme – I-type

$t = $s + SE(i)



ADDIU 명령어는 ADDU 처럼 unsigned add를 진행하되 ADDIU는 imme 값을 받아

서 덧셈을 하는 I-type 명령어이다. 파란 path는 pc+4의 경로이다. 빨간 path는 Control unit과 RF의 값 불러오는 것을 관리하며, 분홍 path는 sign extend를 진행하여 ALU로 넘긴다. 노란 path는 ALU에서 연산하여 결과를 RF에 write한다. 녹색 path는 ALUop를 ALU에 전달한다.

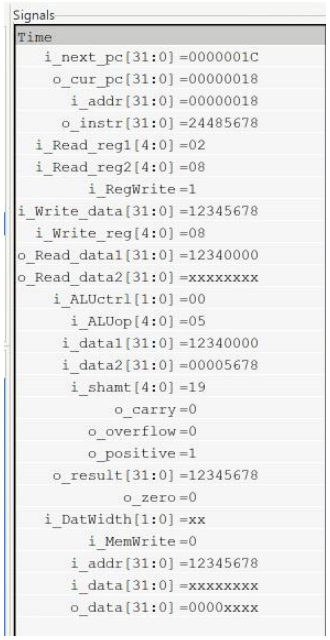
그러므로 PLA_OR은 00_00_1_1_01_00_00101_xxx_0_0_000_00_xxxxx

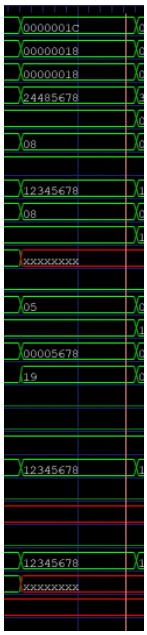
- Test_bench

  00111100_00000010_00010010_00110100 //lui $2 0x1234

  001001_00010_01000_0101_0110_0111_1000 //addiu $8 $2 0x5678

  $2에는 0x12340000이 저장되어 있고 imme는 0x00005678일 때, 이를 addiu로 unsigned add를 진행하면 0x12345678이다. 이는 아래 파형으로 확인할 수 있다.

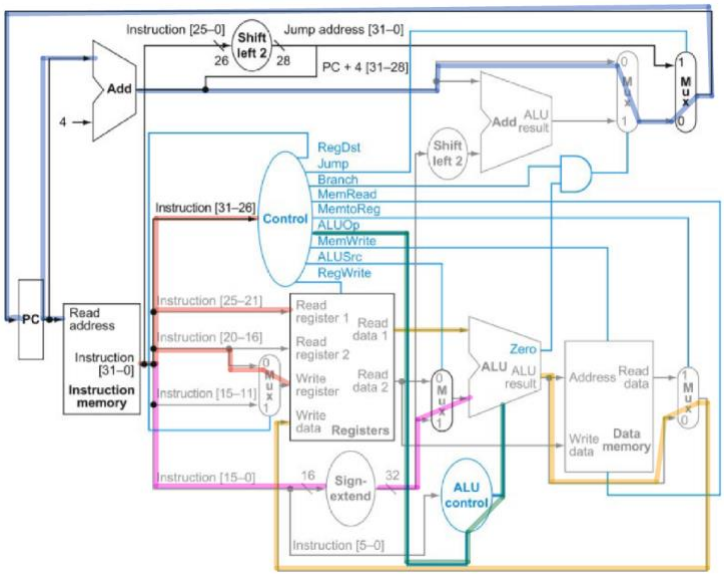




ALUop에 원하는 명령값이 들어갔고 input도 올바르게 들어감을 확인할 수 있다.

D. XORI – xori $t $s imme – I type

$t = $s ^ ZE(i)



XORI 명령어는 XOR 명령어에서 Imme 값을 B로 ALU로 가져와 연산하는 I-type 명령어이다. 파란 path는 pc+4하는 경로이고 빨간 path는 control unit과 RF의 값을 읽는 것을 관리한다. 분홍 Path는 Zero extend를 진행하는 경로이다. Sign-extend 타입을 결정하는 연결이 해당 회로에서는 보이지 않지만 testbench에서는 zero extend가 되었다. 노란 path를 통해서 ALU에서 연산되어 $t에 저장된다. 초록 path는 ALUop을 ALU에 전달한다.

그러므로 PLA_OR은 00_00_1_0_01_00_00011_xxx_0_0_000_00_xxxxx이다.

