컴퓨터구조

Project #1

MIPS Single Cycle CPU Implementation

Class : **월 3 수 4 분반**

Professor : **이성원 교수님**

Student ID : **2020202031**

Name : **김재현**

1. Introduction

We look at what might be thought of as the simplest possible implementation of our MIPS subset. This simple implementation covers load word(lw), store word(sw), or immediate (ori), and jump(j). Constant manipulation instruction (lui) and user defined instructions (llo, lhi) are also implemented. The following figure shows an example of the similar single-cycle CPU implementations.

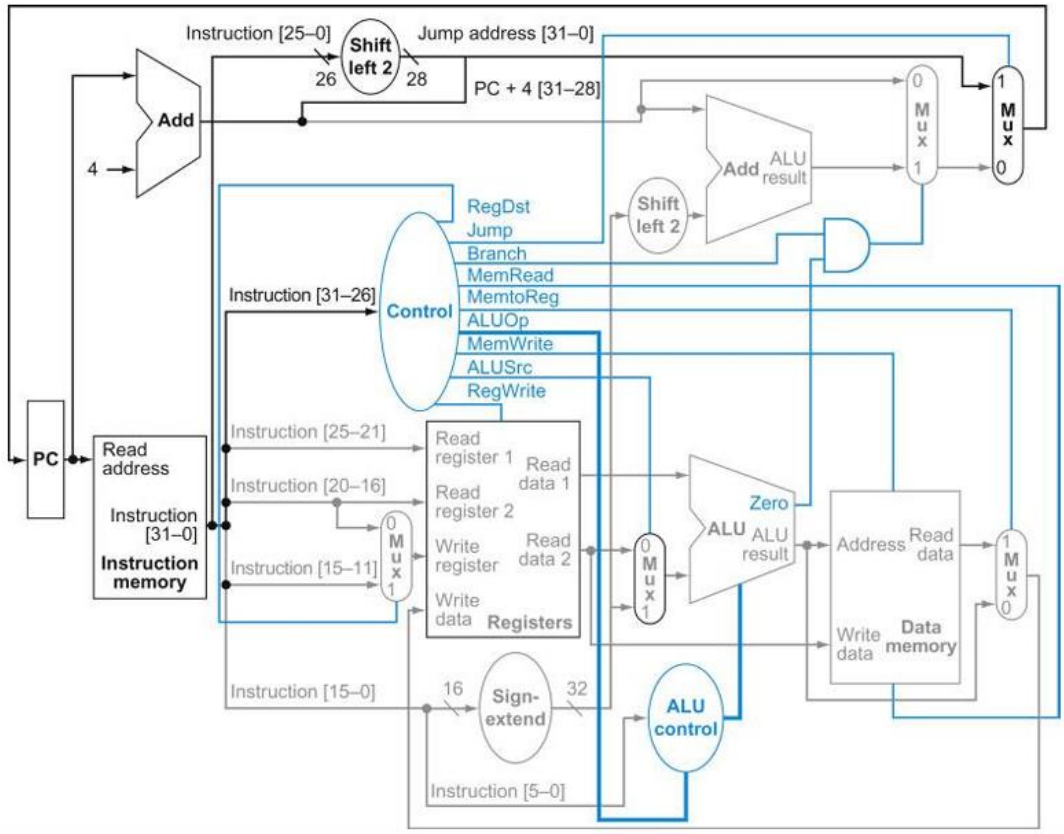


Figure 1 - The single cycle CPU datapath and control path

Table 1 – Instance name of top module

Instance name	Description
SC_CPU	Top module. Single-Cycle CPU
U0_PC	Program counter
U1_IM	Instruction memory
U2_RF	Register file
U3_SEU	Sign Extension Unit
U4_ALU	Arithmetic Logic Unit
U5_MULT	Multiplier Logic Unit
U6_DM	Data memory
U7_CTRL	Control unit -MainControl, MyControl

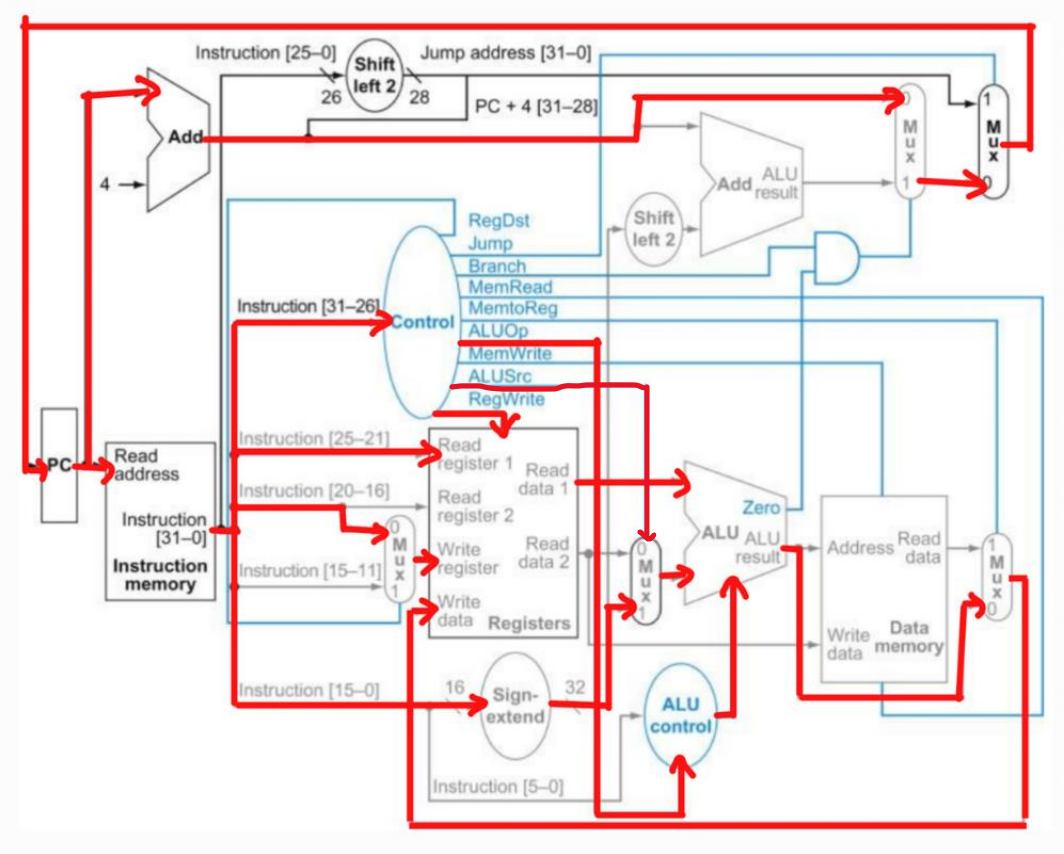
2. Assignment

MIPS Instructions **LW, SW, ORI, ADD, SUB, J, LUI, BREAK, LLO, LHI** are already implemented. You have to add the following MIPS Instructions:

XORI, SLT, SUBU, SRA, MULTU, MFLO, SB, LHU, BLEZ, JR

1) XORI

instruction	opcode/function	syntax	operation
xori	001110	o $t, $s, i	$t =$s ^ ZE(i)



00_00_1_0_01_0x_00011_xxx_0_0_000_00_xxxxx

xori 는 rs 와 ZE(imm)를 xor 한 값을 rt 에 저장하는 연산을 수행하는 I-type 명령어입니다.

Control unit 에서 op 값 001110 을 읽어 다양한 신호들을 각 unit 에 전달합니다. xori 는 register 에서 값을 읽어, ALU 에서 연산을 수행하고, 결과 값을 register 에 저장합니다.

