컴퓨터구조 프로젝트 보고서

Project 1

MIPS Single Cycle CPU Implementation

수업 명: 컴퓨터구조

담당 교수: 이성원 교수님

학과: 컴퓨터정보공학부

학번: 2023202070

이름: 최현진

제출일: 2025.04.17

1. Introduction

MIPS Single Cycle CPU에서, 기본 명령어 lw, sw, ori, j, lui, llo, lhi는 구현되어 있고, 추가로 AND, NOR, ADDI, SLTU, SRL, SH, LB, BNE, BGEZ, JALR을 설계해야 한다. 이를 위해 PLA(Programmable Logic array) 구조를 이용해서 명령어를 해석하고, 제어 신호를 생성해야 한다. PLA_AND.txt에는 각 명령어에 대해 opcode, func, regimm를 설정한다. PLA_OR.txt에는 설정한 줄에 맞춰 해당 명령이 작동할 control signal를 채워 넣는다. 구현한 명령어들을 testbench waveform 비교 분석을 통해 동작을 검증한다.

2. Assignment

<초기 M_TEXT_SEG.txt>

001111_00000_00010_0001001000110100 // lui $2, 0x1234

001101_00010_00011_0101011001111000 // ori $3, $2, 0x5678

001111_00000_00100_0001000100100010 // lui $4, 0x1122

001101_00100_00101_0011001101000100 // ori $5, $4, 0x3344

<실행 결과>

$2 = 0x12340000

$3 = $2 | 0x5678 = 0x12340000 | 0x00005678 = 0x12345678

$4 = 0x11220000

$5 = 0x11220000 | 0x00003344 = 0x11223344

1. AND

두 레지스터의 비트 단위 AND 연산 결과를 레지스터에 저장한다, R-type

Syntax: f $d, $s, $t

Operation: $d = $s & $t

Op	Func	Regimm
000000	100100	xxxxx

000000_100100_xxxxx // 0x15 : and

RegDst	RegDatSel	RegWrite	SEUmode	ALUsrcB	ALUctrl
01	00	1	x	00	0x
ALUop	DataWidth	MemWrite	MemtoReg	Branch	Jump
00000	xxx	0	0	000	00

01_00_1_x_00_0x_00000_xxx_0_0_000_00_xxxxx // 0x15 : and $d = $s & $t

RegDst: 01, 결과값을 rd에 쓸 것이므로 목적지 레지스터로 rd를 선택한다.

RegDatSel: 00, ALU 결과값을 register file에 쓰도록 선택한다.

RegWrite: 1, register file에 값을 쓴다.

SEUmode: x, imm값을 extension할 필요가 없다.

ALUsrcB: 00, ALU 입력으로 register file port B를 사용한다.

ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]= x (shift 수행하지 않음)

ALUop: 00000, bitwise AND

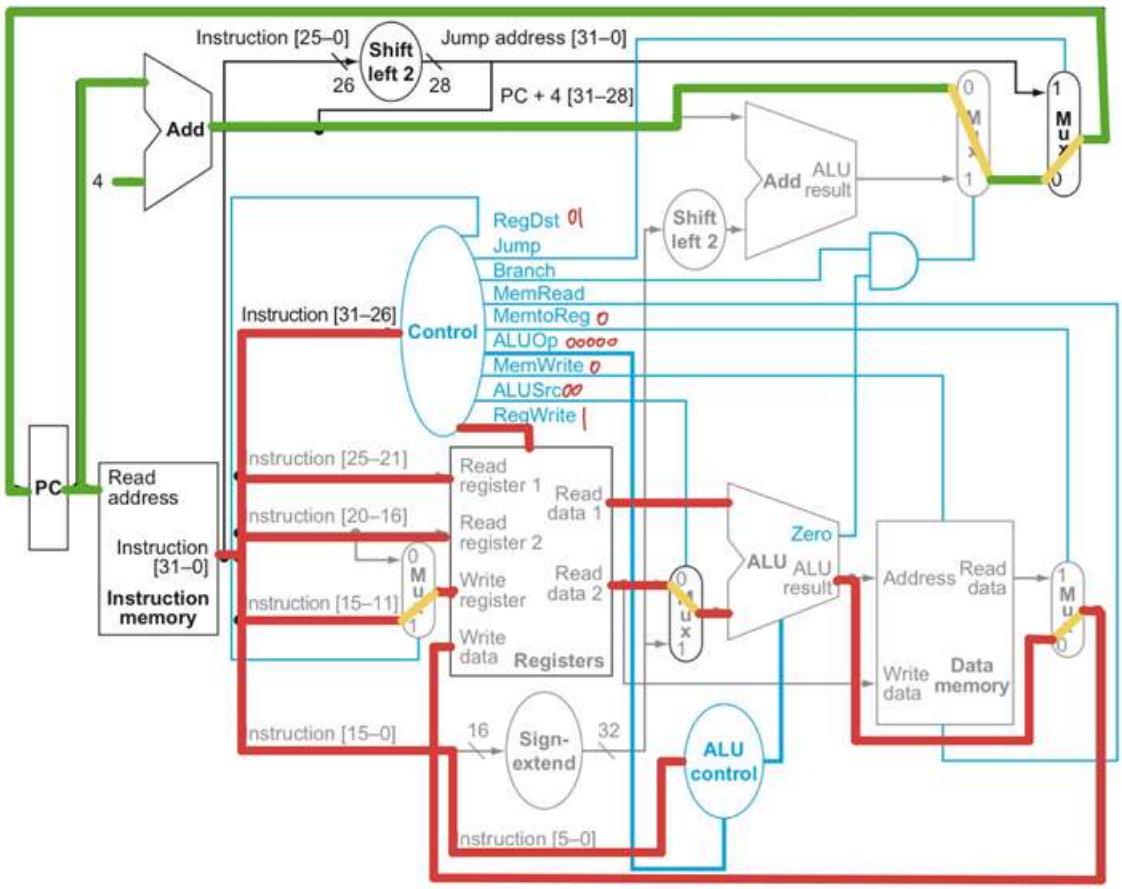
DataWidth: xxx, 메모리 접근하지 않음.

MemWrite: 0, 메모리에 쓰지 않음.

MemtoReg: 0, register file에 ALU값을 쓴다.

Branch: 000, Use PC+4

Jump: 00, jump 수행하지 않음.



$s, $t 레지스터를 읽고 ALU에서 bitwise AND연산한 결과가 $d 레지스터에 저장된다. PC=PC+4 이다.

2. NOR

두 레지스터의 비트 단위 NOR 연산 결과를 레지스터에 저장한다, R-type

Syntax: f $d, $s, $t

Operation: $d = ~($s | $t)

Op	Func	Regimm
000000	100111	xxxxx

000000_100111_xxxxx // 0x18 : nor

RegDst	RegDatSel	RegWrite	SEUmode	ALUsrcB	ALUctrl
01	00	1	x	00	0x
ALUop	DataWidth	MemWrite	MemtoReg	Branch	Jump
00010	xxx	0	0	000	00

01_00_1_x_00_0x_00010_xxx_0_0_000_00_xxxxx // 0x18 : nor $d = ~($s | $t)

RegDst: 01, 결과값을 rd에 쓸 것이므로 목적지 레지스터로 rd를 선택한다.

RegDatSel: 00, ALU 결과값을 register file에 쓰도록 선택한다.

RegWrite: 1, register file에 값을 쓴다.

SEUmode: x, imm값을 extension할 필요가 없다.

ALUsrcB: 00, ALU 입력으로 register file port B를 사용한다.

ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]= x (shift 수행하지 않음)

ALUop: 00010, bitwise NOR

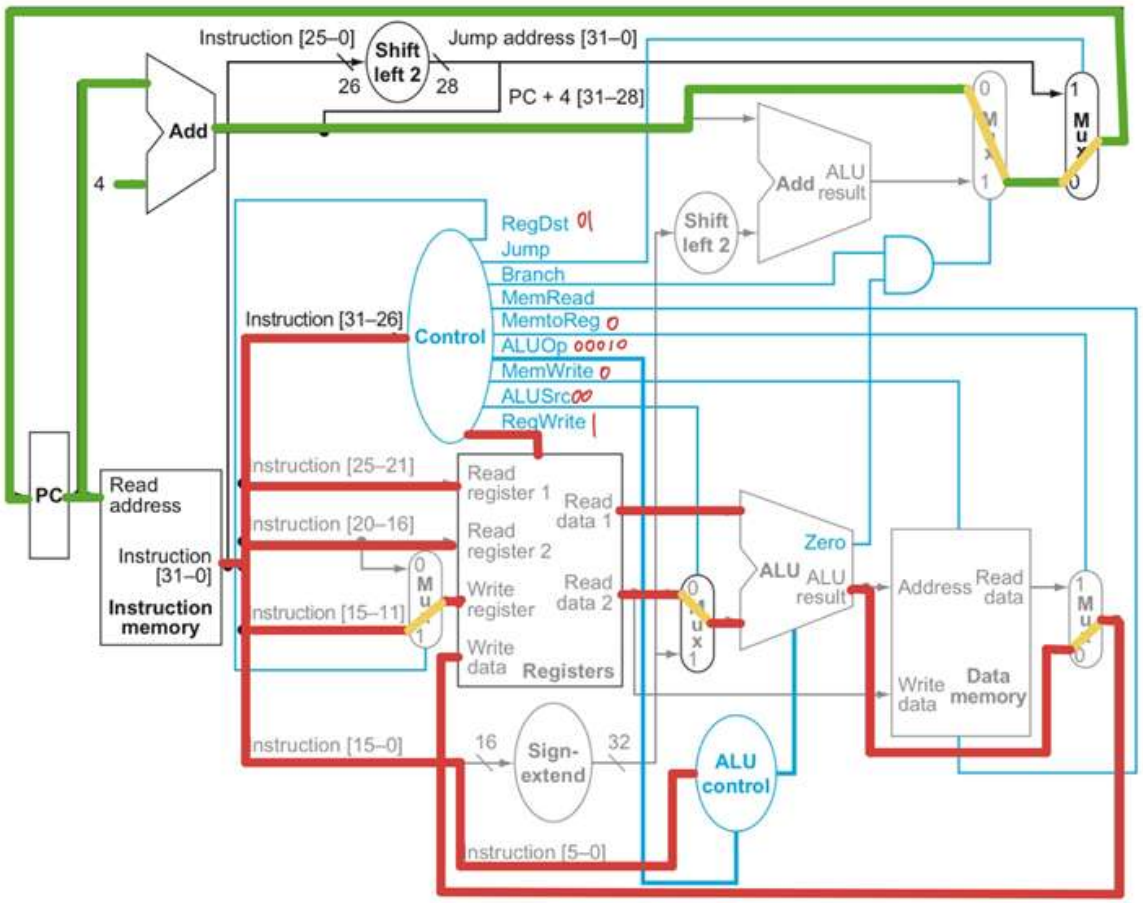
DataWidth: xxx, 메모리 접근하지 않음.

MemWrite: 0, 메모리에 쓰지 않음.

MemtoReg: 0, register file에 ALU값을 쓴다.

Branch: 000, Use PC+4

Jump: 00, jump 수행하지 않음.



$s, $t 레지스터를 읽고 ALU에서 bitwise NOR 연산한 결과가 $d 레지스터에 저장된다. PC=PC+4이다.

3. ADDI

imm값을 sign extend하여 $s와 더한 결과를 $t에 저장한다, I-type

Syntax: o $t, $s, i

Operation: $t = $s + SE(i)

Op	Func	Regimm
001000	xxxxxx	xxxxx

001000_xxxxxx_xxxxx // 0x23 : addi

RegDst	RegDatSel	RegWrite	SEUmode	ALUsrcB	ALUctrl
00	00	1	1	01	0x
ALUop	DataWidth	MemWrite	MemtoReg	Branch	Jump
00100	xxx	0	0	000	00

00_00_1_1_01_0x_00100_xxx_0_0_000_00_xxxxx // 0x23 : addi $t = $s + SE(i)

RegDst: 00, 결과값을 rt에 쓸 것이므로 목적지 레지스터로 rt를 선택한다.

RegDatSel: 00, ALU 결과값을 register file에 쓰도록 선택한다.

RegWrite: 1, register file에 값을 쓴다.

SEUmode: 1, imm값을 sign extension한다.

ALUsrcB: 01, ALU 입력으로 imm값을 사용한다.

ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]=x (shift 수행하지 않음)

ALUop: 00100, a + b

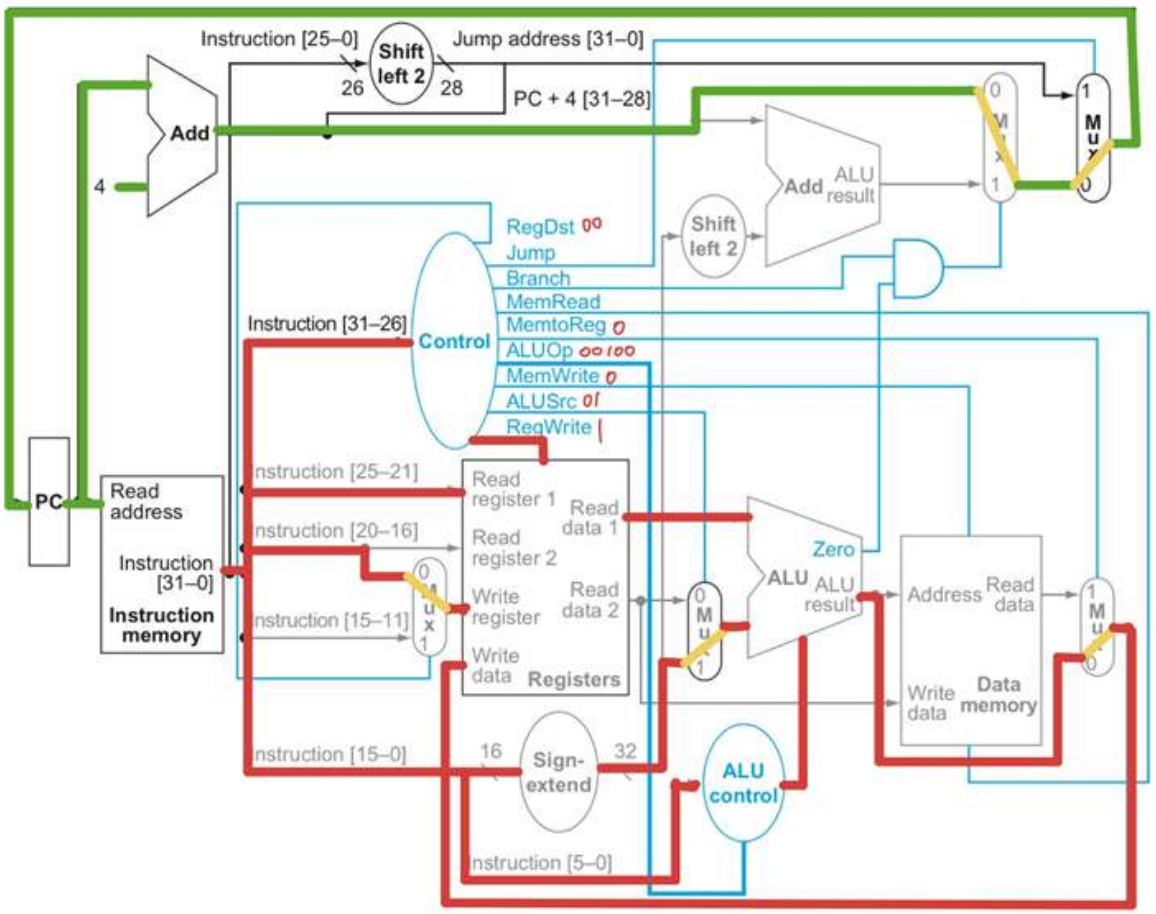
DataWidth: xxx, 메모리 접근하지 않음.