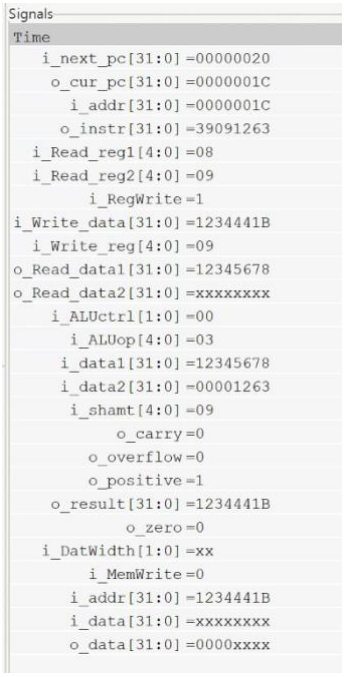
-Test_bench

00111100_00000010_00010010_00110100 //lui $2 0x1234

001001_00010_01000_0101_0110_0111_1000 //addiu $8 $2 0x5678

001110_01000_01001_0001_0010_0110_0011 //xori $9 $8 0x1263

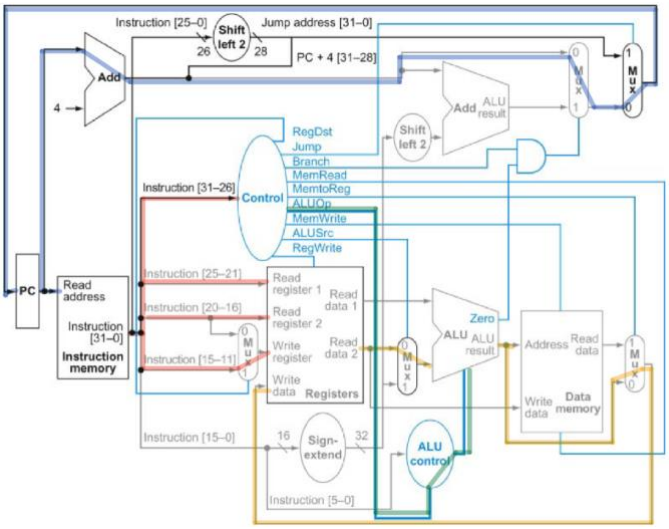
xori 명령어는 $9레지스터에 $8 레지스터 값과 0x1263을 xor 하였을 때 값을 저장한다. $8의 값은 이전 addiu 명령으로 0x12345678이 되었고 0x1263은 0x00001263이다. 이를 xor 연산하면 0b0001_0010_0011_0100_0101_0110_0111_1000 과 0b0000_0000_0000-0000_0001_0010_0110_0011 의 xor 연산과 같으므로 이를 계산하면 0b0001_0010_0011_0100_0100_0100_0001_1011는 0x1234441B이다. 이는 아래 파형과 같다.



ALUop에 원하는 명령값이 들어갔고 input도 올바르게 들어감을 확인할 수 있다.

E. SLL – sll $d $t shamt – R-type

$d = $t << a



SLL 명령어는 Shift left logical 명령으로 shamt에 지정된 값만큼 left shift를 하는 R-type 명령어이다. $s는 쓰이지 않고 ALU에서 shift 연산을 한다. 파란 path는 pc+4를 하는 경로이고 빨간 path는 Control unit과 RF로부터 값을 불러오도록 제어한다. 노란 path는 ALU에서 shift 연산을 하고 이는 $d에 저장된다. 초록 path는 ALUop를 ALU에 전달한다.

그러므로 PLA_OR은 01_00_1_x_00_00_01101_xxx_0_0_000_00_xxxxx이다.

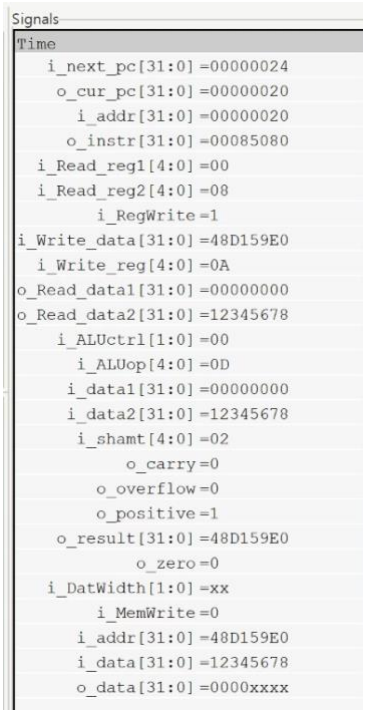
- Test_bench

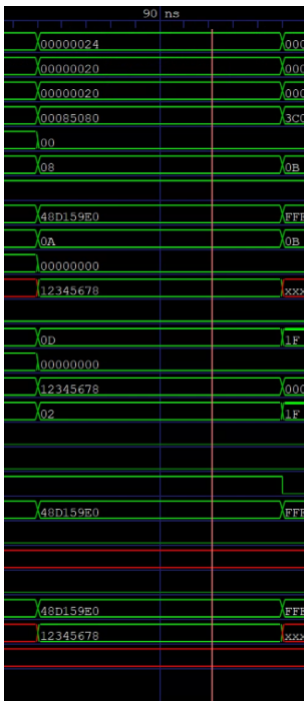
00111100_00000010_00010010_00110100 //lui $2 0x1234

001001_00010_01000_0101_0110_0111_1000 //addiu $8 $2 0x5678

000000_00000_01000_01010_00010_000000 //ssl $10 $8 0x2

이 때 $8은 이전에 했던 덧셈값으로 0x12345678이다. ssl은 $8 레지스터를 shamt 만큼 shift left logical을 하는데 이 때 shamt는 0b00010으로 2이다. 이를 shift left하면 $8 레지스터 값에 4를 곱한 것과 같다. 즉 0x48D159E0 가 $10 레지스터에 저장된다. 이에 대한 파형은 아래와 같다.