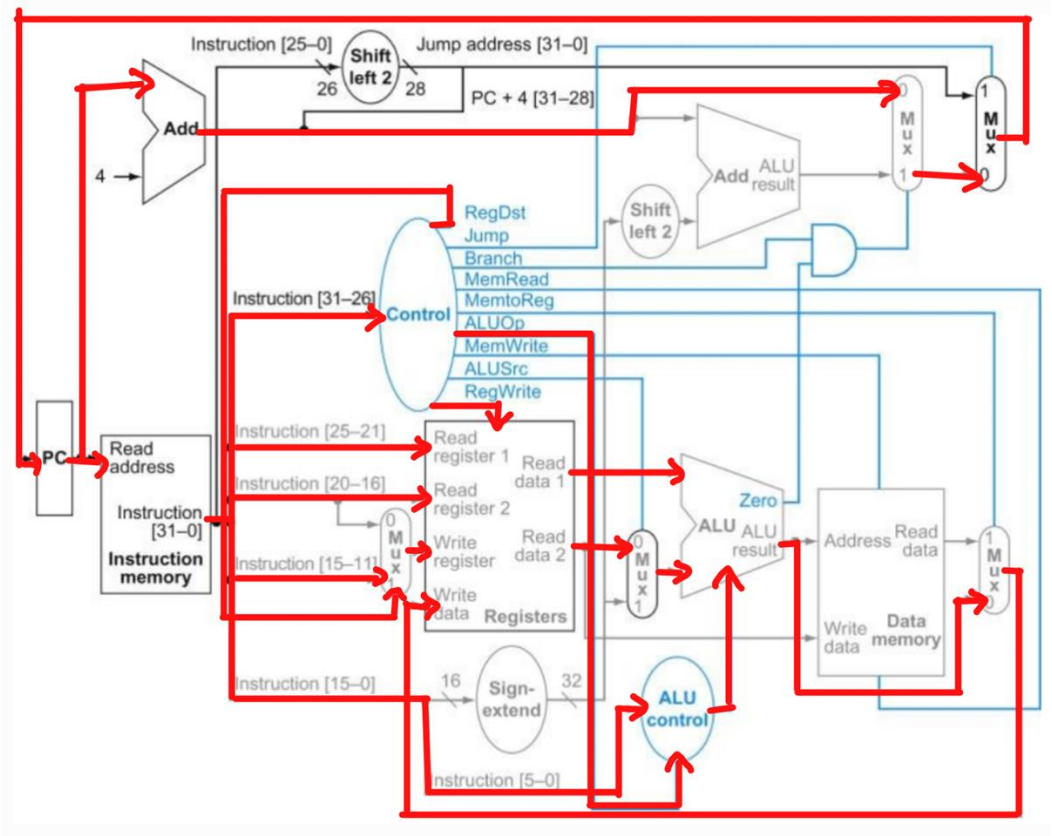
ALU 에서 zero-extend 된 16bits imm 값과 rs 값이 xor 연산돼야합니다. 이 때 imm 값이 zero-extend 이므로 SEUmode 0, ALU 연산에 imm 값이 사용되므로 ALUSrcB 01, ALUctrl 0x, ALUop 00011 가 됩니다.

Memory 에는 접근하지 않으므로 DataWidth xxx, MemWrite 0 입니다. ALU 출력 값이 rt 에 저장돼야 하므로 MemtoReg 0, RegDst 00, RegDatSel 00, RegWrite 1 입니다.

xori 는 branch 나 jump 명령어가 아니므로 Branch 000, Jump 00 입니다. 따라서 pc 현재 값이 Adder 에서 +4 되어 다시 pc 의 입력으로 들어갑니다.

2) SLT

instruction	opcode/function	syntax	operation
slt	101010	f $d, $s, $t	$d = ($s < $t)



01_00_1_x_00_0x_10000_xxx_0_0_000_00_xxxxx

slt 는 rs < rt 면 1 을, rs >= rt 면 0 을 rd 에 저장하는 연산을 수행하는 R-type 명령어입니다.

Control unit 에서 op 값 101010 을 읽어 다양한 신호들을 각 unit 에 전달합니다. slt 은 register 에서 값을 읽어, ALU 에서 연산을 수행하고, 결과 값을 register 에 저장합니다.

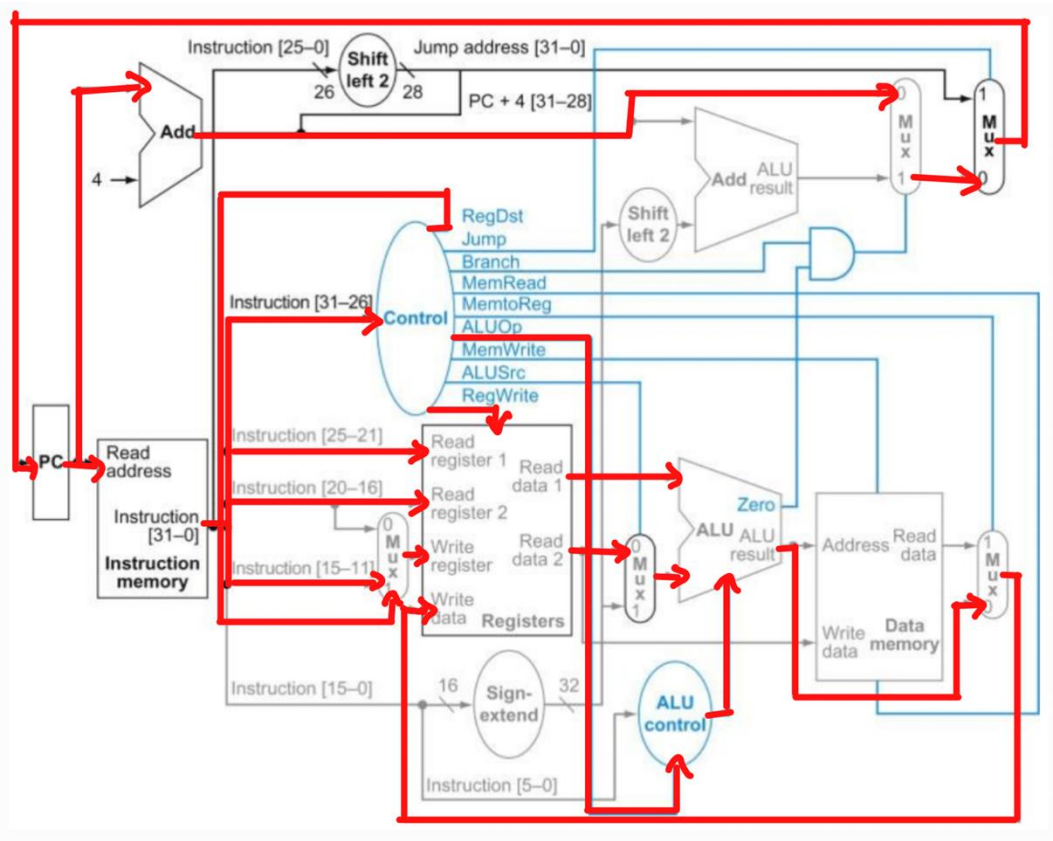
ALU 에서 rs 값과 rt 값이 set less than 연산돼야 합니다. 이 때 imm 값은 사용되지 않으므로 SEUmode x, ALU 연산에 rt 값이 사용되므로 ALUSrcB 00, ALUctrl 0x, ALUop 10000 가 됩니다.

Memory 에는 접근하지 않으므로 DataWidth xxx, MemWrite 0 입니다. ALU 출력 값이 rd 에 저장돼야 하므로 MemtoReg 0, RegDst 01, RegDatSel 00, RegWrite 1 입니다.

slt 은 branch 나 jump 명령어가 아니므로 Branch 000, Jump 00 입니다. 따라서 pc 현재 값이 Adder 에서 +4 되어 다시 pc 의 입력으로 들어갑니다.

3) SUBU

instruction	opcode/function	syntax	operation
subu	100011	f $d, $s, $t	$d = $s - $t



01_00_1_x_00_0x_00111_xxx_0_0_000_00_xxxxx

subu 는 rs 와 rt 를 -연산한 값을 rd 에 저장하는 연산을 수행하는 R-type 명령어입니다.