MemWrite: 0, 메모리에 쓰지 않음.

MemtoReg: 0, register file에 ALU값을 쓴다.

Branch: 000, Use PC+4

Jump: 00, jump 수행하지 않음.



$s를 읽고, imm값을 sign extend하여 ALU에서 $s와 더한 결과를 $t에 저장한다 PC=PC+4이다.

4. SLTU

Set Less Than Unsigned, unsigned 기준으로 $s < $t 비교 결과를 $d에 저장한다, R-type

Syntax: f $d, $s, $t

Operation: $d = ($s < $t)

Op	Func	Regimm
000000	101011	xxxxx

000000_101011_xxxxx // 0x1a : sltu

RegDst	RegDatSel	RegWrite	SEUmode	ALUsrcB	ALUctrl
01	00	1	x	00	0x
ALUop	DataWidth	MemWrite	MemtoReg	Branch	Jump
10001	xxx	0	0	000	00

01_00_1_x_00_0x_10001_xxx_0_0_000_00_xxxxx // 0x1a : sltu $d = ($s < $t)

RegDst: 01, 결과값을 rd에 쓸 것이므로 목적지 레지스터로 rd를 선택한다.

RegDatSel: 00, ALU 결과값을 register file에 쓰도록 선택한다.

RegWrite: 1, register file에 값을 쓴다.

SEUmode: x, imm값을 extension할 필요가 없다.

ALUsrcB: 00, ALU 입력으로 register file port B를 사용한다.

ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]=x (shift 수행하지 않음)

ALUop: 10001, Unsigned SLT

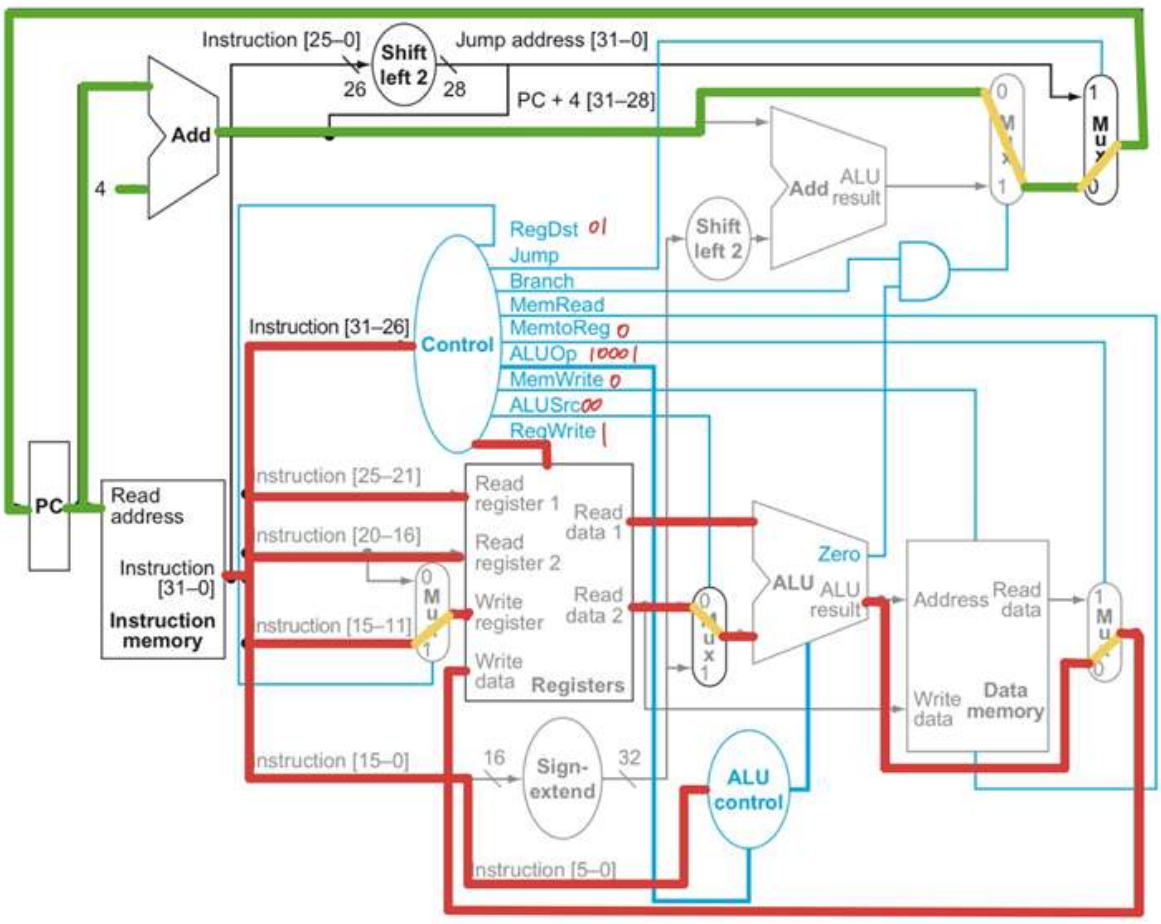
DataWidth: xxx, 메모리 접근하지 않음.

MemWrite: 0, 메모리에 쓰지 않음.

MemtoReg: 0, register file에 ALU값을 쓴다.

Branch: 000, Use PC+4

Jump: 00, jump 수행하지 않음.



$s, $t 레지스터를 읽고 ALU에서 Unsigned SLT 연산한 결과가 $d 레지스터에 저장된다. PC=PC+4이다.

5. SRL

Shift Right Logical: shamt만큼 logical shift한 값을 레지스터에 저장한다. R-type

Syntax: f $d, $t, sa

Operation: $d = $t >> a

Op	Func	Regimm
000000	000010	xxxxx

000000_000010_xxxxx // 0x01 : srl

RegDst	RegDatSel	RegWrite	SEUmode	ALUsrcB	ALUctrl
01	00	1	x	00	00
ALUop	DataWidth	MemWrite	MemtoReg	Branch	Jump
01110	Xxx	0	0	000	00

01_00_1_x_00_00_01110_xxx_0_0_000_00_xxxxx // 0x01 : srl $d = $t >> a

RegDst: 01, 결과값을 rd에 쓸 것이므로 목적지 레지스터로 rd를 선택한다.

RegDatSel: 00, ALU 결과값을 register file에 쓰도록 선택한다.

RegWrite: 1, register file에 값을 쓴다.

SEUmode: x, imm값을 extension할 필요가 없다.

ALUsrcB: 00, ALU 입력으로 register file port B를 사용한다.

ALUctrl: 00, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]=0 (shift = shift amount)

ALUop: 01110, b >> a

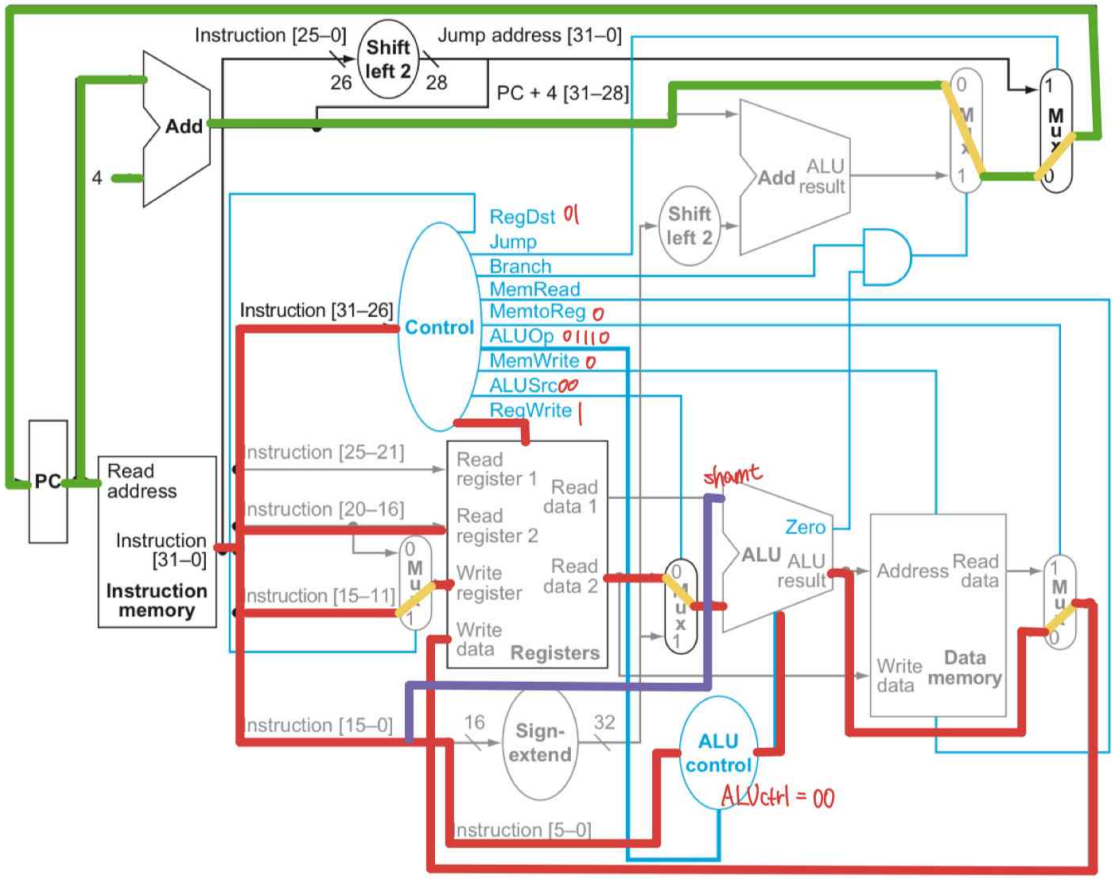
DataWidth: xxx, 메모리 접근하지 않음.

MemWrite: 0, 메모리에 쓰지 않음.

MemtoReg: 0, register file에 ALU값을 쓴다.

Branch: 000, Use PC+4

Jump: 00, jump 수행하지 않음.



$t를 읽고, ALU contorl을 통해 shamt만큼 ALU에서 logical shift 연산한다는 것을 새로운 선을 추가해서 표현했다. ALU 결과값을 $d에 저장한다. PC=PC+4이다.

6. SH

Store Halfword: MEM [$s + i]에 레지스터 $t의 하위 2바이트 값을 쓴다, I-type

Syntax: o $t, i ($s)

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

Op	Func	Regimm
101001	xxxxxx	xxxxx

101001_xxxxxx_xxxxx // 0x31 : sh

RegDst	RegDatSel	RegWrite	SEUmode	ALUsrcB	ALUctrl
xx	xx	0	1	01	0x
ALUop	DataWidth	MemWrite	MemtoReg	Branch	Jump
00100	010	1	x	000	00

xx_xx_0_1_01_0x_00100_010_1_x_000_00_xxxxx // 0x31 : sh MEM [$s + i]:2 = LH ($t)

RegDst: xx, register file에 쓰지 않음.

RegDatSel: xx, register file에 쓰지 않음

RegWrite: 0, register file에 값을 쓰지 않음..

SEUmode: 1, imm값을 sign extension한다.

ALUsrcB: 01, ALU 입력으로 imm값을 사용한다.

ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]=x (shift 수행하지 않음)

ALUop: 00100, a + b

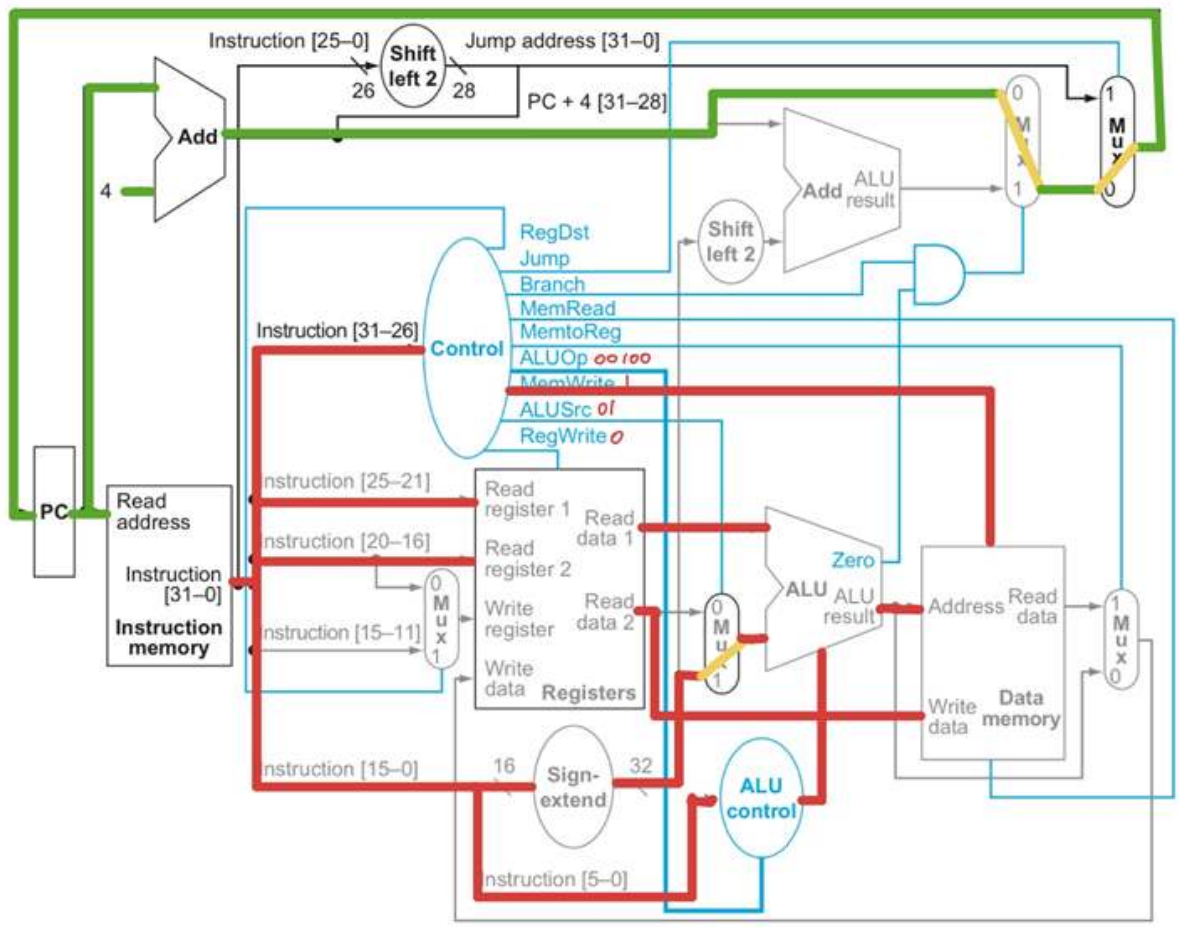
DataWidth: 010, 16-bit Halfword

MemWrite: 1, 메모리에 씀..

MemtoReg: x, register file에 쓰지 않음

Branch: 000, Use PC+4

Jump: 00, jump 수행하지 않음.



$t, $s를 읽고, 메모리 [$s + i]에 $t의 하위 2바이트 값을 쓴다. imm값을 sign extend하여 $s와 ALU에서 주소 연산한다. PC=PC+4이다.

7. LB

Load Byte: MEM [$s + i]의 1바이트 데이터를 가져와서 sign extend 후 $t에 저장한다, I-type

Syntax: o $t, i ($s)

Operation: $t = SE (MEM [$s + i]:1)

Op	Func	Regimm
100000	xxxxxx	xxxxx

100000_xxxxxx_xxxxx // 0x2b : lb

RegDst	RegDatSel	RegWrite	SEUmode	ALUsrcB	ALUctrl
00	00	1	1	01	0x
ALUop	DataWidth	MemWrite	MemtoReg	Branch	Jump
00100	111	0	1	000	00

00_00_1_1_01_0x_00100_111_0_1_000_00_xxxxx // 0x2b : lb $t = SE (MEM [$s + i]:1)

RegDst: 00, 결과값을 rt에 쓸 것이므로 목적지 레지스터로 rt를 선택한다.

RegDatSel: 00, MEM 결과값을 register file에 쓰도록 선택한다.

RegWrite: 1, register file에 값을 쓴다.

SEUmode: 1, imm값을 sign extension한다.

ALUsrcB: 01, ALU 입력으로 imm값을 사용한다.

ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]=x (shift 수행하지 않음)

ALUop: 00100, a + b

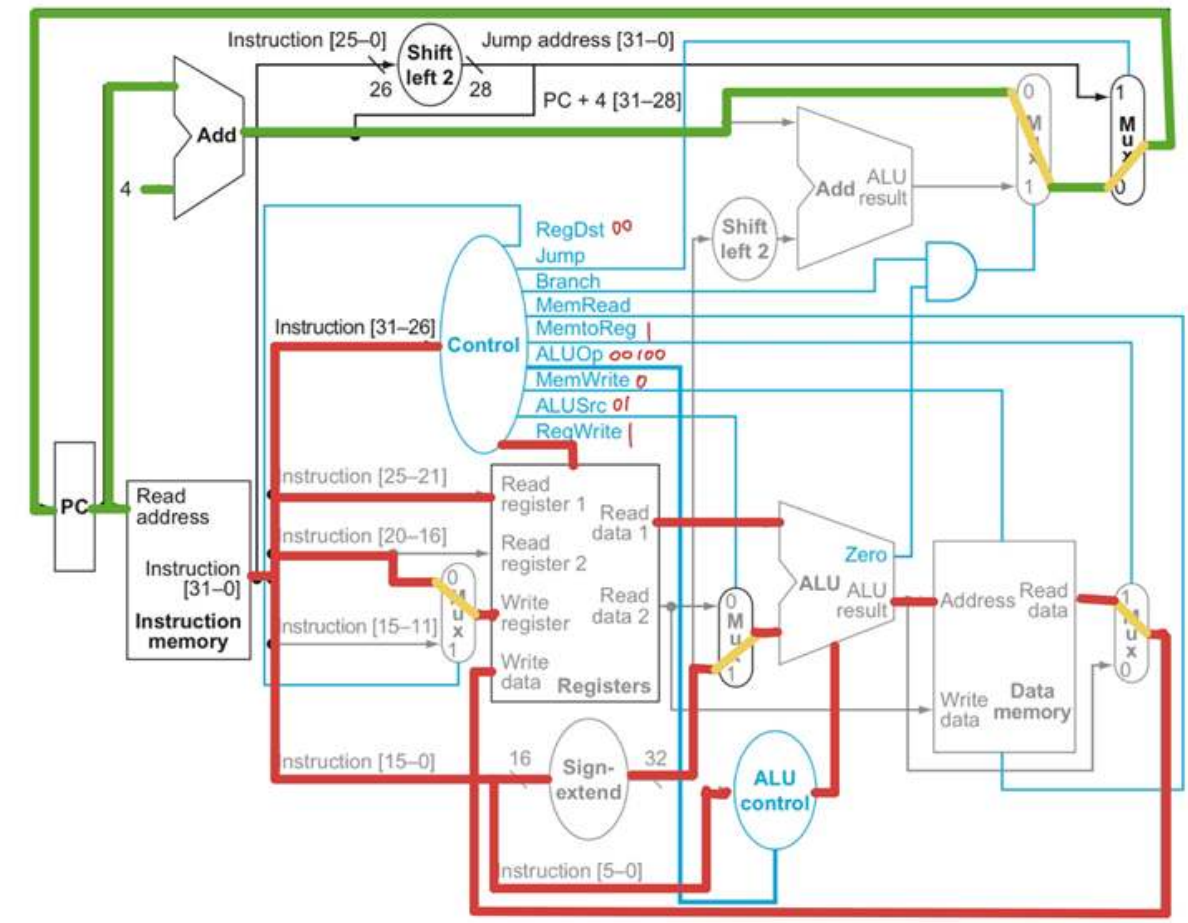
DataWidth: 111, 8-bit Byte with Sign Ext

MemWrite: 0, 메모리에 쓰지 않음.

MemtoReg: 1, register file에 메모리 값을 쓴다.

Branch: 000, Use PC+4

Jump: 00, jump 수행하지 않음.



$s를 읽고 MEM [$s + i]의 1바이트 데이터를 가져와서 sign extend 후 $t에 저장한다. imm값을 sign extend하여 $s와 ALU에서 주소 연산한다. PC=PC+4이다.

8. BNE

Branch Not Equal: $s != $t이면 branch한다, I-type

Syntax: o $s, $t, label

Operation: l if ($s != $t) pc += i << 2

Op	Func	Regimm
000101	xxxxxx	xxxxx

000101_xxxxxx_xxxxx // 0x20 : bne

RegDst	RegDatSel	RegWrite	SEUmode	ALUsrcB	ALUctrl
xx	xx	0	1	00	0x
ALUop	DataWidth	MemWrite	MemtoReg	Branch	Jump
00110	xxx	0	x	101	00

xx_xx_0_1_00_0x_00110_xxx_0_x_101_00_xxxxx // 0x20 : bne if ($s != $t) pc += i << 2

RegDst: xx, register file에 쓰지 않음.

RegDatSel: xx, register file에 쓰지 않음

RegWrite: 0, register file에 값을 쓰지 않음..

SEUmode: 1, imm값을 sign extension한다.

ALUsrcB: 00, ALU 입력으로 register file port B를 사용한다.

ALUctrl: 0c, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]=x (shift 수행하지 않음)

ALUop: 00110, a – b (두 레지스터 값 비교)

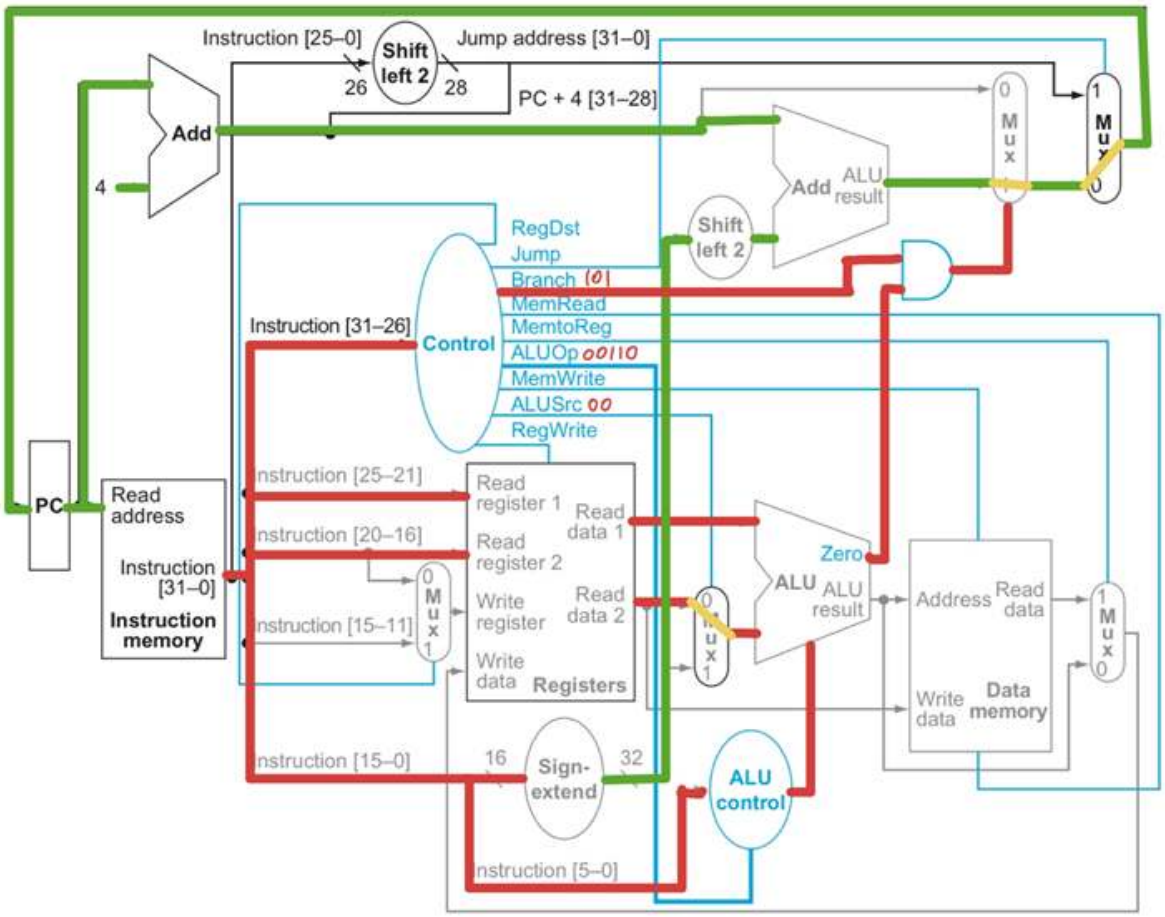
DataWidth: xxx, 메모리 접근하지 않음

MemWrite: 0, 메모리에 쓰지 않음.

MemtoReg: x, register file에 쓰지 않음

Branch: 101, Branch if not zero

Jump: 00, jump 수행하지 않음.



$s, $t값을 읽고, $s != $t 조건이 참이면 PC = PC + i << 2로 branch한다. imm값은 먼저 sign extend를 수행한다. ALU에서 뺄셈 연산을 통해 두 레지스터 값을 비교한 결과인 zero를 사용한다.

9. BGEZ

I-type, RegImm-type: opcode=000001 고정하고, rt(=regimm)로 명령어를 구분한다. $s >=0이면 branch한다.

Syntax: r $s, label

Operation: if ($s >= 0) pc += i << 2

Op	Func	Regimm
000001	xxxxxx	00001

000001_xxxxxx_00001 // 0x1c : bgez

RegDst	RegDatSel	RegWrite	SEUmode	ALUsrcB	ALUctrl
Xx	Xx	0	1	10	0x
ALUop	DataWidth	MemWrite	MemtoReg	Branch	Jump
10000	Xxx	0	X	011	00

xx_xx_0_1_10_0x_10000_xxx_0_x_011_00_xxxxx // 0x1c : bgez if ($s >= 0) pc += i << 2

RegDst: xx, register file에 쓰지 않음.

RegDatSel: xx, register file에 쓰지 않음

RegWrite: 0, register file에 값을 쓰지 않음..

SEUmode: 1, imm값을 sign extension한다.

ALUsrcB: 10, ALU 입력으로 0을 사용한다.