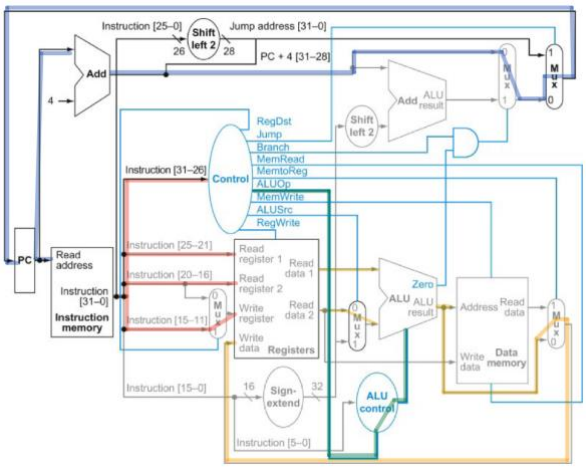




ALUop에 원하는 명령값이 들어갔고 input도 올바르게 들어감을 확인할 수 있다.

F. SRAV – srav $d $t $s – R-type

$d = $t >>> $s



SRAV 명령어는 Shift right arithmetic variable 이란 명령어로 $t을 $s 만큼 arithmetic shift right 하는 R-type 명령어이다. 파란 path는 pc+4하는 경로이고 빨간 path는 control unit과 RF로부터 값을 불러내도록 제어한다. 노란 path는 $t를 $s 만큼 arithmetic shift right 하기 위해 ALU에 들어가 연산되어 결과는 %d에 저장된다. 초록 path는 ALUop를 ALU에 전달한다.

그러므로 PLA_OR은 01_00_1_x_00_01_01111_xxx_0_0_000_00이다.

- Test_bench

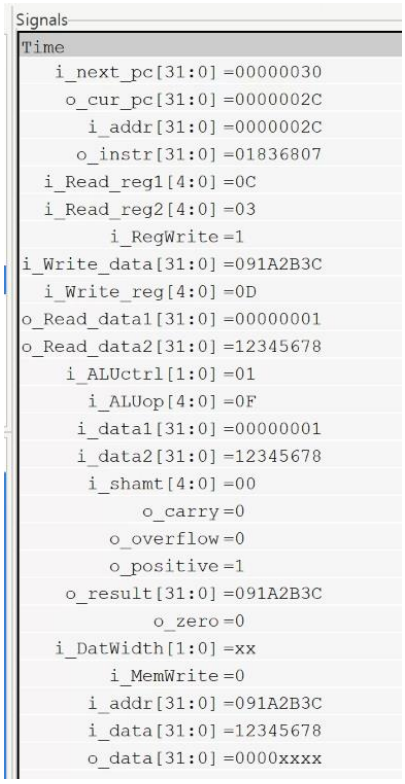
  00111100_00000010_00010010_00110100 //lui $2 0x1234

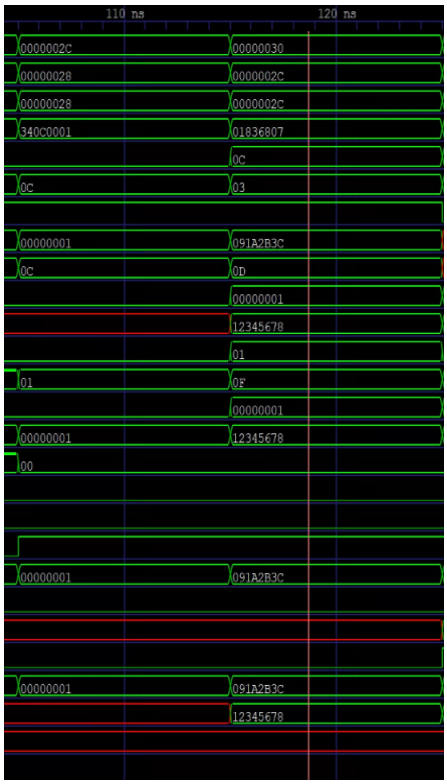
  00110100_01000011_01010110_01111000 //ori $3 $2 0x5678

  001101_00000_01100_0000_0000_0000_0001 //ori $12 $0 0x0001

  000000_01100_00011_01101_00000_000111 //srav $13 $3 $12

  srav의 $d는 $13이고 $t는 $3이고 $s 는 $12이다. 이 때 $3은 0x12340000과 0x00005678을 or 연산한 0x12345678이다. $t는 0x0과 0x0001을 or 연산한 0x00000001이다. 즉 srav는 0x12345678을 1만큼 arithmetic shift right한 결과이다. 결과는 0x091A2B3C이다. 이는 아래 파형을 통해 확인할 수 있다.