Control unit 에서 op 값 100011 을 읽어 다양한 신호들을 각 unit 에 전달합니다. subu 는 register 에서 값을 읽어, ALU 에서 연산을 수행하고, 결과 값을 register 에 저장합니다.

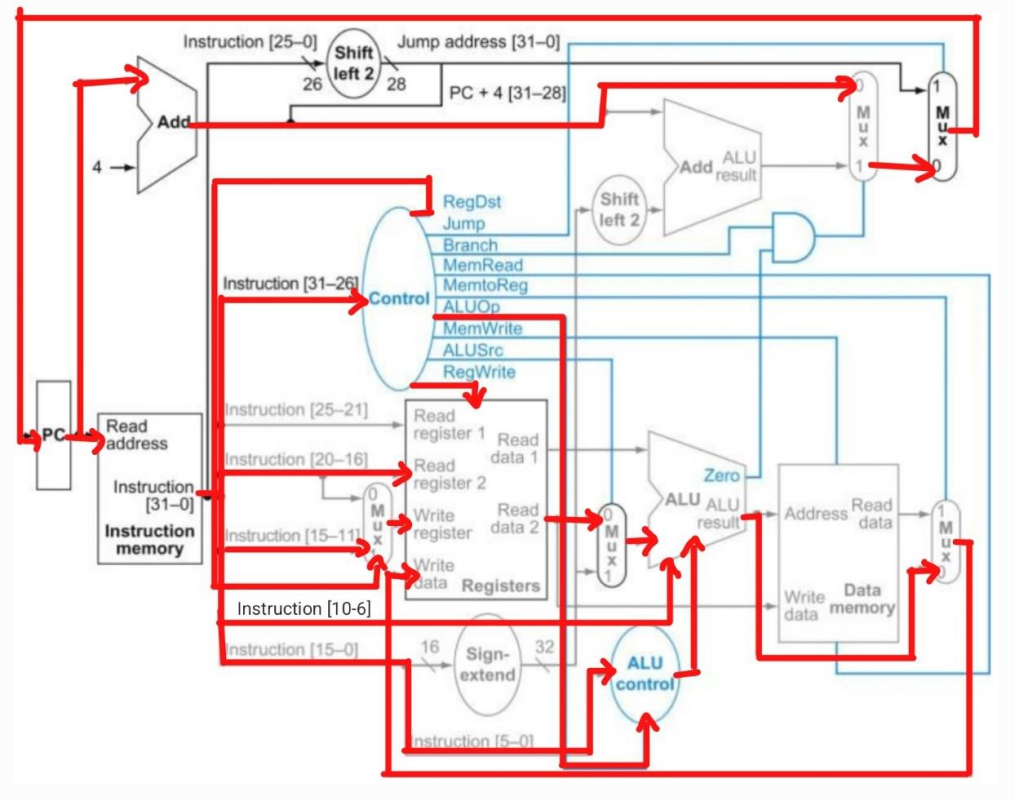
ALU 에서 rs 값과 rt 값이 - 연산돼야 합니다. 이 때 imm 값은 사용되지 않으므로 SEUmode x, ALU 연산에 rt 값이 사용되므로 ALUSrcB 00, ALUctrl 0x, ALUop 00111 이 됩니다.

Memory 에는 접근하지 않으므로 DataWidth xxx, MemWrite 0 입니다. ALU 출력 값이 rd 에 저장돼야 하므로 MemtoReg 0, RegDst 01, RegDatSel 00, RegWrite 1 입니다.

subu 는 branch 나 jump 명령어가 아니므로 Branch 000, Jump 00 입니다. 따라서 pc 현재 값이 Adder 에서 +4 되어 다시 pc 의 입력으로 들어갑니다.

4) SRA

instruction	opcode/function	syntax	operation
sra	000011	f $d, $t, sa	$d = $t >>> a



01_00_1_x_00_00_01111_xxx_0_0_000_00_xxxxx

sra 은 rt 를 shamt 만큼 sra 한 값을 rd 에 저장하는 연산을 수행하는 R-type 명령어입니다.

Control unit 에서 op 값 000011 을 읽어 다양한 신호들을 각 unit 에 전달합니다. sra 은 register 에서 값을 읽어, ALU 에서 연산을 수행하고, 결과 값을 register 에 저장합니다.

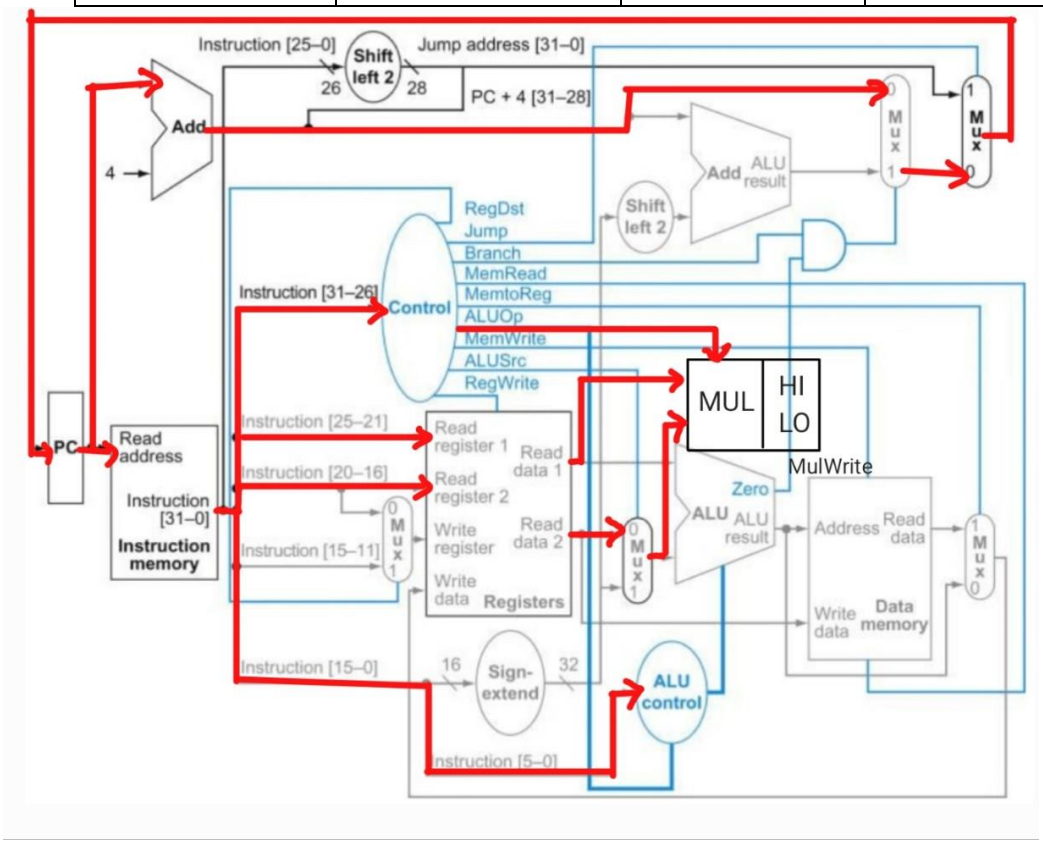
ALU 에서 rt 값이 <<< shamt 연산돼야 합니다. 이 때 imm 값은 사용되지 않으므로 SEUmode x, ALU 연산에 rt 값이 사용되므로 ALUSrcB 00, shamt 만큼 shift 연산을 진행하므로 ALUctrl 00, ALUop 01111 이 됩니다.

Memory 에는 접근하지 않으므로 DataWidth xxx, MemWrite 0 입니다. ALU 출력 값이 rd 에 저장돼야 하므로 MemtoReg 0, RegDst 01, RegDatSel 00, RegWrite 1 입니다.

sra 은 branch 나 jump 명령어가 아니므로 Branch 000, Jump 00 입니다. 따라서 pc 현재 값이 Adder 에서 +4 되어 다시 pc 의 입력으로 들어갑니다.