ALUctrl: 0x, ALUctrl[1]=0 (normal ALU input), ALUctrl[0]=x (shift 수행하지 않음)

ALUop: 10000, set less than

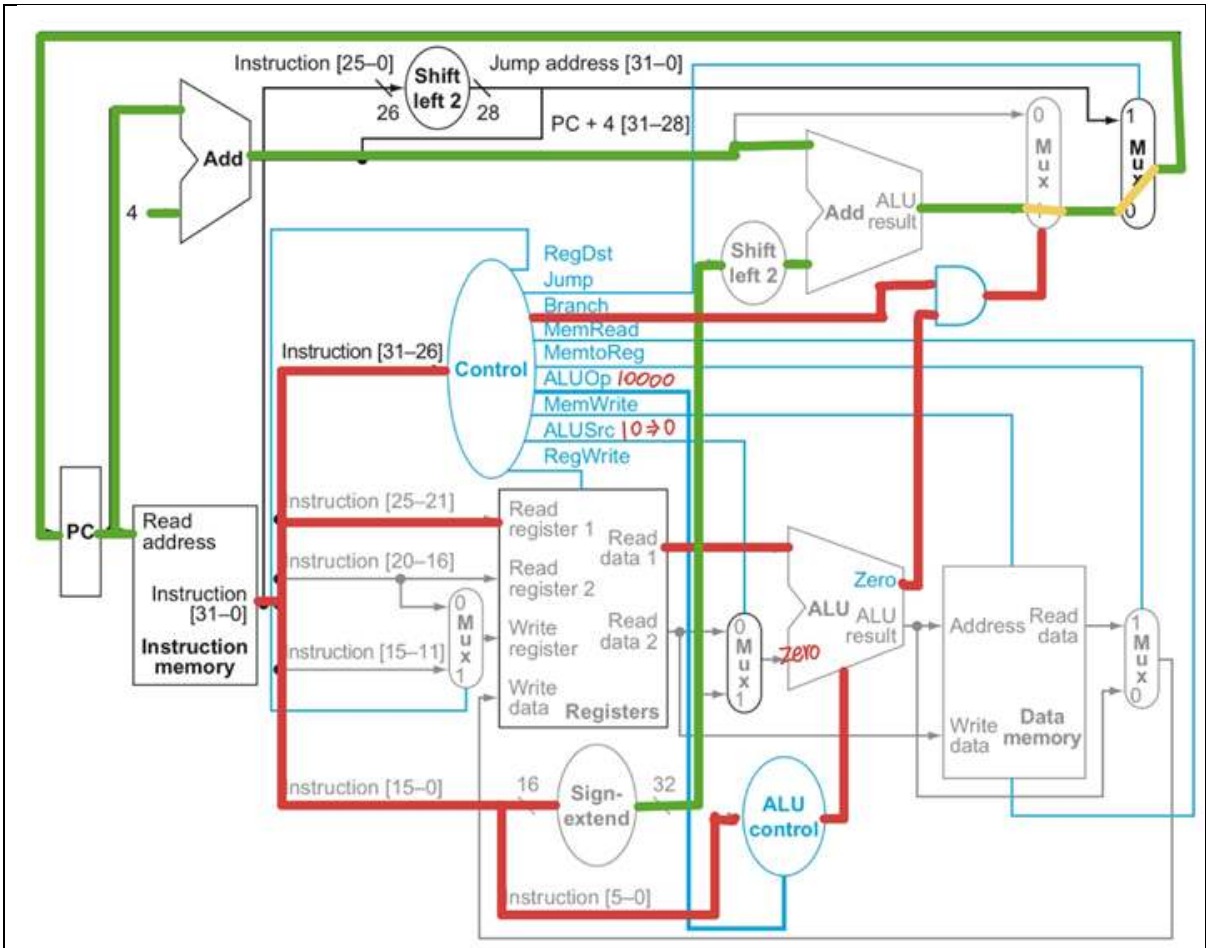
DataWidth: xxx, 메모리 접근하지 않음

MemWrite: 0, 메모리에 쓰지 않음.

MemtoReg: x, register file에 쓰지 않음

Branch: 011, Unconditional Branch to PC+imm16 (ALU에서 먼저 조건 검사 수행하므로)

Jump: 00, jump 수행하지 않음.



$s를 읽고, $s >= 0 조건이 참이면 PC = PC + i << 2로 branch한다. imm값은 먼저 sign extend 한다. ALU 입력으로 0을 사용하고 $s과의 set less than 연산을 통해 조건 검사를 수행한다. 결과값인 zero를 사용한다.

10. JALR

Jump And Link Register: PC를 $31에 저장하고 $s 레지스터가 가리키는 주소로 점프한다, R-type

Syntax: f labelR

Operation: $31 = pc; pc = $s

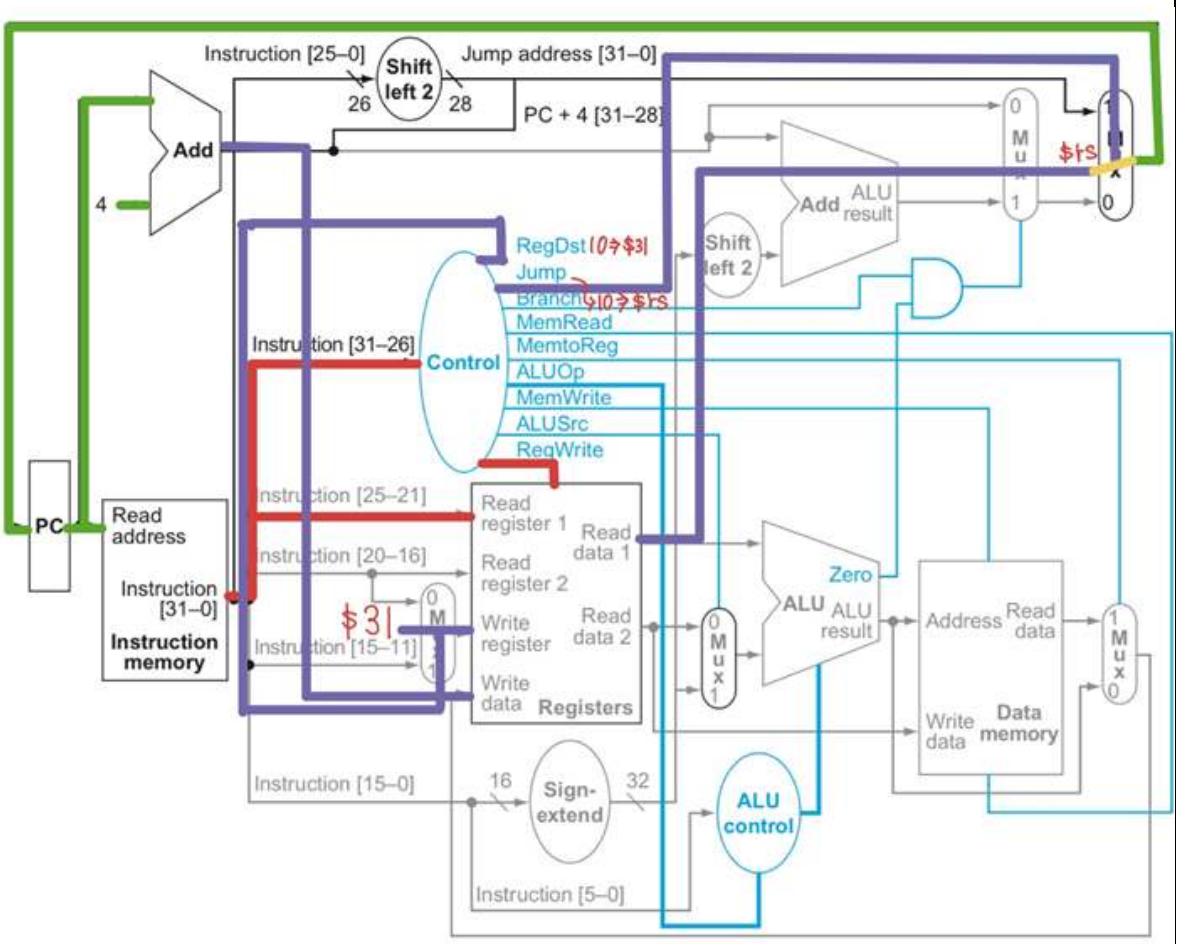
Op	Func	Regimm
000000	001001	xxxxx

000000_001001_xxxxx // 0x07 : jalr

RegDst	RegDatSel	RegWrite	SEUmode	ALUsrcB	ALUctrl
10	11	1	x	xx	xx
ALUop	DataWidth	MemWrite	MemtoReg	Branch	Jump
xxxxx	xxx	0	x	xxx	10

10_11_1_x_xx_xx_xxxxx_xxx_0_x_xxx_10_xxxxx // 0x07 : jalr $31 = pc; pc = $s

RegDst: 10, PC를 $31에 쓸 것이므로 목적지 레지스터로 $31를 선택한다.
RegDatSel: 11, PC를 register file에 쓴다.
RegWrite: 1, register file에 값을 쓴다.
SEUmode: x, imm값을 extension할 필요가 없다.
ALUsrcB: xx, ALU 연산 수행하지 않음.
ALUctrl: xx, ALU 연산 수행하지 않음.
ALUop: xxxxx, ALU 연산 수행하지 않음.
DataWidth: xxx, 메모리 접근하지 않음.
MemWrite: 0, 메모리에 쓰지 않음.
MemtoReg: x, PC를 바로 레지스터 파일에 씀. (ALU 결과값과 메모리값을 쓰지 않으므로 사용x)
Branch: xxx, branch 수행하지 않음.
Jump: 10, Use $rs

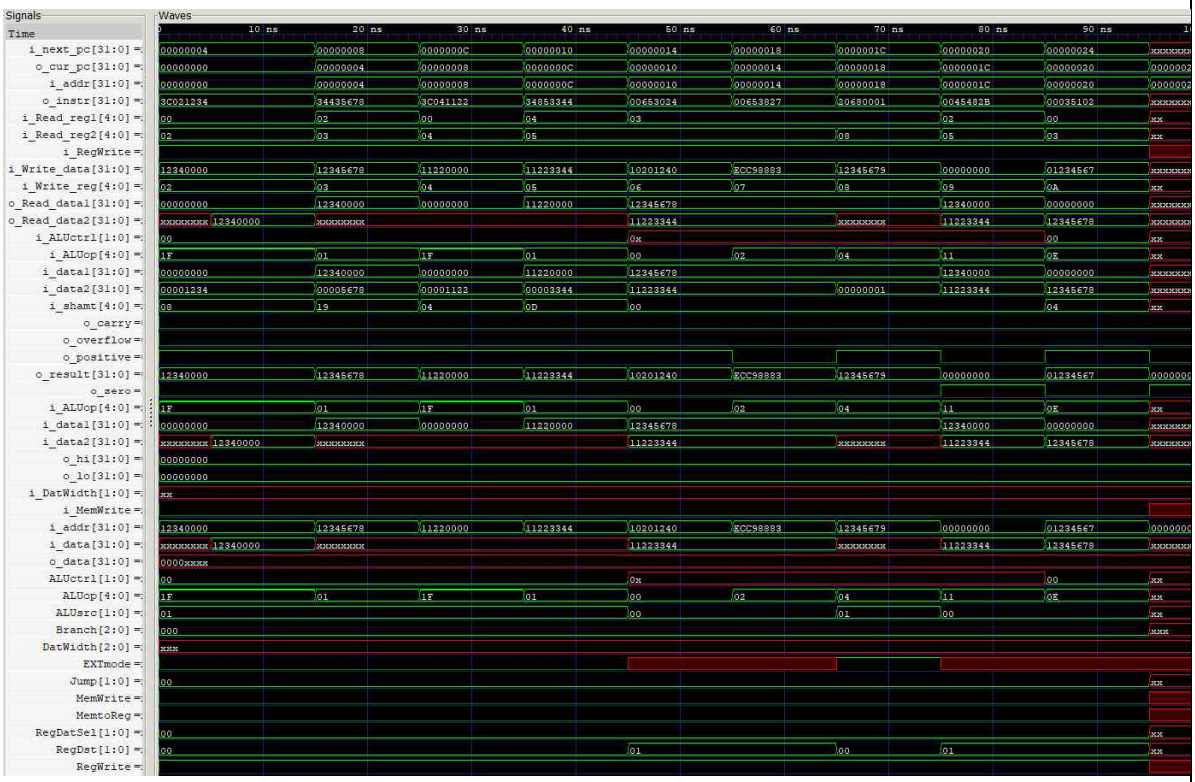


PC를 $31에 저장하고 $s가 가리키는 주소로 점프한다

주어진 회로 그림의 control signal에는 표시되지 않았지만, RegDatSel을 11로 설정하여 레지스터 파일에 쓸 값이 PC로 선택되었음이 동작하고 있다. 해당 동작에 맞추어, 좌측 위에서 계산된 PC+4가 레지스터 파일의 Write data로 들어가는 선을 추가하였다. 또한 RegDst가 write register로 $31를 선택하는 것을 표현했다.

마찬가지로 control signal인 Jump를 10으로 설정하여 $rs를 사용하는 동작이 수행되고 있다. 따라서 Jump 신호가 다음 PC를 선택하는 mux로 들어가는 선택선과, 읽은 $rs의 값을 해당 mux의 입력선으로 사용하기 위해 해당 선들을 추가로 그려서 위의 동작을 표현했다.

<Simulation>



먼저 기본 ALU 연산 동작을 하는 명령어들의 시뮬레이션을 진행했다. 레지스터 연산을 하는 R-type 명령어 and, nor, sltu, srl과 imm값을 사용하여 연산하는 I-type 명령어 addi의 검증을 수행했다. 다음은 사용한 명령어와 그 동작이다.