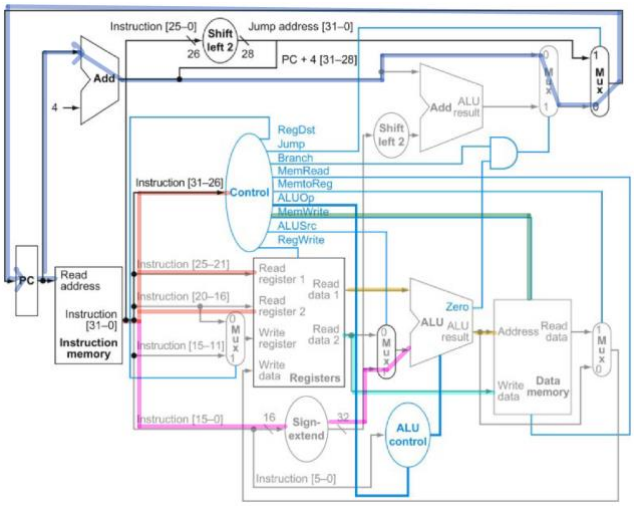


ALUop에 원하는 명령값이 들어갔고 input도 올바르게 들어감을 확인할 수 있다.

ori $12 $0 0x0001을 한 결과와 srav $13 $3 $12한 결과가 파형으로 나타났다.

G. SH – SH $5 $8 0x0010 – I-type

MEM[$s+i]:2 = LH($t)

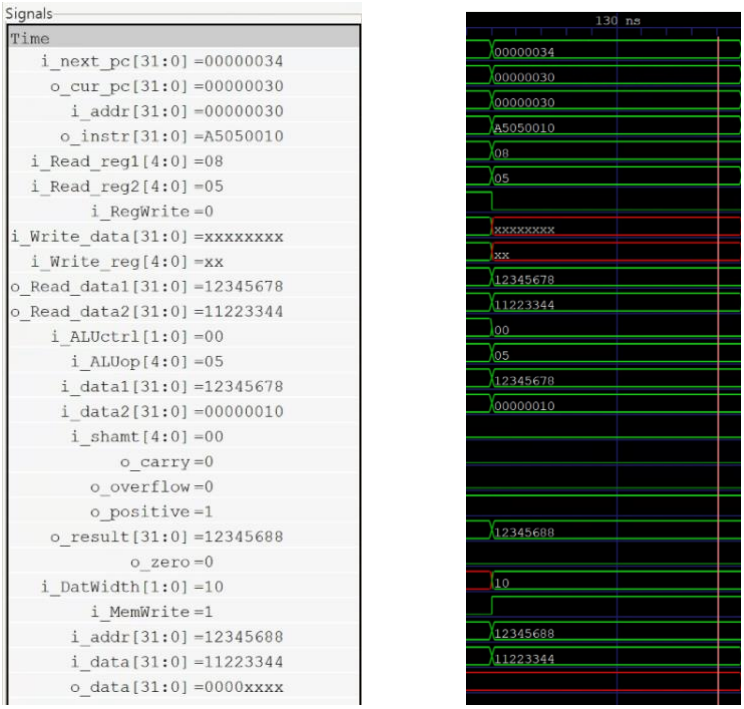


SH 명령어는 store half-word 16bit(2byte)를 실행하는 I-type 명령어로 하위 16bit만큼 메모리에 저장한다. 메모리에 저장할 때도 파란 path는 pc+4를 진행하는 경로이고 빨간 path는 control unit과 RF로부터 값을 불러오도록 제어한다. 분홍 path는 sign extender를 거쳐 ALU에 입력되고 이때 노란 path와 같이 덧셈 연산을 하여 memory address에 접근한다. 하늘 path는 $t로부터 값을 읽어 메모리에 저장하도록 한다. 초록 path는 MemWrite 권한을 부여하는 경로이다.
그러므로 PLA_OR은 xx_xx_0_x_01_00_00101_010_1_0_000_00_xxxxx이다.

- Test_bench

  00111100_00000010_00010010_00110100 //lui $2 0x1234
  00111100_00000100_00010001_00100010 //lui $4 0x1122
  00110100_10000101_00110011_01000100 //ori $5, $4, 0x3344
  001001_00010_01000_0101_0110_0111_1000//addiu $8 $2 0x5678
  101001_01000_00101_0000_0000_0001_0000//sh $5 $8 0x0010