5) MULTU

instruction	opcode/function	syntax	operation
multu	011001	f $s, $t	hi:lo = $s * $t



xx_xx_0_x_00_0x_01010_xxx_0_x_000_00_xxxxx

multu 은 rs, rt 를 곱한 값을 MUL unit 의 HI:LO 에 저장하는 연산을 수행하는 R-type 명령어입니다.

Control unit 에서 op 값 011001 을 읽어 다양한 신호들을 각 unit 에 전달합니다. multu 은 register 에서 값을 읽어, MUL 에서 연산을 수행하고, 결과 값을 HI:LO 에 저장합니다.

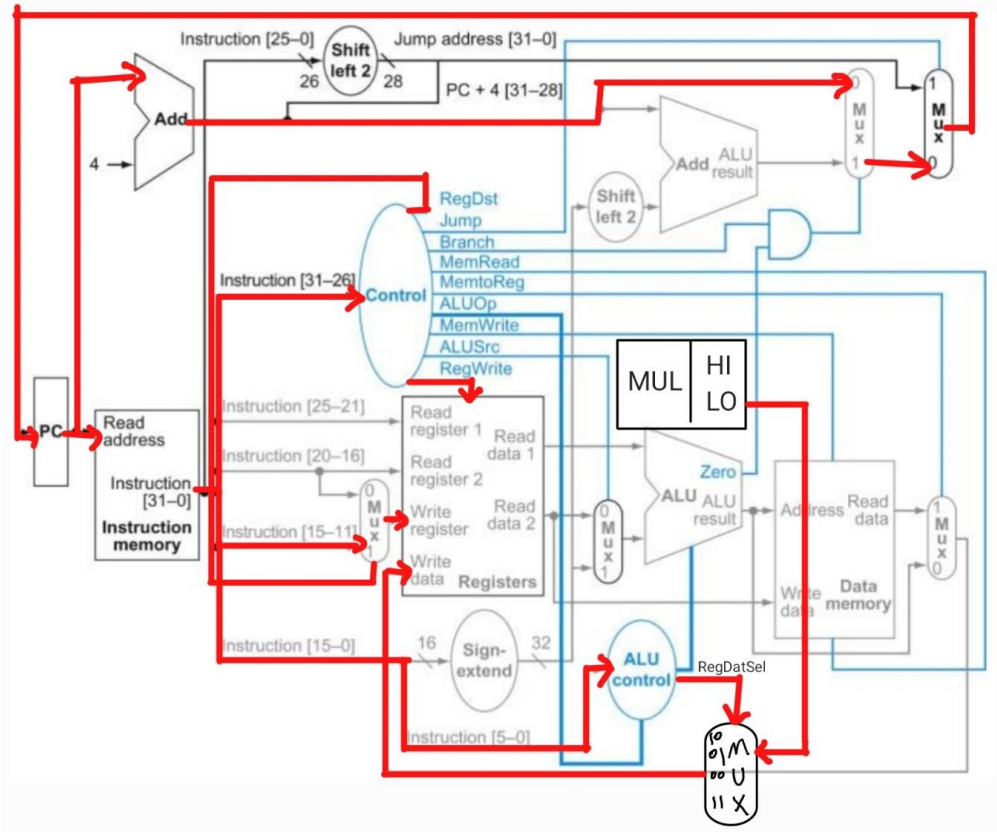
MUL 에 rs, rt 가 전달되면 내부에서 곱연산이 진행됩니다. 이 때 imm 값은 사용되지 않으므로 SEUmode x, MUL 연산에 rt 값이 사용되므로 ALUSrcB 00, ALUctrl 0x, ALUop 01010 이 됩니다.

Memory 에는 접근하지 않으므로 DataWidth xxx, MemWrite 0 입니다. 연산의 결과값은 MUL 내부 HI:LO register 에 저장되므로 MemtoReg x, RegDst xx, RegDatSel xx, RegWrite 0 입니다.

multu 은 branch 나 jump 명령어가 아니므로 Branch 000, Jump 00 입니다. 따라서 pc 현재 값이 Adder 에서 +4 되어 다시 pc 의 입력으로 들어갑니다.

6) MFLO

instruction	opcode/function	syntax	operation
mflo	010010	xf $d	$d = lo



01_01_1_x_xx_xx_xxxxx_xxx_0_x_000_00_xxxxx

mflo 은 MUL unit 의 LO 값을 rd 에 저장하는 연산을 수행하는 R-type 명령어입니다.

Control unit 에서 op 값 010010 을 읽어 다양한 신호들을 각 unit 에 전달합니다. mflo 은 MUL 에서 register LO 값을 읽어 rd 에 저장합니다.

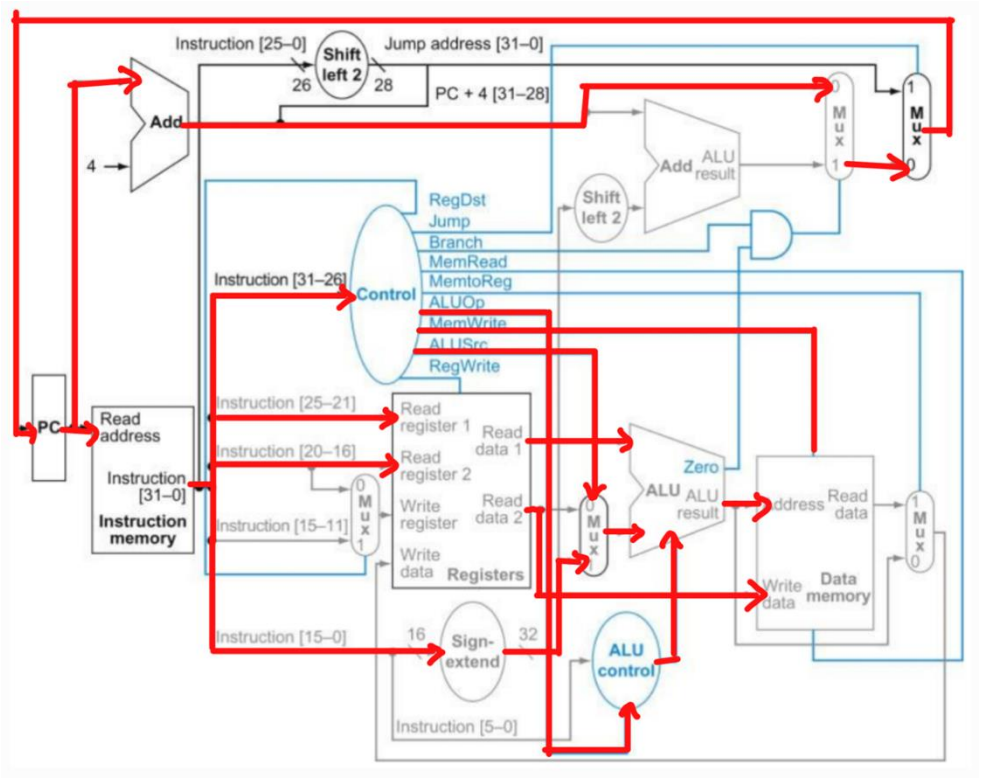
MUL Lo register 에 저장된 값이 register file 의 write data 값으로 입력돼야합니다. mflo 는 ALU 에서의 연산이 필요 없기 때문에 SEUmode x, ALUSrcB xx, ALUctrl xx, ALUop xxxxx 입니다.

Memory 에는 접근하지 않으므로 DataWidth xxx, MemWrite 0 입니다. MUL 내부 LO register 의 값이 rd 에 저장되므로 MemtoReg x, RegDst 01 RegDatSel 01, RegWrite 1 입니다.

multu 은 branch 나 jump 명령어가 아니므로 Branch 000, Jump 00 입니다. 따라서 pc 현재 값이 Adder 에서 +4 되어 다시 pc 의 입력으로 들어갑니다.