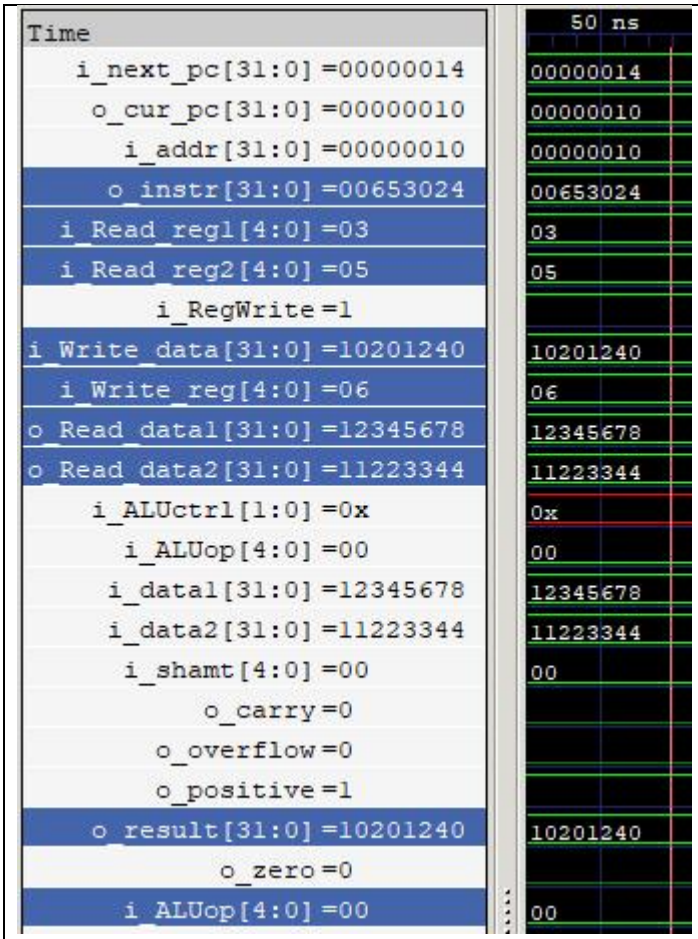
000000_00011_00101_00110_00000_100100 // and $6, $3, $5

000000_00011_00101_00111_00000_100111 // nor $7, $3, $5

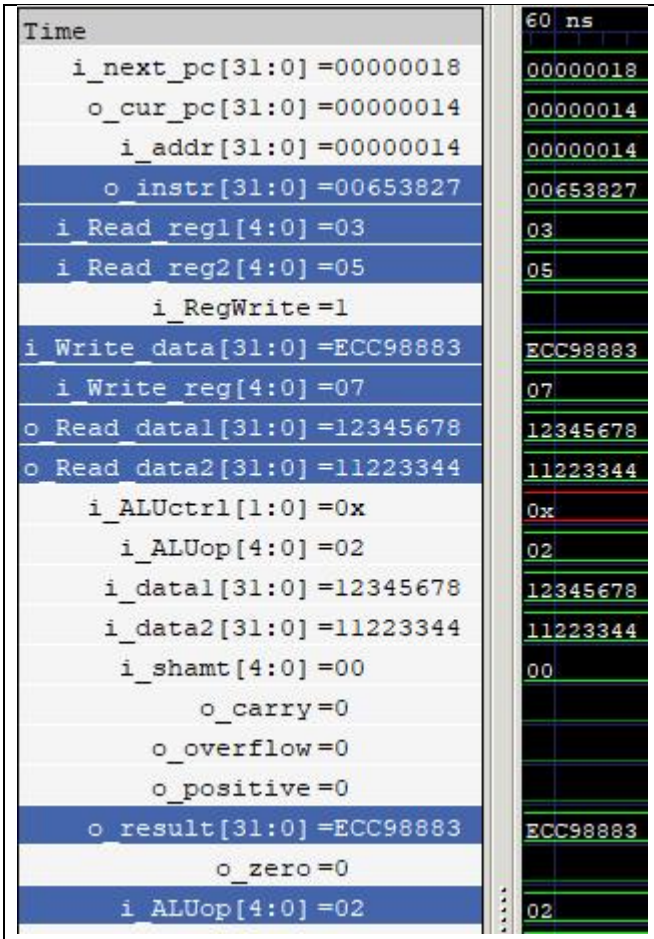
001000_00011_01000_0000000000000001 // addi $8, $3, 1

000000_00010_00101_01001_00000_101011 // sltu $9, $2, $5

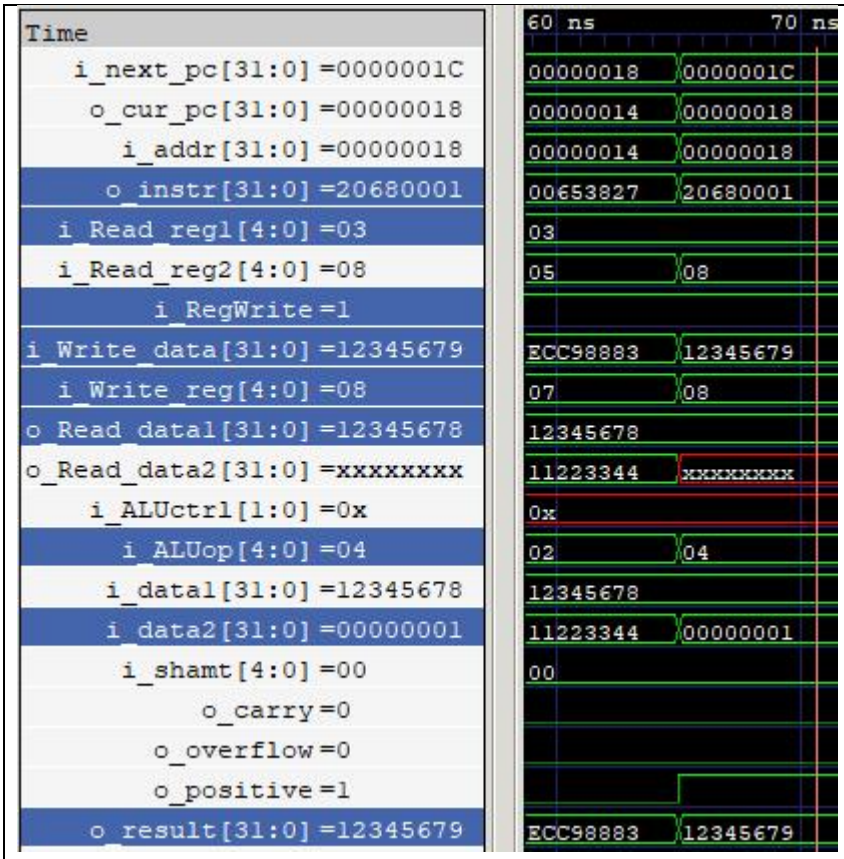
000000_00000_00011_01010_00100_000010 // srl $10, $3, 4



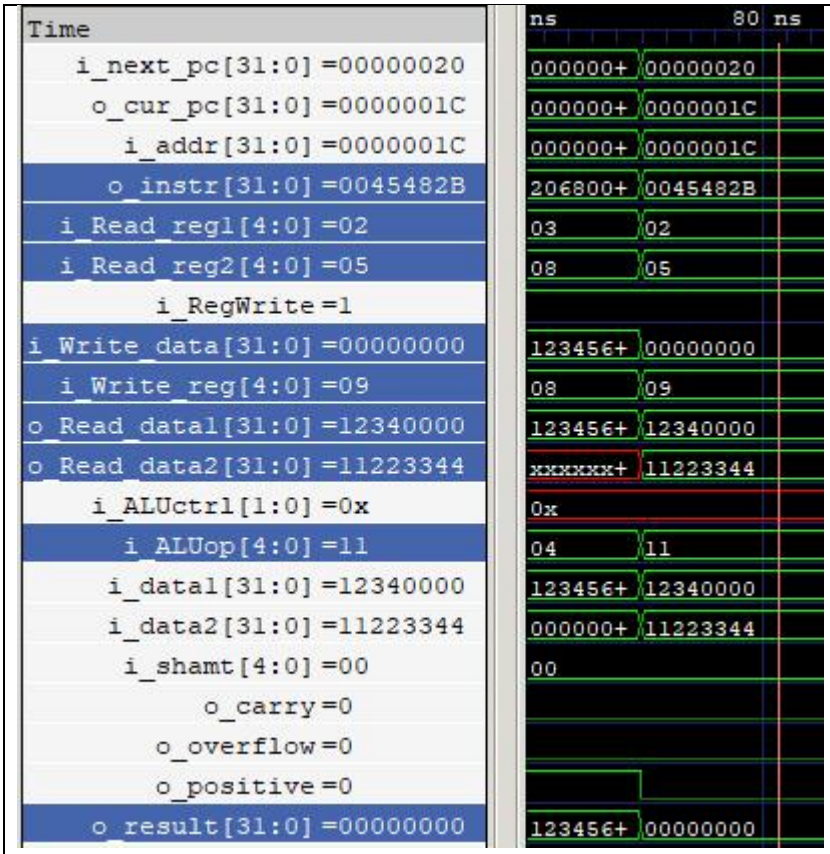
먼저 add 명령어를 읽고 수행 결과, $3, $5 두 레지스터를 읽고 $6에 $3 & $5 연산 결과인 0x12345678 & 0x11223344 = 0x10201240의 값이 저장되었다. PC = PC + 4이다.



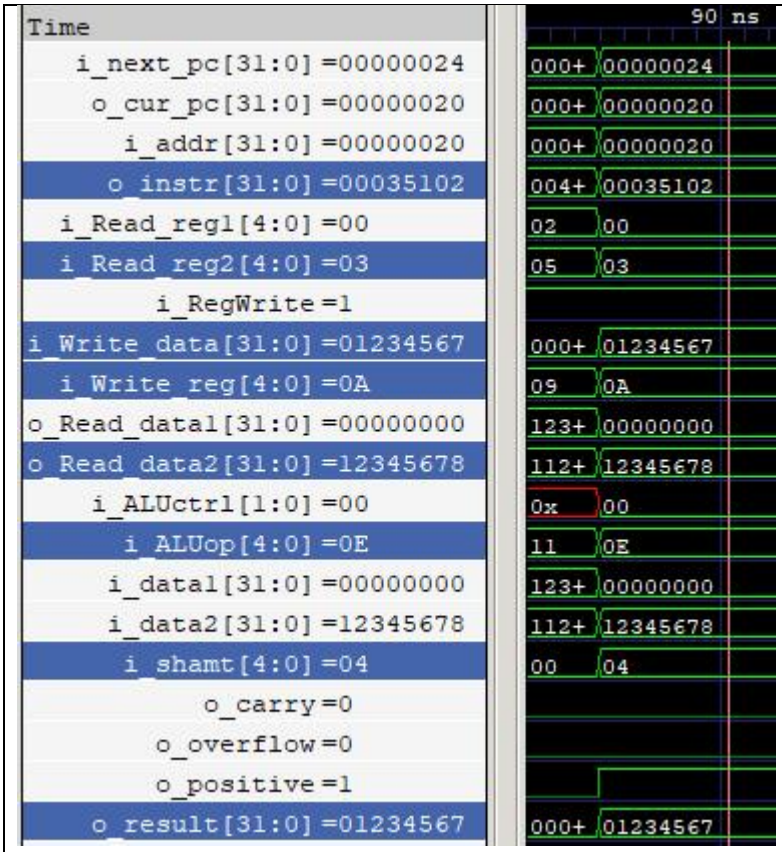
다음으로 nor 명령어를 읽고 수행 결과, $3, $5 두 레지스터를 읽고 $7에 ~($3 | $5) 연산 결과인 ~(0x1336777C) = 0xECC98883의 값이 저장되었다. PC = PC + 4이다.



addi 명령어를 읽고 수행한 결과, $3과 imm값인 1을 읽은 후 $8에 0x12345678 + 1 연산 결과인 0x12345679가 저장되었다. PC = PC + 4이다.



sltu 명령어를 읽고 수행 결과, $2와 $5를 읽고 비교하여 0x12340000 > 0x11223344이므로 $9에 unsigned 기준 연산 결과인 0이 저장되었다. PC = PC + 4이다.



srl 명령어를 읽고 연산 결과, $3와 shamt 값을 읽은 후 $10에 $3를 shamt만큼 shift한 $3 >> 4 연산 결과인 0x012345670이 저장되었다. PC = PC + 4이다.

모든 명령어 수행 후 저장된 reg_dump.txt 파일이다.

reg_dump.txt - Windows 메모장

파일(F) 편집(E) 서식(O) 보기(V) 도움말(H)

```
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 : 00010001_00100010_00000000_00000000 : 11220000
00000005 : 00010001_00100010_00110011_01000100 : 11223344
00000006 : 00010000_00100000_00010010_01000000 : 10201240
00000007 : 11101100_11001001_10001000_10000011 : ecc98883
00000008 : 00010010_00110100_01010110_01111001 : 12345679
00000009 : 00000000_00000000_00000000_00000000 : 00000000
0000000a : 00000001_00100011_01000101_01100111 : 01234567
```

다음으로 메모리 접근을 수행하는 두 명령어 sh, lb의 시뮬레이션을 수행하였다.