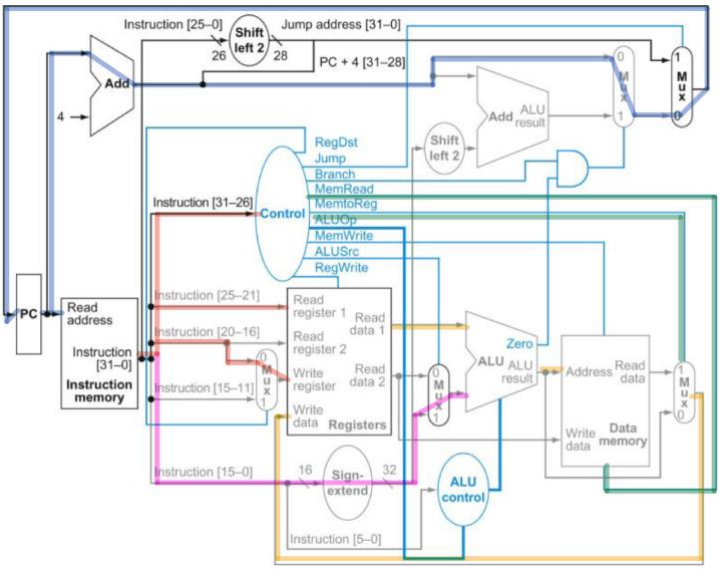
  $5 값에는 0x11220000과 0x00003344를 or 연산한 값인 0x11223344이고 $8은 0x12340000과 0x00005678을 합한 0x12345678이다. sh 명령을 실행하면 $s인 $8 레지스터는 주소로 들어가고 $t인 $5 레지스터는 메모리에 입력되도록한다. 이때 half-word만 저장되므로 하위 2byte만 저장된다. 주소로 0x12345678+0x0010을 더하면 0x12345688 주소에 0x3344가 메모리에 저장되었다. 이 후 저장된 값은 LH 명령을 통해 확인한다. 아래는 SH 명령에 대한 파형 결과이다.



ALUop에 원하는 명령값이 들어갔고 input도 올바르게 들어감을 확인할 수 있다.

H. LH - LH $14 $8 0x0010 – I-type

$t = SE(MEM[$s+i]:2)



LH 명령어는 load half word를 하는 명령어로 주소에 저장된 값을 $t에 저장하는 I-type 명령어이다. 파란 path는 PC+4이고 빨간 path는 control unit과 RF로부터 불러오는 값을 제어한다. 분홍 path는 imme 값을 extend 하고 노란 path와 함께 alu에서 덧셈 연산이 진행되어 memory address에 접근한다. 또한 녹색 path를 통해 memRead와 memtoReg를 허용하여 노란 path가 $t에 저장될 수 있도록 한다.

그러므로 PLA_OR은 00_00_1_1_01_00_00101_110_0_1_000_00_xxxxx이다.

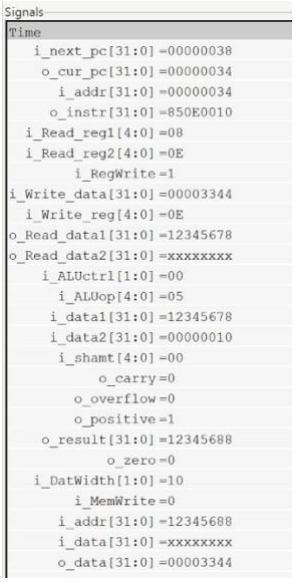
- Test_bench

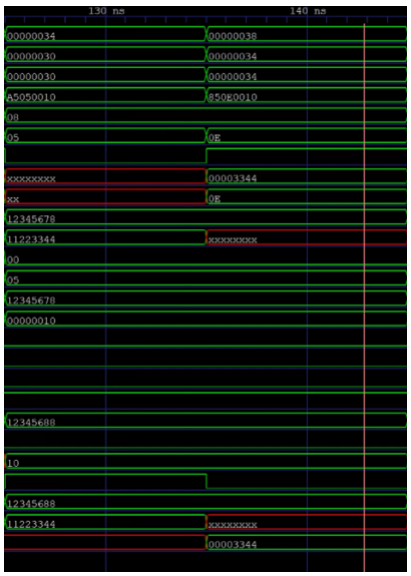
00111100_00000010_00010010_00110100 //lui $2 0x1234

001001_00010_01000_0101_0110_0111_1000//addiu $8 $2 0x5678

100001_01000_01110_0000_0000_0001_0000//lh $14 $8 0x0010

LH에서 $8은 0x12345678이고 이에 10을 더하여 memory 주소에 접근한다. 이는 이전 SH에서 값을 저장한 0x12345688과 같다. 이 때 값을 읽으면 전에 저장한 하위 2byte 값이고 이는 0x3344이다. 이는 sign extend 되어 $14에 저장된다.



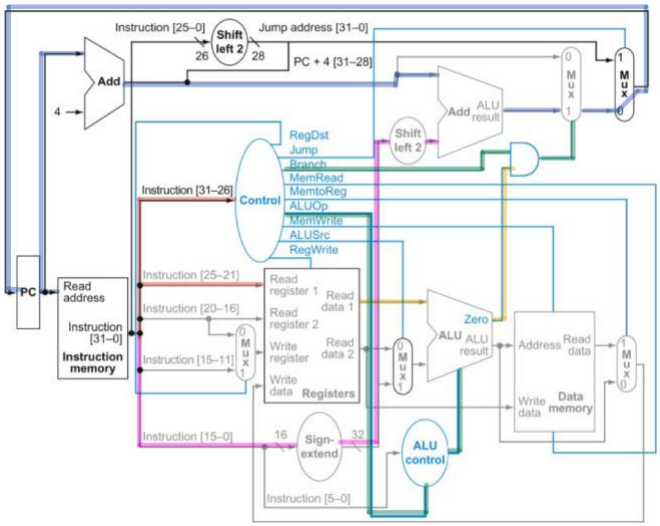


ALUop에 원하는 명령값이 들어갔고 input도 올바르게 들어감을 확인할 수 있다.