7) SB

instruction	opcode/function	syntax	operation
sb	101000	o $t, i ($s)	MEM [$s + i]:1 = LB ($t)



xx_xx_0_1_01_0x_00100_011_1_x_000_00_xxxxx

sb 은 rs 와 imm 를 +한 값을 Memory 에 address 로 전달하고, Memory 의 address 에 rt 값을 저장하는 I-type 명령어입니다.

Control unit 에서 op 값 101000 을 읽어 다양한 신호들을 각 unit 에 전달합니다. sb 은 register 에서 값을 읽어, ALU 에서 Memory 에 접근할 address 값을 계산하고, 해당 address 에 rt 값 중 하위 1byte 를 저장합니다..

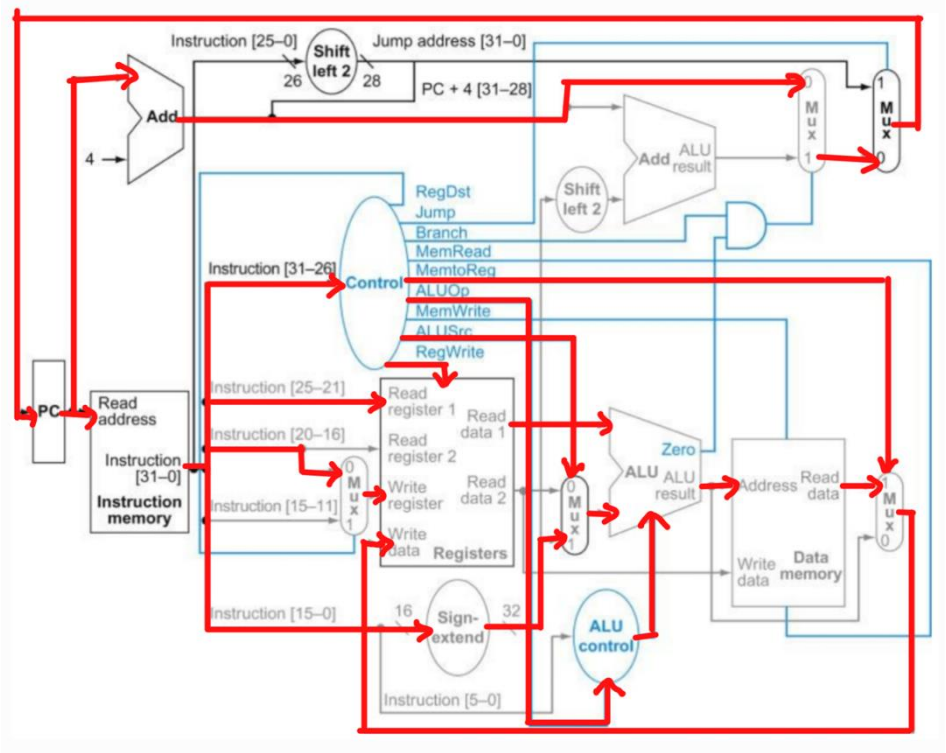
ALU 에서 sign extend 된 imm 값과 rs 값이 + 연산돼야 합니다. 이 때 imm 값이 sign extended 이므로 SEUmode 1, ALU 연산에 imm 값이 사용되므로 ALUSrcB 01, ALUctrl 0x, ALUop 00100 이 됩니다.

Memory 에는 접근하여 1byte 만을 저장하므로 DataWidth 011, MemWrite 1 입니다. register 에 write 하지 않으므로 MemtoReg x, RegDst xx, RegDatSel xx, RegWrite 0 입니다.

sb 은 branch 나 jump 명령어가 아니므로 Branch 000, Jump 00 입니다. 따라서 pc 현재 값이 Adder 에서 +4 되어 다시 pc 의 입력으로 들어갑니다.

8) LHU

instruction	opcode/function	syntax	operation
lhu	100101	o $t, i ($s)	$t = ZE (MEM [$s + i]:2)



00_00_1_1_01_0x_00100_010_0_1_000_00_xxxxx

lhu 은 rs 와 imm 를 +한 값을 Memory 에 address 로 전달하고, Memory 의 address 에서 2byte 를 zero-extend 한 값을 rt 에 저장하는 I-type 명령어입니다.

Control unit 에서 op 값 100101 을 읽어 다양한 신호들을 각 unit 에 전달합니다. lhu 은 register 에서 값을 읽어, ALU 에서 Memory 에 접근할 address 값을 계산하고, 해당 address 에서 2byte 데이터를 zero extend 한 값을 rt 에 저장합니다..

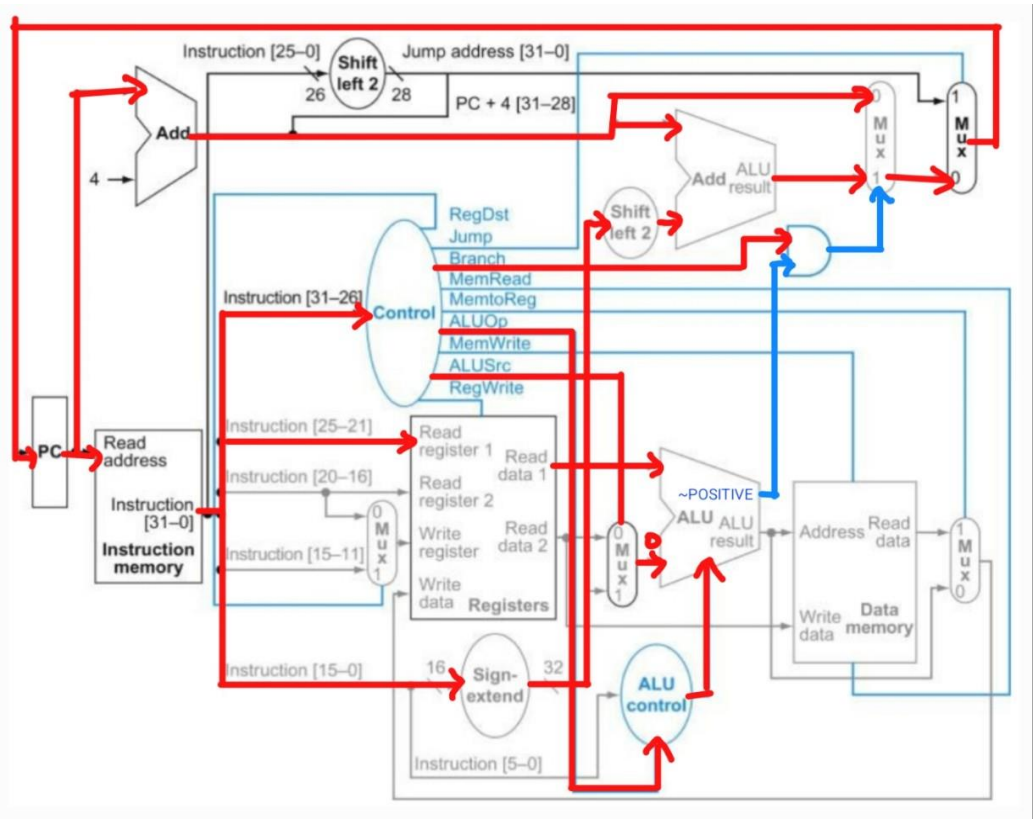
ALU 에서 sign extend 된 imm 값과 rs 값이 + 연산돼야 합니다. 이 때 imm 값이 sign extended 이므로 SEUmode 1, ALU 연산에 imm 값이 사용되므로 ALUSrcB 01, ALUctrl 0x, ALUop 00100 이 됩니다.

Memory 에 접근하여 2byte 만을 읽어오기만 하므로 DataWidth 010, MemWrite 0 입니다. register 에 write 하므로 MemtoReg 1, RegDst 00, RegDatSel 00, RegWrite 1 입니다.

lhu 은 branch 나 jump 명령어가 아니므로 Branch 000, Jump 00 입니다. 따라서 pc 현재 값이 Adder 에서 +4 되어 다시 pc 의 입력으로 들어갑니다.