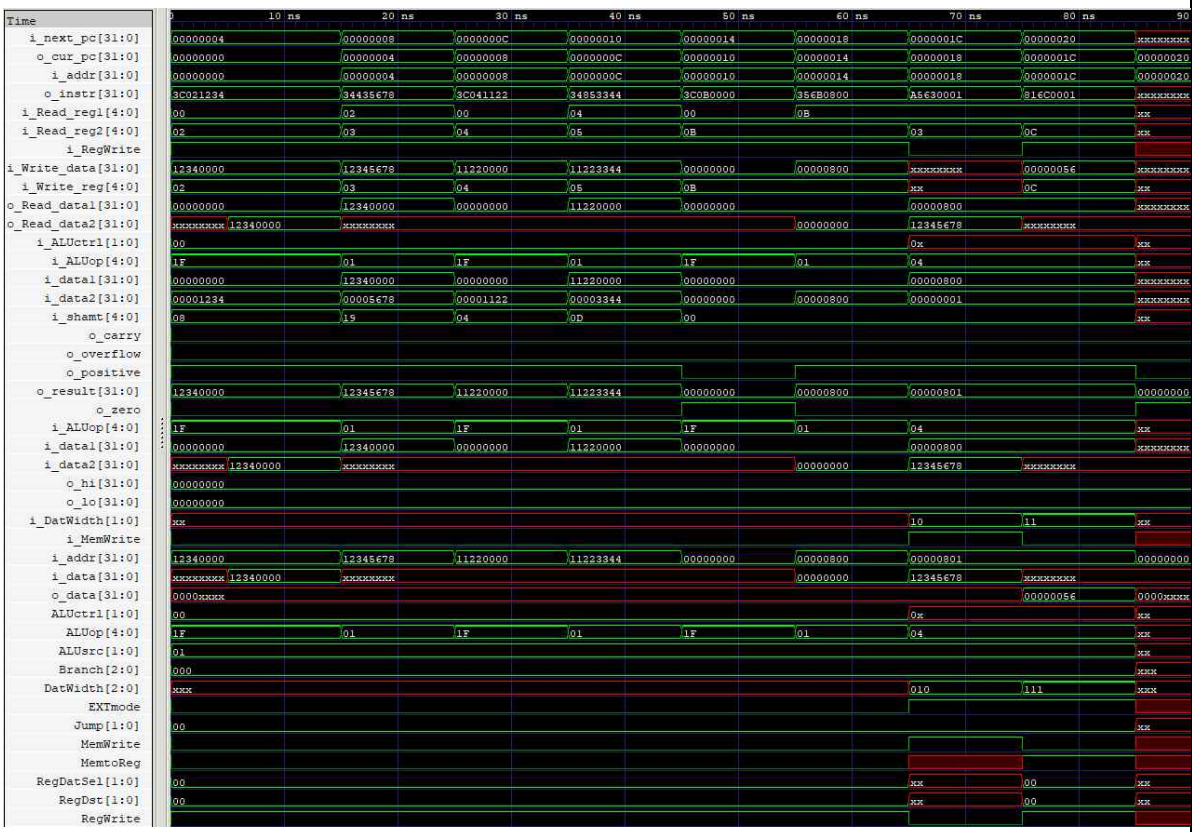
001111_00000_01011_0000000000000000 // lui $11, 0x0000

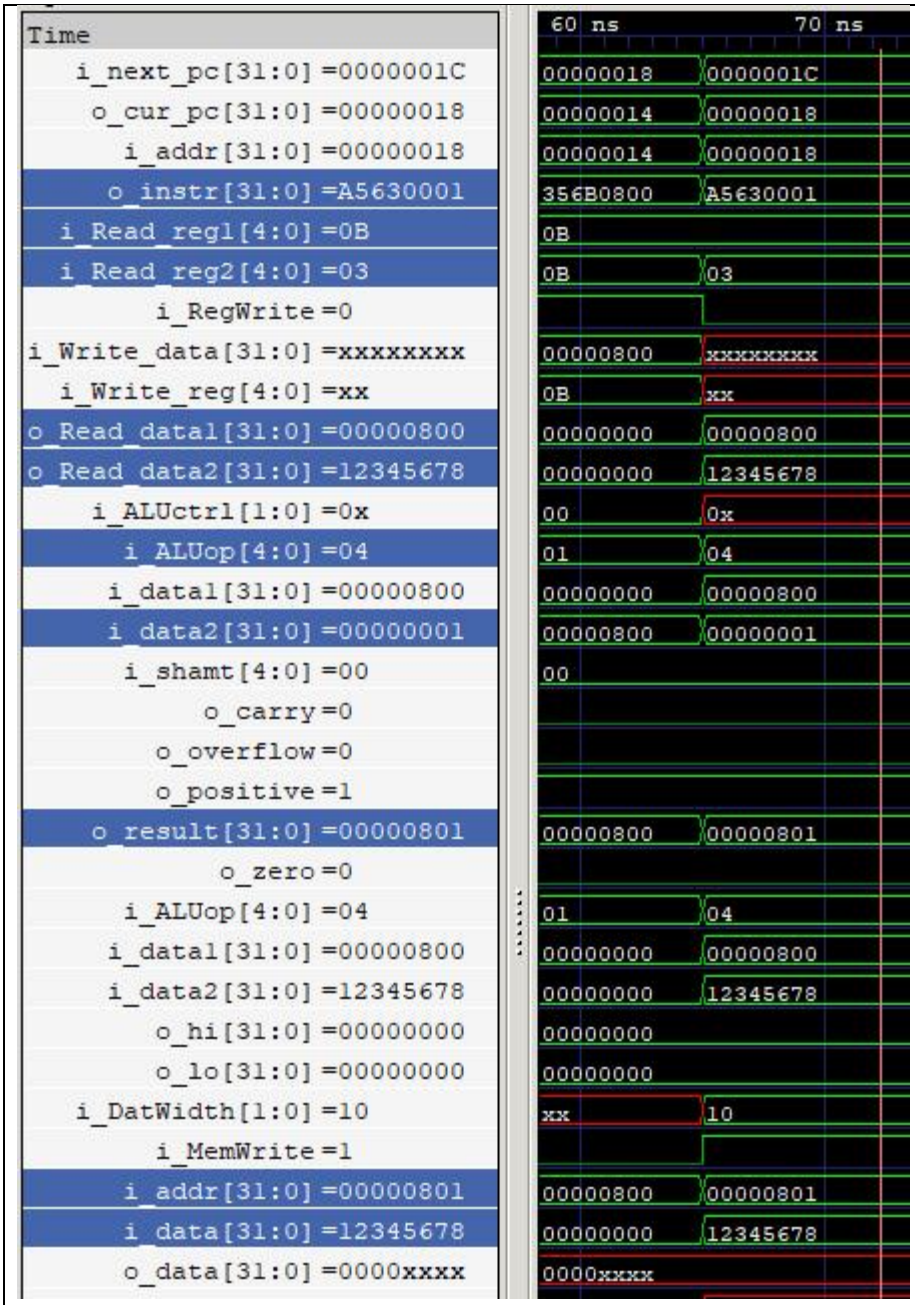
001101_01011_01011_0000100000000000 // ori $11, $11, 0x0800

101001_01011_00011_0000000000000001 // sh $3, 1($11)

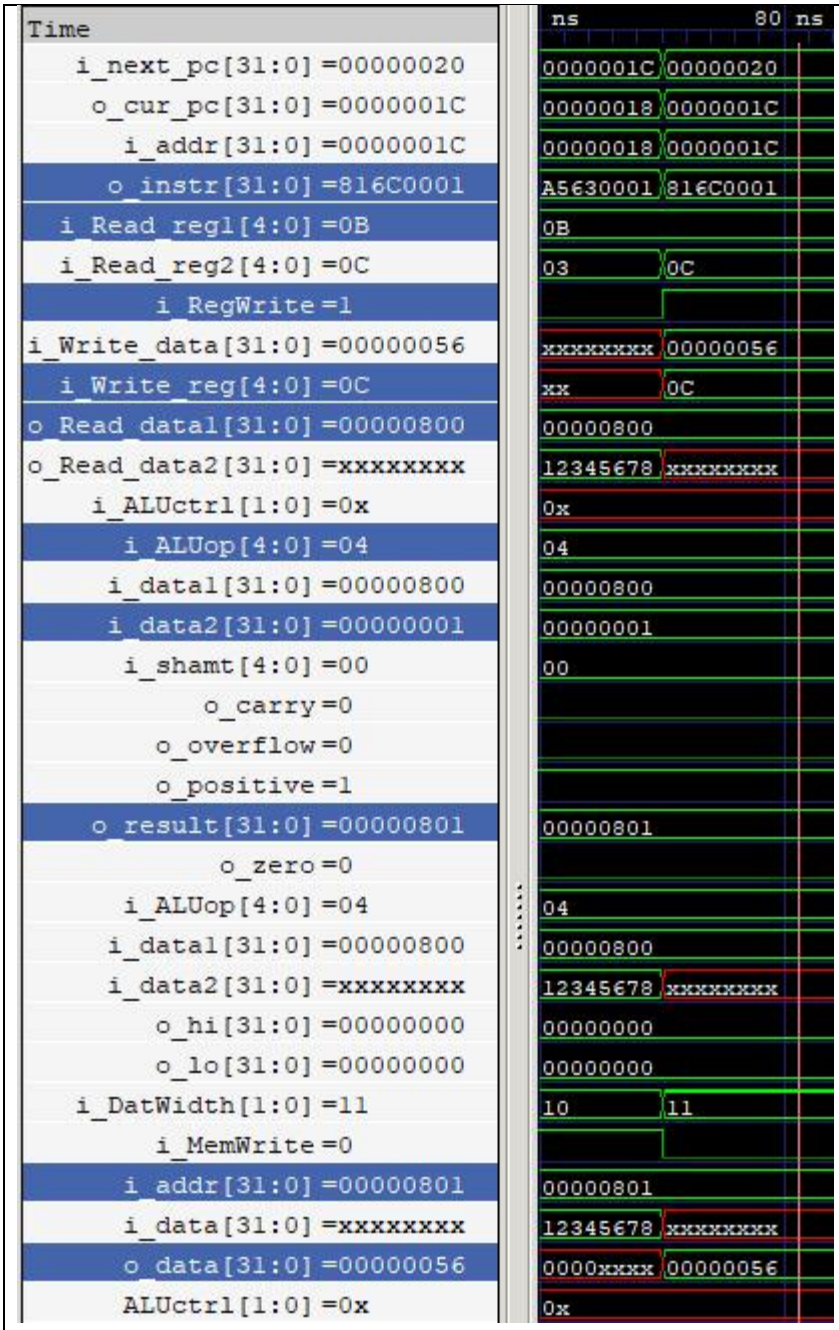
100000_01011_01100_0000000000000001 // lb $12, 1($11)

먼저 위 명령어들 수행을 통해 $11에 0x00000800 값이 저장될 것이다.



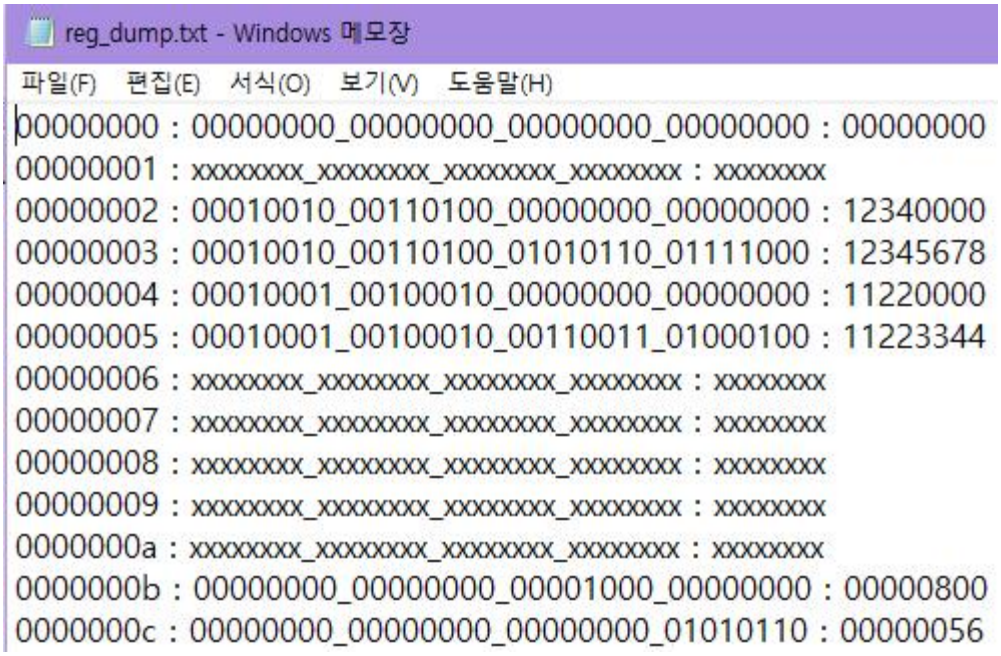


sh 명령어를 읽고 수행한 결과, MEM[$11 + 1]에 $3의 하위 2바이트 값을 쓸 것이다. 0x0800에 imm 값인 1이 더해진다. $3=0x12345678 중 리틀 앤디언 방식에 따라 메모리 주소 0x0800에는 0x78이, 0x0801에는 0x56이 저장되었다. RegWrite는 0, MemWrite는 1이다. PC = PC + 4이다.