파형 사진은 sh 후에 해당 주소로 lh 명령을 실행했을 때 파형이다.

I. BLTZ – bltz $s imme – I type

if($s<0), pc +=2<<2



BLTZ 명령어는 Branch less than zero 명령으로 $s 값이 0보다 작으면 imme 만큼 branch를 진행한다. 파란 path는 pc+4를 더한 후 imme 값을 shift left 2(4배)를 하여 더한 후 이를 다시 pc에 저장한다. 빨간 path는 control unit과 RF로부터 값을

불러오도록 하고 분홍 path는 imme 값을 extend 하여 이전 파란 path와 함께 더해진다. 노란색 path는 0과 set less than 연산을 진행하고 이것의 참과 거짓을 판단하여 위의 AND연산으로 보내 MUX signal을 보낸다.

그러므로 PLA_OR은 xx_xx_0_0_10_00_10000_xxx_0_0_111_00_xxxxx 이다.

- Test_bench

001111_00000_01011_1111_1111_1111_0110//lui $11 0xfff6

000001_01011_00000_0000_0000_0000_0001//bltz $11 $0x0001

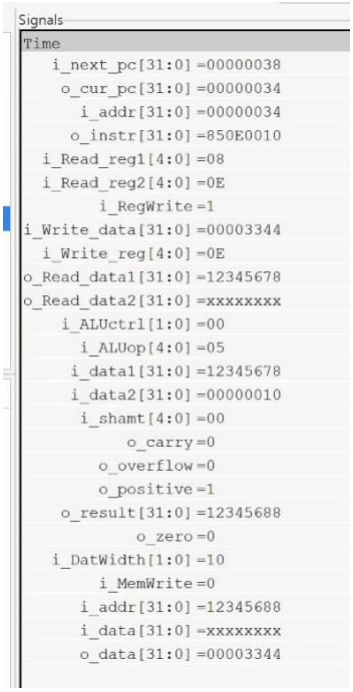
000000_00010_00011_01111_00000_100001// addu $15 $2 $3

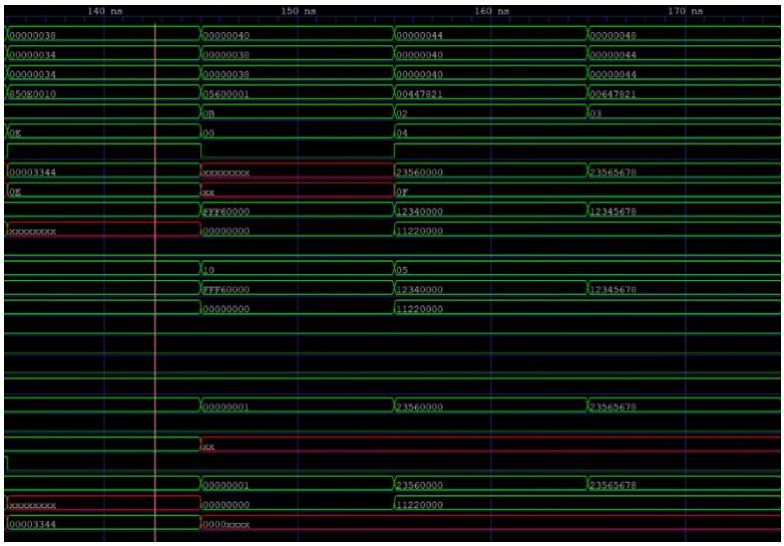
000000_00010_00100_01111_00000_100001// addu $15 $2 $4

000000_00011_00100_01111_00000_100001// addu $15 $3 $4

000000_00011_00101_01111_00000_100001// addu $15 $2 $3

$11 레지스터 값은 0xFFF60000이고 이를 2's complement 수로 보면 음수이다. 이를 bltz 명령의 $s에 넣어서 비교하면 0보다 작으므로 참인 결과가 나온다. 이는 위에서 말한 AND연산에 들어가 imme 의 4배값의 덧셈만큼 더하여 PC값을 새롭게 저장한다. 예를 들어 0x0001이면 4배를 하여 pc+4+4를 하여 그 다음 명령으로 이동한다. 아래는 해당 예시의 파형이다.