9) BLEZ

instruction	opcode/function	syntax	operation
blez	000110	o $s, label	if ($s <= 0) pc += i << 2



xx_xx_0_1_10_0x_00110_xxx_0_x_110_00_xxxxx

blez 은 rs <= 0 이면 branch 를 실행하고, rs > 0 이면 PC += 4 를 수행하는 I-type 명령어입니다.

Control unit 에서 op 값 000110 을 읽어 다양한 신호들을 각 unit 에 전달합니다. blez 은 register 에서 값을 읽어, ALU 에서 0 과 비교연산을 진행하고, 그 결과 값에 따라 branch 를 실행할지를 결정합니다.

ALU 에서 rs 값과 0 을 비교하기 위해 -연산을 진행합니다. ALU 연산에 rs 와 0 이 사용되므로 ALUSrcB 10, ALUctrl 0x, ALUop 00110 이 됩니다.

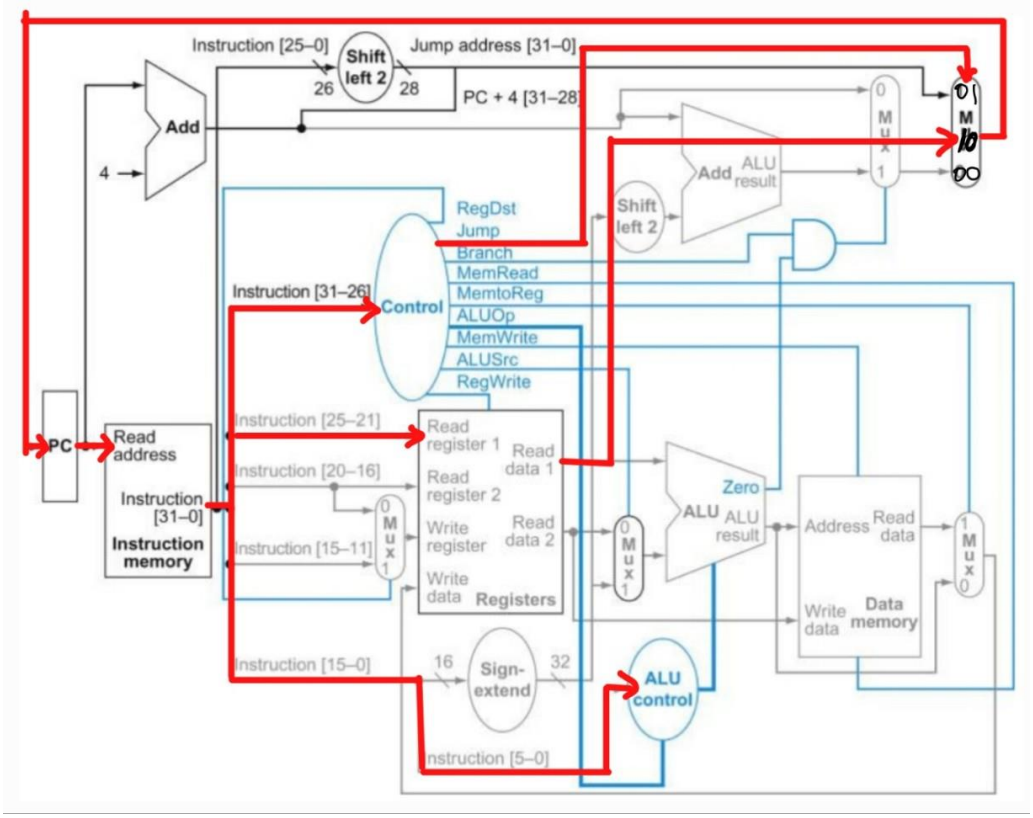
Memory 에 접근하지 않으므로 DataWidth xxx, MemWrite 0 입니다. register 에 write 하지 않으므로 MemtoReg x, RegDst xx, RegDatSel xx, RegWrite 0 입니다.

imm 값은 branch 할 PC 값을 생성하기 위해 PC + 4 + (SE(i) << 2) 해야하므로 SEUmode 1 입니다.

blez 은 rs <= 0 일 때 수행되는 branch 명령어이므로 Branch 110, Jump 00 입니다. ALU Positive flag 가 0 이 되면 rs <= 0 을 만족하므로 mux 에서 branch 결과 값을 pc 입력으로 들어가게 합니다. Positive flag 가 1 이 되면 rs > 0 이므로 pc 현재 값이 Adder 에서 +4 되어 다시 pc 의 입력으로 들어갑니다.

10) JR

instruction	opcode/function	syntax	operation
jr	001000	f labelR	pc = $s



xx_xx_0_x_10_0x_00100_xxx_0_x_000_10_xxxxx

jr 는 pc 를 rs 값으로 jump 시키는 R-type 명령어입니다.

Control unit 에서 op 값 001000 을 읽어 다양한 신호들을 각 unit 에 전달합니다. jr 는 rs 값을 읽어 곧바로 pc 의 입력으로 전달합니다.

ALU 연산이 필요없고, imm 값 또한 사용되지 않으므로, SEUmode x, ALUSrcB 10, ALUctrl 0x, ALUop 00100 가 됩니다.

Memory 에는 접근하지 않으므로 DataWidth xxx, MemWrite 0 입니다. register 에 write 되지 않으므로 MemtoReg x, RegDst xx, RegDatSel xx, RegWrite 0 입니다.

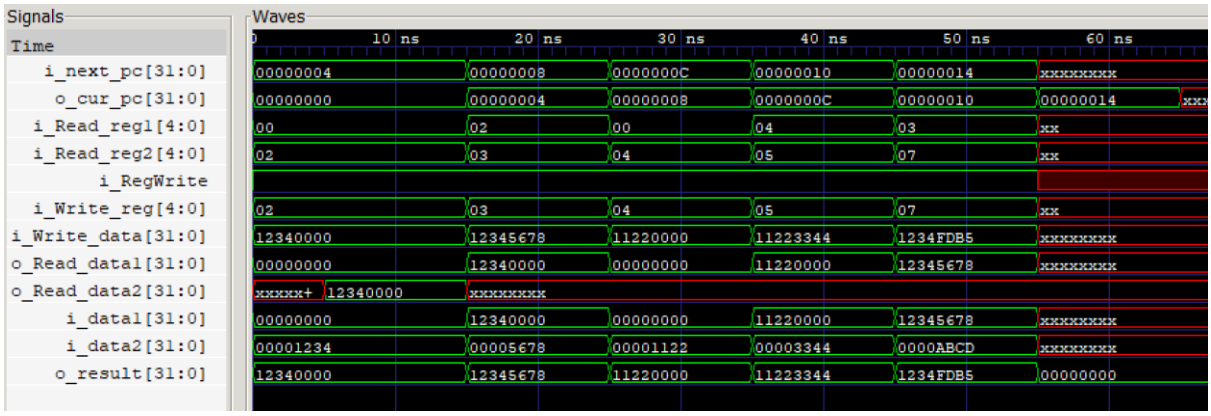
jr 는 jump 명령어이므로 Branch 000, Jump 10 입니다. 따라서 mux 의 sel signal jump(10)에 따라 rs 의 값이 pc 의 입력으로 들어갑니다.

- **testbench 분석**

-xori

```
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
// $2: 0x12340000   $3: 0x12345678    $4: 0x11220000    $5: 0x11223344

001110_00011_00111_1010101111001101 // xori $7, $3, 0xabcd
// expected $7: 0x1234FDB5

xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
000000_xxxxx_xxxxx_xxxxx_xxxxx_001101 // break
```

$3 register 값 0x12345678 ^ imm 0x0000abcd = 0x1234FDB5 이 $7 register 에 저장될 것으로 예상합니다.



pc 가 0x10, 즉 xori $7, $3, 0xabcd 를 수행할 차례에,

next_pc = 10 + 4 = 14

ALU input1 = 0x12345678 (= $3), input2 = 0x0000abcd (= imm)

ALU result = 0x1234FDB5

RegWrite = 1

Write_reg = 0x07

임을 확인할 수 있습니다.