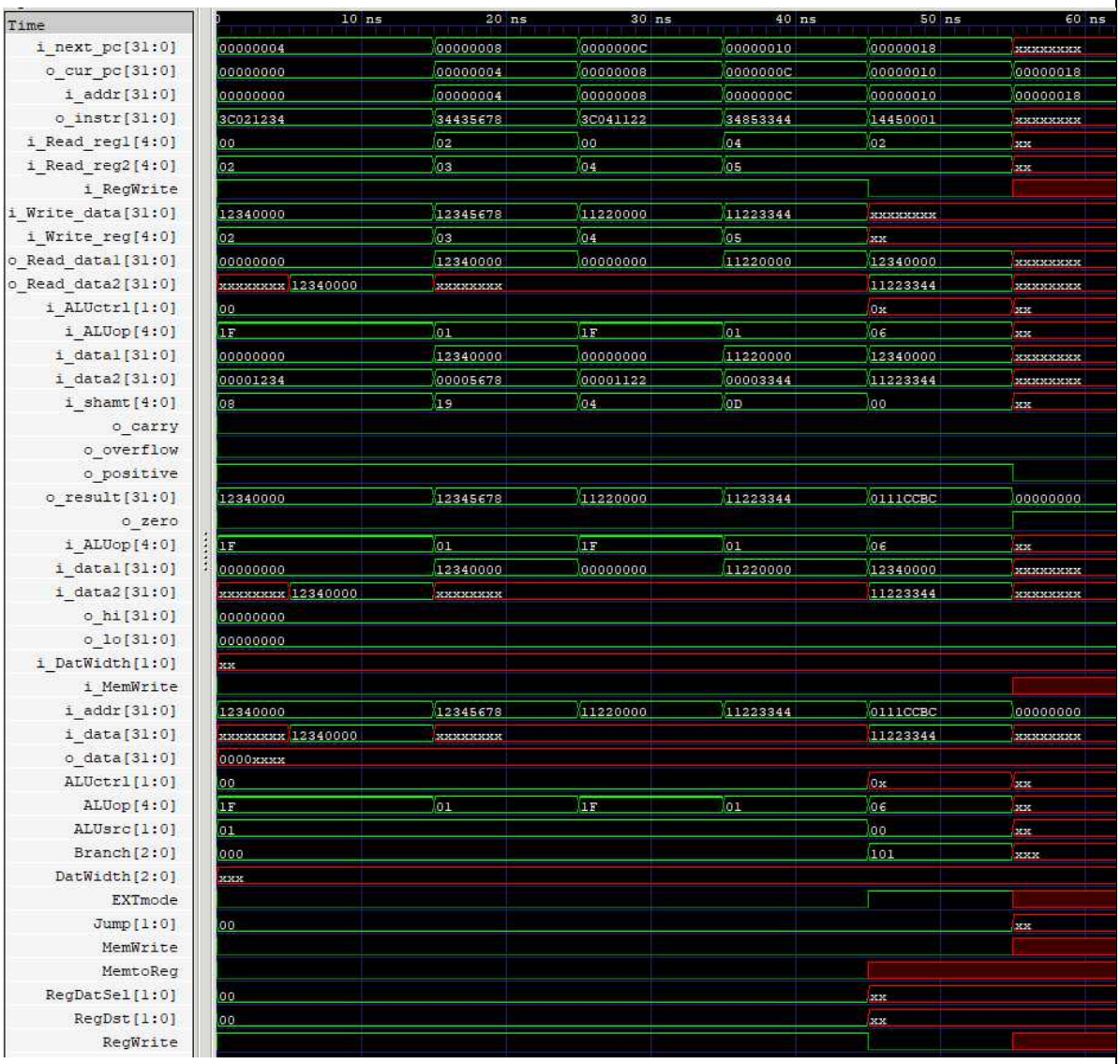


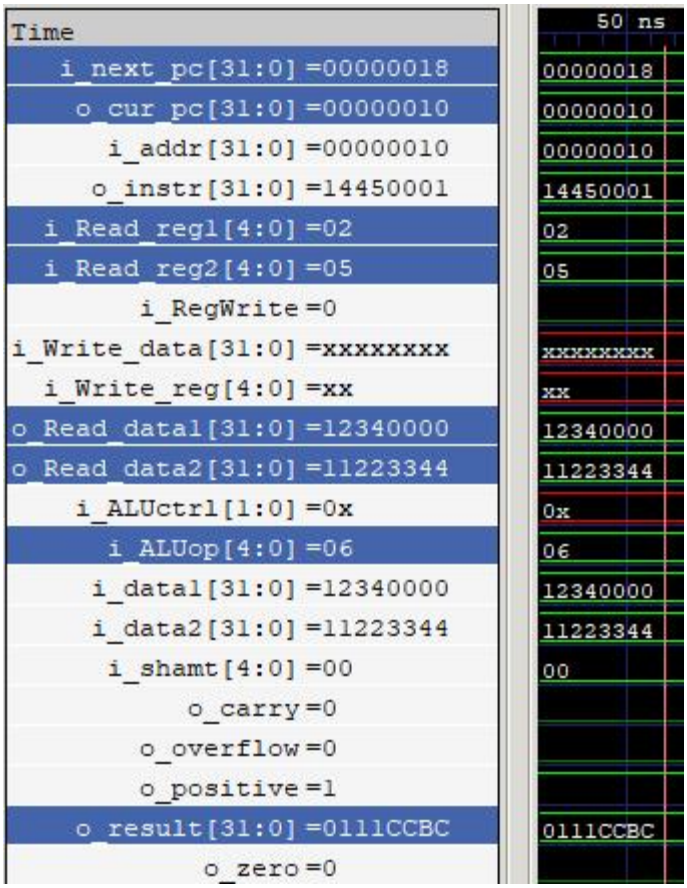
다음으로 lb 명령어를 읽고 수행 결과, MEM[$11 + 1]의 1바이트 데이터를 가져와 sign extend 후 $12에 저장할 것이다. 0x0800에 imm 값인 1이 더해진 결과인 0x0801에는 이전 sh 명령어 수행 결과 0x56이 저장되어 있다. write register인 $12에 0x56이 저장되었다. 따라서 RegWrite는 1, MemWrite는 0이다. PC = PC + 4이다.

모든 명령어 수행 후 저장된 reg_dump.txt 파일이다.



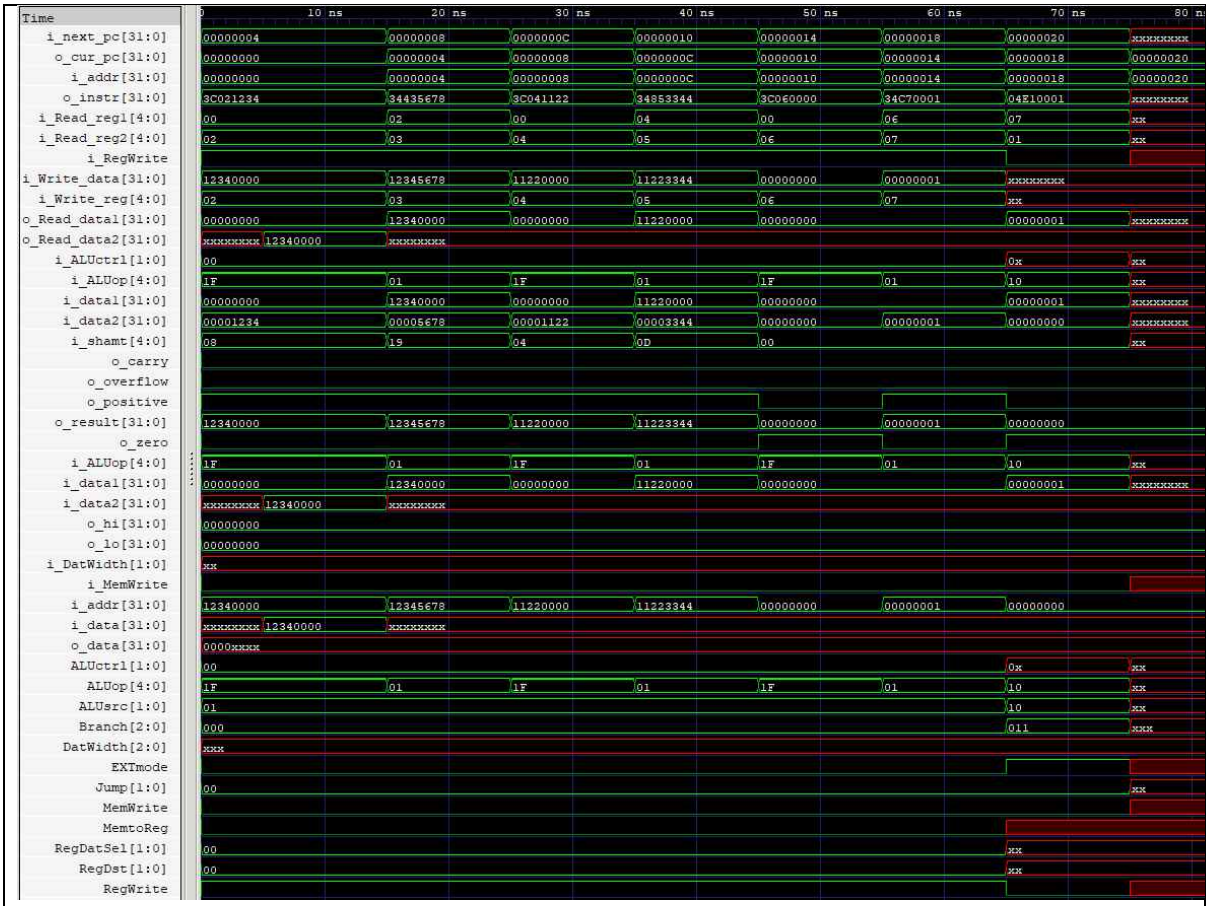
다음으로 branch와 jump 명령어들인 bne, bgez, jalr을 각각 검증했다.

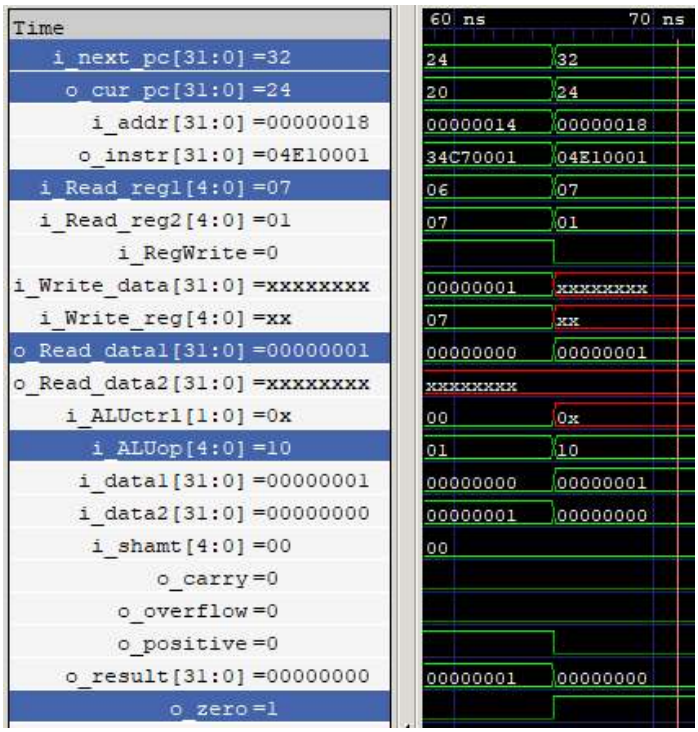




000101_00010_00101_0000000000000001 // bne $2, $5, 1

먼저 bne 명령어를 읽고 수행한 결과, $2와 $5를 읽었고 ALU에서 a-b 연산을 수행한 결과 0이 아니므로 branch조건인 0x12340000 =/ 0x11223344를 만족한다. 계산한 pc 주소는 pc = pc+4+(imm<<2)이므로, next_pc 값이 0x0010 + 0x0004 + 0x0004 = 0x0018인 것을 확인했다.



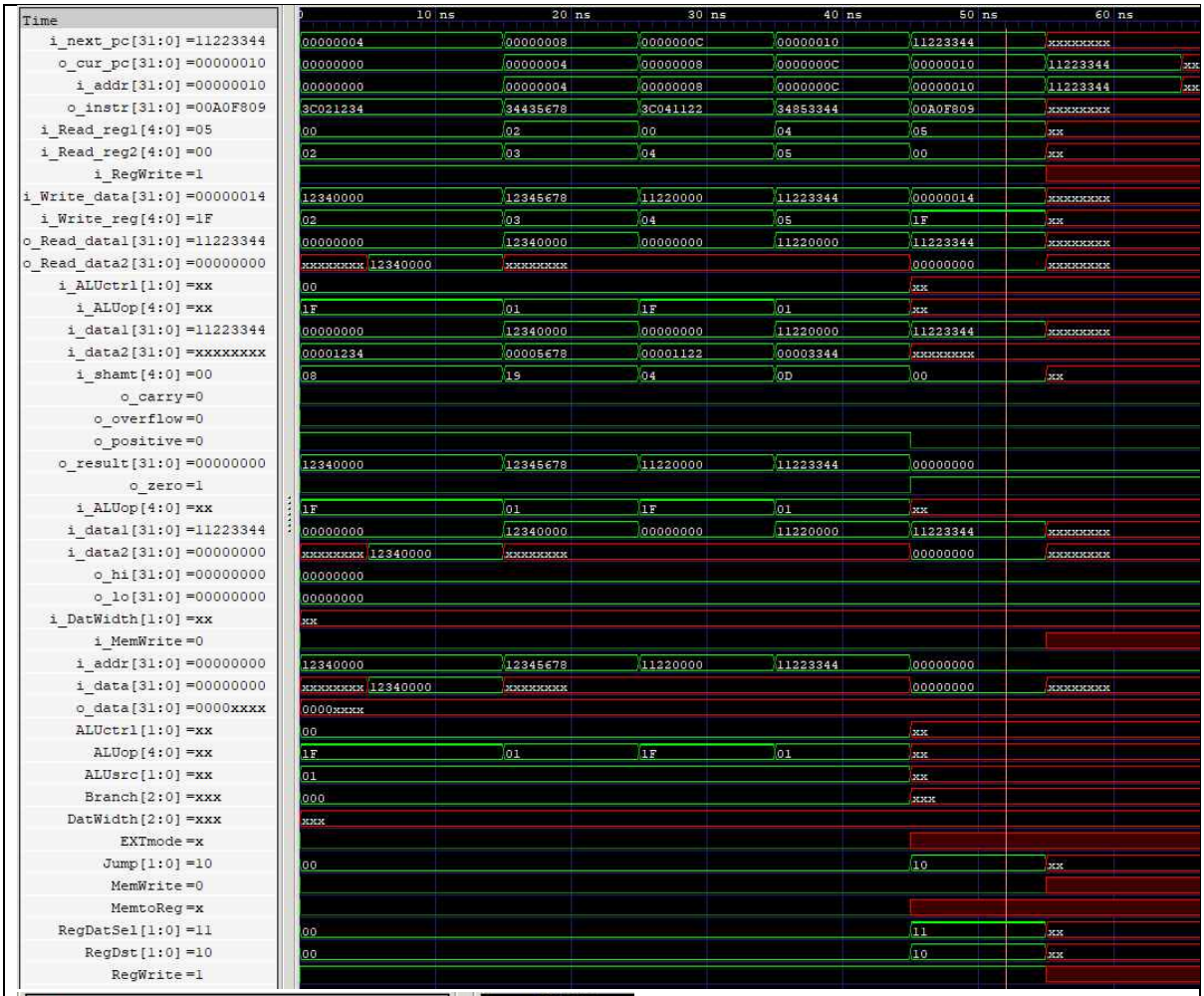


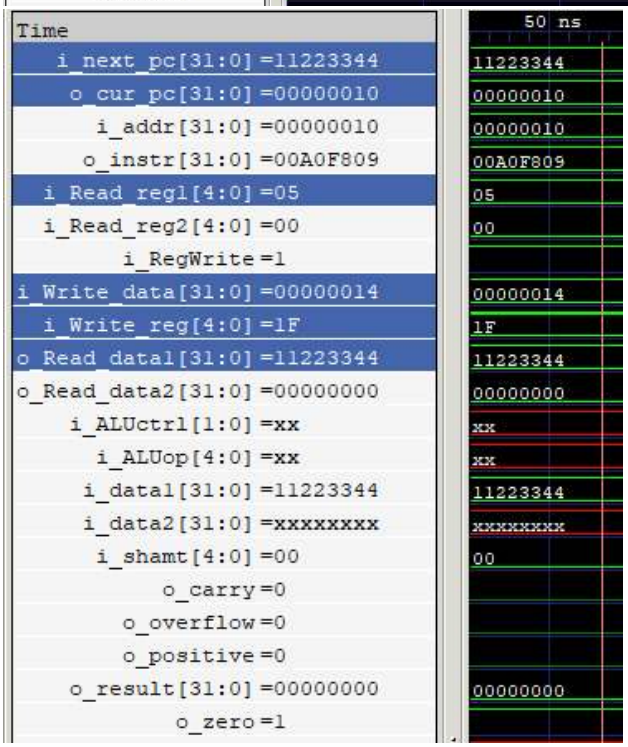
001111_00000_00110_0000000000000000 // lui $6, 0x0000

001101_00110_00111_0000000000000001 // ori $7, $6, 0x0001

000001_00111_00001_0000000000000001 // bgez $7, 1

다음으로 bgez 명령어 검증을 위해 일단 lui와 ori 명령어를 통해 $7에 0x0001을 저장했다. bgez 명령어를 읽고 수행한 결과, $7를 읽고 0과 set less than ALU 연산한 결과 조건인 0x00000001 >= 0을 만족하므로 branch한다. imm값은 1이고, 계산한 pc 주소는 pc = pc+4+(imm<<2)이므로, next_pc 값이 decimal로 표현했을 때 24 + 4 + 4 = 32인 것을 확인했다.