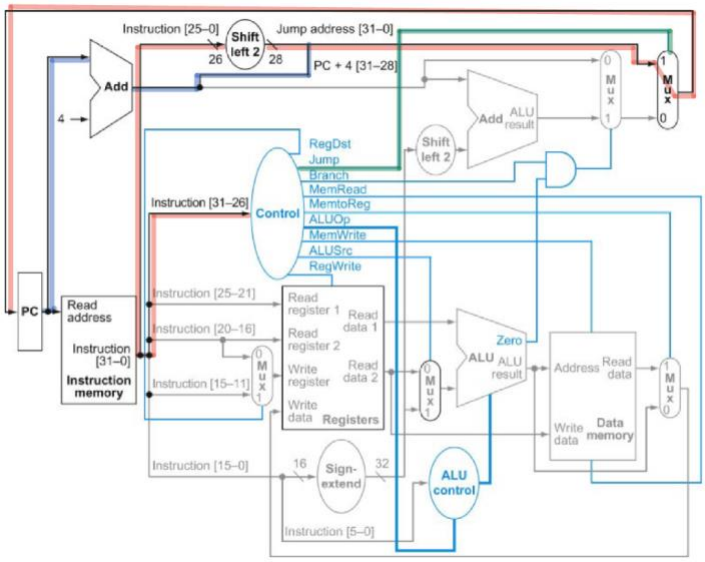


ALUop에 원하는 명령값이 들어갔고 input도 올바르게 들어감을 확인할 수 있다.

bltz의 명령의 pc값은 0x38이고 다음은 0x3C가 되어야 하지만 0x40이 되어 addu $15 $2 $3이 아닌 addu $15 $2 $4가 되어 해당 명령이 실행됨을 확인할 수 있다.

J. JAL- jal imme – J-type

$31 = pc; pc=pc4|i26<<2



JAL 명령은 jump and link 명령으로 imme 해당 위치로 jump하고 $31에는 pc를 저장한다. 파란 path와 빨간 path는 imme를 shift left 2 하고 이를 pc+4 더한 값[31-28]을 jump address [31-28]을 더하여 다음 pc 값으로 둔다. 이 jump는 ALU를 사용하지 않는다. 초록 path는 jump 값이 pc로 가도록 mux를 조절한다.

그러므로 PLA_OR은 10_11_1_1_xx_xx_xxxxx_xxx_0_0_000_01_xxxxx이다.

- Test_bench

  00111100_00000010_00010010_00110100 //lui $2 0x1234

  00110100_01000011_01010110_01111000 //ori $3 $2 0x5678

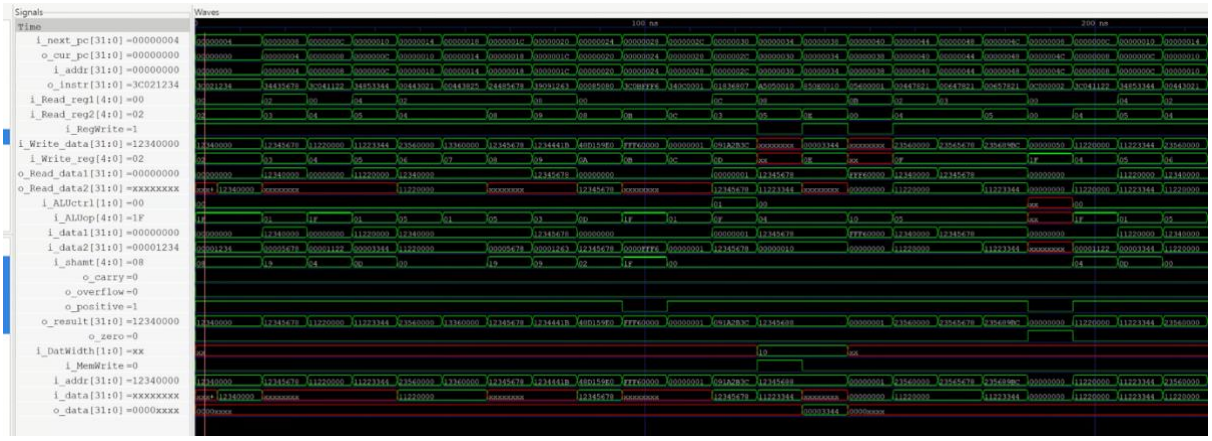
  00111100_00000100_00010001_00100010 //lui $4 0x1122

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

  000011_00_0000_0000_0000_0000_0000_0010 //jal 0x2

  를 하면 계속 pc 8로 루프가 된다.

JAL을 마지막으로 넣었을 때 전체 결과이다. 이 또한 명령을 다 실행하고 PC=8로 돌아간다.