따라서 다음 clock 에 $7 register 에 0x1234FDB5 가 저장될 것을 예상할 수 있습니다.

```
1 00000000 00000000 00000000 00000000 : 00000000
2 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
3 00010010 00110100 00000000 00000000 : 12340000
4 00010010 00110100 01010110 01111000 : 12345678
5 00010001 00100010 00000000 00000000 : 11220000
6 00010001 00100010 00110011 01000100 : 11223344
7 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
8 00010010 00110100 11111101 10110101 : 1234fdb5
```

예상대로 $7 register 에 0x1234FDB5 가 write 된 것을 확인할 수 있습니다.

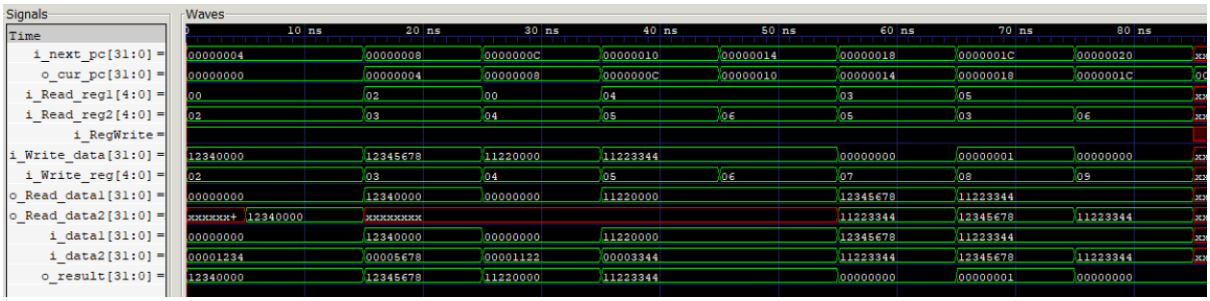
-slt

```
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
001101_00100_00110_0011001101000100 // ori $6, $4, 0x3344
// $2: 0x12340000   $3: 0x12345678    $4: 0x11220000    $5: 0x11223344    $6: 0x11223344

000000_00011_00101_00111_xxxxx_101010 // slt $7, $3, $5
// expected $7: 0x00000000

000000_00101_00011_01000_xxxxx_101010 // slt $8, $5, $3
// expected $8: 0x00000001

000000_00101_00110_01001_xxxxx_101010 // slt $9, $5, $6
// expected $9: 0x00000000

xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
000000_xxxxx_xxxxx_xxxxx_xxxxx_001101 // break
```

$3 register 값 0x12345678 > $5 register 값 0x11223344 이므로 $7 에 0x00000000 이 write 될 것으로 예상합니다.

$5 register 값 0x11223344 > $3 register 값 0x12345678 이므로 $8 에 0x00000001 이 write 될 것으로 예상합니다.

$5 register 값 0x11223344 == $6 register 값 0x11223344 이므로 $8 에 0x00000000 이 write 될 것으로 예상합니다.



pc 가 0x14, 즉 slt $7, $3, $5 를 수행할 차례에,

next_pc = 14 + 4 = 18

ALU input1 = 0x12345678 (= $3), input2 = 0x11223344 (= $5)

ALU result = 0x00000000

RegWrite = 1

Write_reg = 0x07

임을 확인할 수 있습니다.

따라서 다음 clock 에 $7 register 에 0x00000000 가 write 될 것을 예상할 수 있습니다.

pc 가 0x18, 즉 slt $8, $5, $3 를 수행할 차례에,

next_pc = 18 + 4 = 1c

ALU input1 = 0x11223344 (= $5), input2 = 0x12345678 (= $3)

ALU result = 0x00000001

RegWrite = 1

Write_reg = 0x08

임을 확인할 수 있습니다.

따라서 다음 clock 에 $8 register 에 0x00000001 가 write 될 것을 예상할 수 있습니다.

pc 가 0x1c, 즉 slt $9, $5, $6 를 수행할 차례에,

next_pc = 1c + 4 = 20

ALU input1 = 0x11223344 (= $5), input2 = 0x11223344 (= $6)

ALU result = 0x00000000

RegWrite = 1

Write_reg = 0x09

임을 확인할 수 있습니다.

따라서 다음 clock 에 $9 register 에 0x00000000 가 write 될 것을 예상할 수 있습니다.

```
1 00000000 00000000 00000000 00000000 : 00000000
2 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
3 00010010 00110100 00000000 00000000 : 12340000
4 00010010 00110100 01010110 01111000 : 12345678
5 00010001 00100010 00000000 00000000 : 11220000
6 00010001 00100010 00110011 01000100 : 11223344
7 00010001 00100010 00110011 01000100 : 11223344
8 00000000 00000000 00000000 00000000 : 00000000
9 00000000 00000000 00000000 00000001 : 00000001
10 00000000 00000000 00000000 00000000 : 00000000
```

예상대로 각 register 에 올바른 값이 write 되는 것을 확인할 수 있습니다.

$7 register: 0x00000000

$8 register: 0x00000001

$9 register: 0x00000000

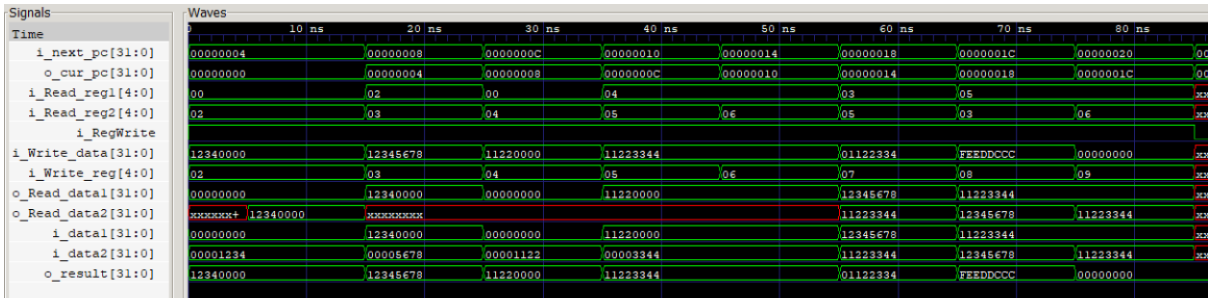
-subu

```
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
001101_00100_00110_0011001101000100 // ori $6, $4, 0x3344
// $2: 0x12340000   $3: 0x12345678    $4: 0x11220000    $5: 0x11223344    $6: 0x11223344

000000_00011_00101_00111_xxxxx_100011 // subu $7, $3, $5
// $7: 0x01122334

000000_00101_00011_01000_xxxxx_100011 // subu $8, $5, $3
// $8: 0xfeeddccc

000000_00101_00110_01001_xxxxx_100011 // subu $9, $5, $6
// $9: 0x00000000

000000_xxxxx_xxxxx_xxxxx_xxxxx_001101 // break
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
```

$3 register 값 0x12345678 - $5 register 값 0x11223344 = 0x01122334 가 $7 에 write 될 것으로 예상합니다.

$5 register 값 0x11223344 - $3 register 값 0x12345678 = 0xfeeddccc 가 $8 에 write 될 것으로 예상합니다.

$5 register 값 0x11223344 - $6 register 값 0x11223344 = 0x00000000 가 $8 에 write 될 것으로 예상합니다.



pc 가 0x14, 즉 subu $7, $3, $5 를 수행할 차례에,

next_pc = 14 + 4 = 18

ALU input1 = 0x12345678 (= $3), input2 = 0x11223344 (= $5)

ALU result = 0x01122334

RegWrite = 1

Write_reg = 0x07

임을 확인할 수 있습니다.

따라서 다음 clock 에 $7 register 에 0x01122334 가 write 될 것을 예상할 수 있습니다.

pc 가 0x18, 즉 subu $8, $5, $3 를 수행할 차례에,

next_pc = 18 + 4 = 1c

ALU input1 = 0x11223344 (= $5), input2 = 0x12345678 (= $3)

ALU result = 0xfeeddccc

RegWrite = 1

Write_reg = 0x08

임을 확인할 수 있습니다.