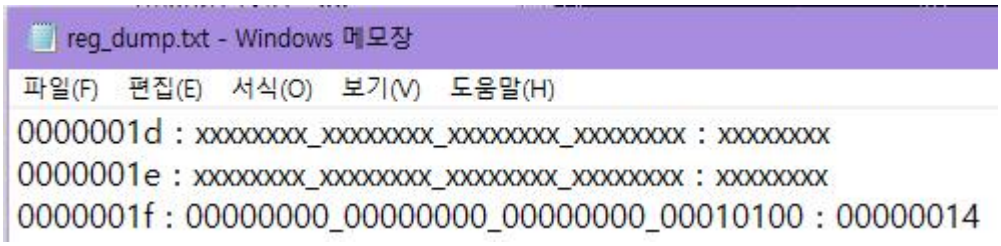

Time
i_next_pc[31:0] =11223344
o_cur_pc[31:0] =00000010
i_addr[31:0] =00000010
o_instr[31:0] =00A0F809
i_Read_reg1[4:0] =05
i_Read_reg2[4:0] =00
i_RegWrite =1
i_Write_data[31:0] =00000014
i_Write_reg[4:0] =1F
o_Read_data1[31:0] =11223344
o_Read_data2[31:0] =00000000
i_ALUctrl[1:0] =xx
i_ALUop[4:0] =xx
i_data1[31:0] =11223344
i_data2[31:0] =xxxxxxxx
i_shamt[4:0] =00
o_carry =0
o_overflow =0
o_positive =0
o_result[31:0] =00000000
o_zero =1
i_ALUop[4:0] =xx
i_data1[31:0] =11223344
i_data2[31:0] =00000000
o_hi[31:0] =00000000
o_lo[31:0] =00000000
i_DatWidth[1:0] =xx
i_MemWrite =0
i_addr[31:0] =00000000
i_data[31:0] =00000000
o_data[31:0] =0000xxxx
ALUctrl[1:0] =xx
ALUop[4:0] =xx
ALUsrc[1:0] =xx
Branch[2:0] =xxx
DatWidth[2:0] =xxx
EXTmode =x
Jump[1:0] =10
MemWrite =0
MemtoReg =x
RegDatSel[1:0] =11
RegDst[1:0] =10
RegWrite =1
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns


Time
i_next_pc[31:0] =11223344
o_cur_pc[31:0] =00000010
i_addr[31:0] =00000010
o_instr[31:0] =00A0F809
i_Read_reg1[4:0] =05
i_Read_reg2[4:0] =00
i_RegWrite =1
i_Write_data[31:0] =00000014
i_Write_reg[4:0] =1F
o_Read_data1[31:0] =11223344
o_Read_data2[31:0] =00000000
i_ALUctrl[1:0] =xx
i_ALUop[4:0] =xx
i_data1[31:0] =11223344
i_data2[31:0] =xxxxxxxx
i_shamt[4:0] =00
o_carry =0
o_overflow =0
o_positive =0
o_result[31:0] =00000000
o_zero =1
50 ns

000000_00101_00000_11111_00000_001001 // jalr $31, $5

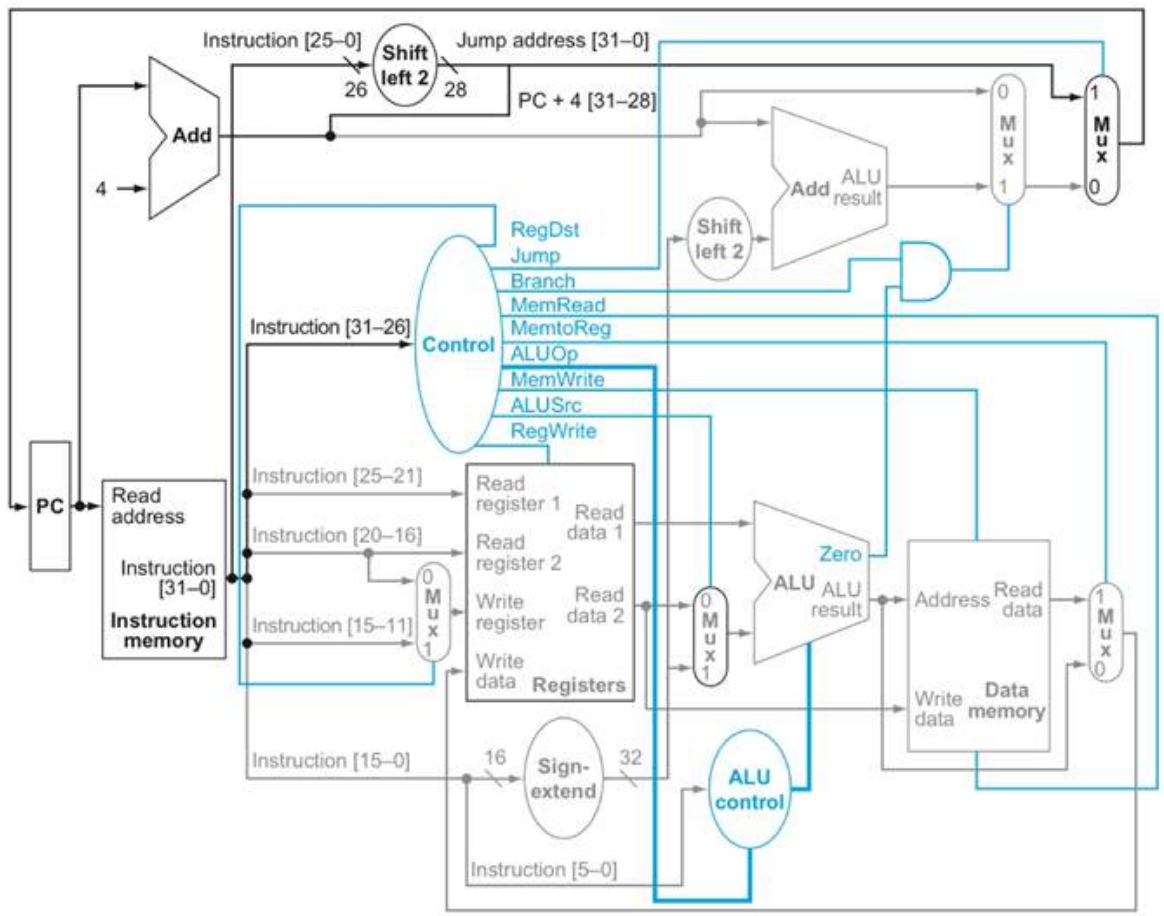
jalr 명령어를 읽고 수행 결과, $31에는 PC+4=0x0014가 저장되고, PC에는 레지스터에서 읽은 $5=0x11223344가 저장된다.

jalr 명령어 수행 후 저장된 reg_dump.txt 파일이다.



이렇게 10개의 명령어 시뮬레이션을 마쳤고, M_TEXT_SEG.txt의 각 명령어 필드를 고려하여 동작 예상을 주석과 함께 작성하였다. 시뮬레이션 결과는 예상 결과와 같았다.

<구현한 Single Cycle CPU 블록도>



Program counter, Instruction memory, Register file, Sign Extension Unit, Arithmetic Logic Unit, Data memory, Control unit로 구성된 블록도이다. 이외에도 추가 control signals, Multiplier Logic Unit, PLA Unit 등과 함께 Single Cycle CPU가 구현되어 있다.

<문제점 및 개선점 기술>

이번 프로젝트에서, Single Cycle MIPS 구조에 10개의 명령어를 추가 구현하면서 기존 PLA_OR 및 PLA_AND 구조를 활용하여 명령어들의 새로운 control signals 조합을 추가하는 과정을 포함했다. ALU를 통한 base + offset 주소 계산 방식이나 명령어/데이터 메모리 및 레지스터 파일 구조를 유지하여 구현하였다.

특히 ALUctrl 신호는, ALU의 연산 동작만을 나타내는 ALUop 신호에 부가적인 ALU 제어를 가능하게 했다. srl 명령어에서, ALU 입력으로 shamt를 사용해 shift 연산하겠다는 ALUctrl 신호를 설정하여 구현했다.

bgez 명령어 구현에서, $s가 0보다 크거나 같아야 한다는 조건을 검사하는 데에 어떤 ALUop를 사용해야 할지 문제점이 발생하였다. ALUsrcB 신호에 zero가 있는 것을 확인하고, ALU에서 zero와 set less than 연산을 하여 결과에 적절히 not gate를 이용하여 조건 연산이 가능할 것이라는 점을 파악했다. 시뮬레이션 결과 기대한 명령어의 작동을 확인하고 문제점을 해결하였다.

jalr는 $31에 PC+4를 저장한 후 $s로 점프해야 하는데, ALU를 통해 PC+4를 계산해야 할지 혼동되는 문제점이 있었다. 본 구현에서는 기존 Fetch 단계에서 사용하는 PC+4 adder 출력을 그대로 활용함으로써, ALU 연산 없이 RegDatSel 신호 설정을 통해 PC를 바로 레지스터에 저장하도록 새로운 방법을 채택하여 개선하였다.

이번 구현에서 새로운 방법을 더 제시해보자면, ALUctl과 ALUop 신호 같은 경우 둘 다 ALU를 제어하는 역할을 수행하고 있다. 명령어 중 R-type는 funct 필드로, I-type은 opcode로 연산을 결정하는데 이를 활용하여 ALUctrl 신호 동작에 더하여 opcode, func, ALUop의 적절한 조합으로 단일 ALU control signal을 새롭게 설정하면 독자적으로 디코딩되어 ALU를 제어하는 신호를 만들 수 있을 것 같다. 연산 구분이 명확해지고 명령어 확장 시 코드 복잡도를 낮출 수 있다. 알아보기에는 더욱 어렵고 복잡해진다는 단점도 존재한다.

메모리 접근 시 사용하는 DatWidth 신호 같은 경우, Byte, Halfword, Word, Sign/Zero Extend 구분을 모두 표현하다 보니 복잡했다. byte, word 등 데이터의 크기를 다루는 신호와, extend 형식을 나타내는 신호로 분리하면 load와 store 관련 명령어 확장 시 구현에 용이할 것 같다.

또한 현재 구현은 PLA 기반으로 동작하는데, 명령어 수가 더 증가하고 조합 논리가 점점 복잡해진다면 마이크로프로그래밍 방식으로 나아갈 수 있다.

3. 고찰

이번 프로젝트에서 명령어 10개의 PLA 구조를 구현했다. 일단 10개 명령어의 타입, 필드 구조와 같은 기본적인 내용과 SCPU MIPS 회로도에서 각 명령어의 datapath를 이해하는 과정이 필요했다. 세부적으로 ALU 연산들에 관한 내용이나, jump와 branch 시 imm값을 사용한 주소 계산과 이용되는 extend unit에 대한 이해도 이루어졌다. 프로젝트 0을 수행한 이후에도 회로에 대한 내용은 어려웠는데, 이번에는 직접 datapath를 그려보는 과제를 통해 명령어들의 수행 단계와 흐름에 대해서는 이제 정말 잘 알 것 같다.

구현이 힘들었던 명령어로는 먼저 srl 명령어가 있다. 이론 시간에는 분명 교수님이 shamt 필드는 생각하지 말라고 했는데, shamt를 사용하는 명령어를 구현하라 해서 처음엔 놀랐지만 Signal Definition을 잘 읽어보니 shamt를 사용할 수 있게 하는 ALUctrl이 있어서 사용했다. jump 수행하는 명령어는 j-type만 있는 줄 알았는데... jalr 명령어는 r-type인데도 jump를 수행하고 register write도 수행하는 명령어라서 매우 생소했다. 회로도에서 그려저 있지 않은 control signal을 표현하느라 힘들었지만 이 과정에서 datapath에 대한 이해가 많이 는 것 같다. 자세한 내용은 datapath 그린 부분이랑 바로 위 페이지에 써놨다. I-type 중 RegImm-type이라는 새로운 명령어 타입도 알아간다. MIPS에 내가 모르는 정말 많은 명령어가 남아 있다는 사실을 깨달았다. SCPU와 MIPS에 대한 전반적인 나의 지식을 확장할 수 있는 정말 유익한 프로젝트가 된 것 같다.

그리고 waveform을 이용해 명령어 동작 확인도 수행했다. 이 과정에서 cmd애서 gtkwave를 실행하는데 오류 메시지가 많이 떴다. 실행은 그래도 정상적으로 되고 파형도 만들어지길래 동작이 이상한 것은 내가 PLA를 잘못 구현해서인 줄 알았다. 아무리 PLA를 고쳐봐도 파형은 그대로였고 그제서야 오류 메시지를 다 천천히 읽어 보니 모듈을 c드라이브에서 찾지 못했다는 내용이었다. 아니나 다를까 나는 iverilog 및 gtkwave를 g드라이브에 설치했다. 그런데 테스트 벤치는 c드라이브에서 모듈들을 찾고 있었다. 재설치를 하고 gtkwave도 실습 자료에서 사용한 구버전으로 바꿨더니 잘 돌아갔다. 오류 메시지가 뜨면 제대로 읽어봐야겠다는 교훈을 얻었다. 오랜만에 디논 수업 때처럼 파형을 해석해보니 재밌었다.