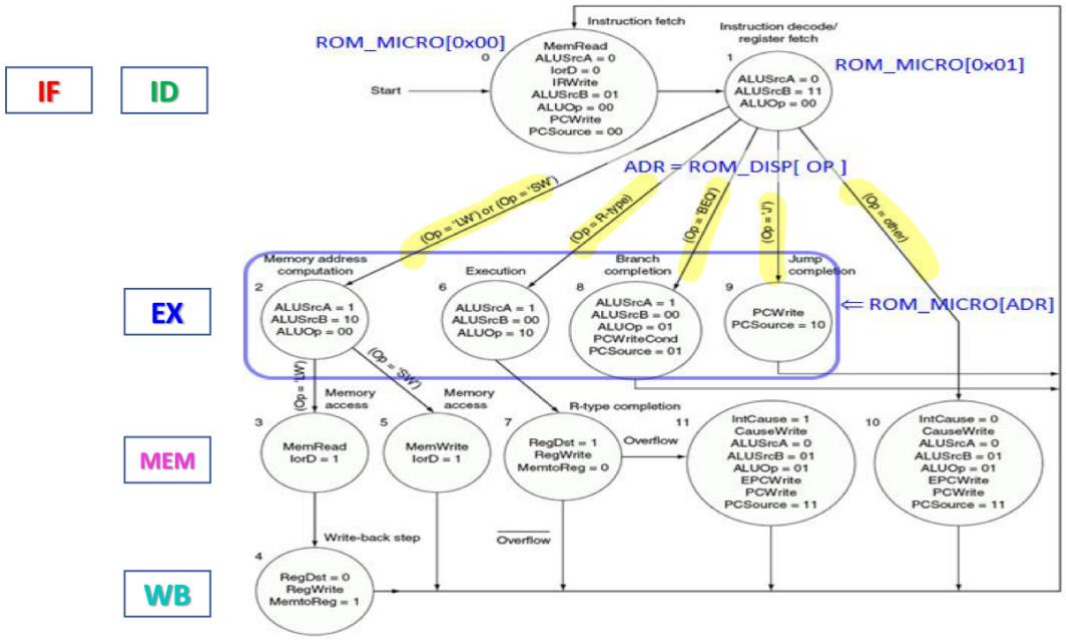
위 과정에서 얘기했듯이 예상했던 결과와 시뮬레이션에서 보이는 파형의 값은 모두 동일하게 잘 나왔음을 확인하였다.

3. 고찰

PLA_AND는 Instruction decoding configuration를 만들 때는 Mips inst datasheet pdf 파일을 읽고 그저 대입하면 되었다. 하지만 PLA_OR은 Control Signal Configuration을 만들 때는 Control Signals for SCPU.pdf를 참고하여 만들 때는 해당 명령이 어떻게 진행되는지 직접 회로에서 그려가며 path를 확인해야 했다. 대신 PLA_OR을 보면 해당 명령마다 식이 어떻게 되는지 자세히 적혀있어서 필요한 ALU가 뭔지 알 수 있었다. 예를 들어 덧셈 연산이면 ALUop에서 해당 bit를 가져오고 RF에 write하면 RegWrite 값을 1로 활성화하였다. 이 때 RF에 쓰는 것이 없으면 RegDst RegDatSel은 xx로 무시해도 되며 MEM도 쓰는 것이 없으면 xx로 무시해도 된다. 또한 immediate 값이 없으면 extendmode 또한 무시해도 된다.

컴퓨터 구조라는 과목이 주는 두려움이 매우 커서 지금까지 모든 프로젝트 모든 과제에 대해서 어렵게 느끼곤 했는데 작년까지 해왔던 컴퓨터공학기초실험이 연계되어 해당 내용을 이해하는 것과 과제를 해내는 것이 비교적 편안하게 느껴진 이번 프로젝트였다. 컴퓨터공학기초실험은 모든 과정을 직접 만들어야 해서 막막했지만 컴퓨터구조는 이미 만들어진 것에 대해 이해만 할 수 있다면 풀 수 있음을 알게 되었다.

이번 Singcle cycle CPU의 문제점은 하나의 명령이 끝날 때까지 다음 명령을 진행할 수 없다는 것이다. 이는 곧 delay를 발생하게 하고 느려진다. 또한 load를 하는 명령의 time이 제일 긴데 이에 맞춰 모든 명령에 time을 할당하여서 빨리 끝날 수 있는 명령도 정해진 시간만큼 delay되어야 함이 문제이다. 이는 다음 과제인 pipelined cpu 혹은 mulpi Cycle cpu를 통해 해결할 수 있으며 명령이 Fetch, Decoding, Excute,Memory, Writeback의 각 진행을 모든 과정이 아닌 한 과정이 끝나면 다른 명령어가 Fetch 되며 시간 딜레이를 줄일 수 있다. 이는 아래 사진과 같은 순서로 만들 수 있다.



마지막으로 이번 프로젝트를 통해 R-type, I-type, J-type에 대해 더 자세히 알고 각 type 별로 instruction을 어떻게 두어야 하는지 이해할 수 있었다.

4. Reference

1)컴퓨터구조실험week5~7/ 컴퓨터구조실험/광운대학교/이성원 교수님/2023

2)컴퓨터구조실험강의자료/ 컴퓨터구조/광운대학교/이성원 교수님/2023

3)컴퓨터구조프로젝트보강영상/ 컴퓨터구조/광운대학교/ 이성원 교수님/2023