따라서 다음 clock 에 $8 register 에 0xfeeddccc 가 write 될 것을 예상할 수 있습니다.

pc 가 0x1c, 즉 subu $9, $5, $6 를 수행할 차례에,

next_pc = 1c + 4 = 20

ALU input1 = 0x11223344 (= $5), input2 = 0x11223344 (= $6)

ALU result = 0x00000000

RegWrite = 1

Write_reg = 0x08

임을 확인할 수 있습니다.

따라서 다음 clock 에 $9 register 에 0x00000000 가 write 될 것을 예상할 수 있습니다.

```
00000000 00000000 00000000 00000000 : 00000000
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
00010010 00110100 00000000 00000000 : 12340000
00010010 00110100 01010110 01111000 : 12345678
00010001 00100010 00000000 00000000 : 11220000
00010001 00100010 00110011 01000100 : 11223344
00010001 00100010 00110011 01000100 : 11223344
00000001 00010010 00100011 00110100 : 01122334
11111110 11101101 11011100 11001100 : feeddccc
00000000 00000000 00000000 00000000 : 00000000
```

예상대로 각 register 에 올바른 값이 write 되는 것을 확인할 수 있습니다.

$7 register: 0x01122334

$8 register: 0xfeeddccc

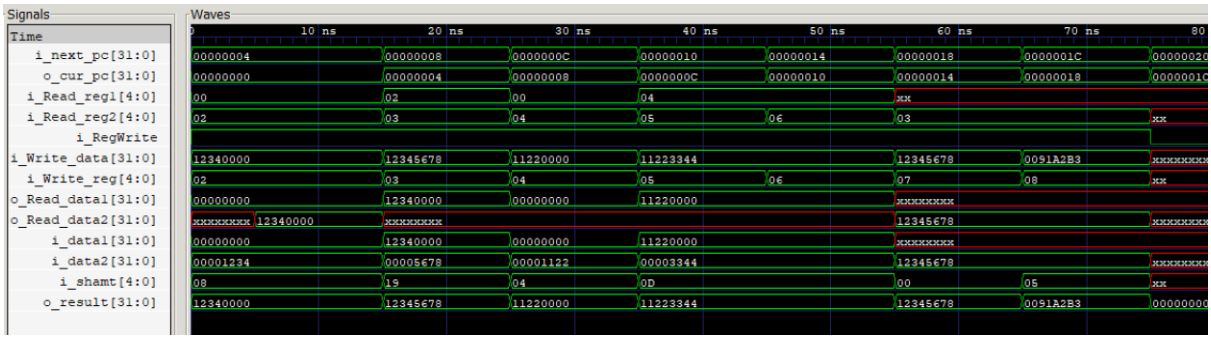
$9 register: 0x00000000

-sra

```
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
001101_00100_00110_0011001101000100 // ori $6, $4, 0x3344
// $2: 0x12340000   $3: 0x12345678    $4: 0x11220000    $5: 0x11223344    $6: 0x11223344

000000_xxxxx_00011_00111_00000_000011 // sra $7, $3, 0
// $7: 0x12345678

000000_xxxxx_00011_01000_00101_000011 // sra $8, $3, 5
// $8: 0x0091A2B3

000000_xxxxx_xxxxx_xxxxx_xxxxx_001101 // break
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
```

$3 register 값 0x12345678 >>> shamt 값 0x00000 = 0x12345678 가 $7 에 write 될 것으로 예상합니다.

$3 register 값 0x12345678 >>> shamt 값 0x00101 = 0x0091A2B3 가 $8 에 write 될 것으로 예상합니다.



pc 가 0x14, 즉 sra $7, $3, 0 를 수행할 차례에,

next_pc = 14 + 4 = 18

ALU input1 = 0x12345678 (= $3), input2 = 0x00000 (= shamt)

ALU result = 0x12345678

RegWrite = 1

Write_reg = 0x07

임을 확인할 수 있습니다.

따라서 다음 clock 에 $7 register 에 0x12345678 가 write 될 것을 예상할 수 있습니다.

pc 가 0x18, 즉 sra $8, $3, 5 를 수행할 차례에,

next_pc = 18 + 4 = 1c

ALU input1 = 0x12345678 (= $3), input2 = 0x00101 (= shamt)

ALU result = 0x0091A2B3

RegWrite = 1

Write_reg = 0x08

임을 확인할 수 있습니다.

따라서 다음 clock 에 $8 register 에 0x0091A2B3 가 write 될 것을 예상할 수 있습니다.

```
00000000 00000000 00000000 00000000 : 00000000
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
00010010 00110100 00000000 00000000 : 12340000
00010010 00110100 01010110 01111000 : 12345678
00010001 00100010 00000000 00000000 : 11220000
00010001 00100010 00110011 01000100 : 11223344
00010001 00100010 00110011 01000100 : 11223344
00010010 00110100 01010110 01111000 : 12345678
00000000 10010001 10100010 10110011 : 0091a2b3
```

예상대로 각 register 에 올바른 값이 write 되는 것을 확인할 수 있습니다.

$7 register: 0x12345678

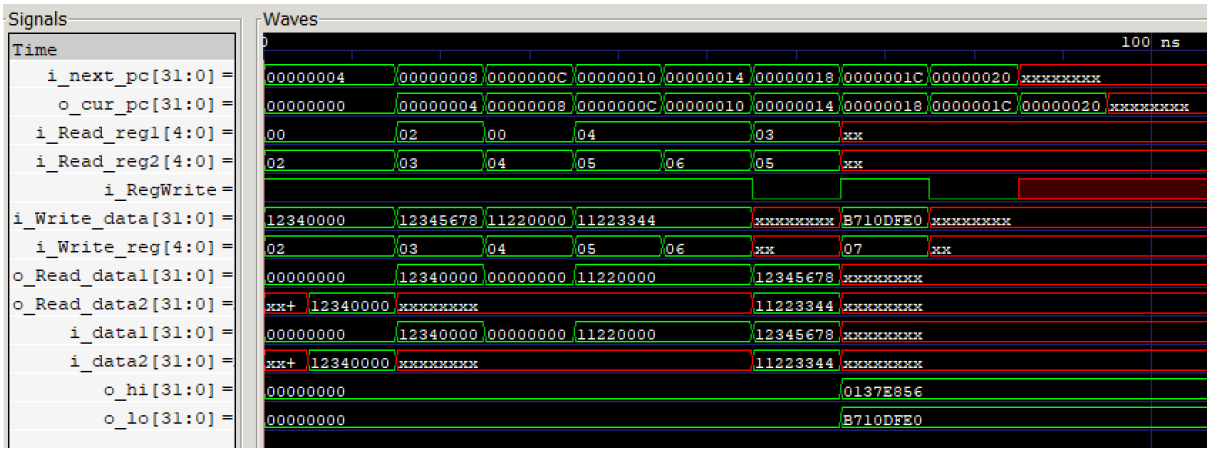
$8 register: 0x0091A2B3

-multu, mflo

```
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
001101_00100_00110_0011001101000100 // ori $6, $4, 0x3344
// $2: 0x12340000   $3: 0x12345678    $4: 0x11220000    $5: 0x11223344    $6: 0x11223344

000000_00011_00101_xxxxx_xxxxx_011001 // multu $3, $5
// hi: 0x0137E856   lo: B710DFE0

000000_xxxxx_xxxxx_00111_xxxxx_010010 // mflo $7
// $7: 0xB710DFE0

000000_xxxxx_xxxxx_xxxxx_xxxxx_001101 // break
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
```

$3 register 값 0x12345678 * $5 register 값 0x11223344 = 0x0137E856B710DFE0 가 MUL 의 register HI:LO 에 write 될 것으로 예상합니다.

MUL 의 LO register 값 0xB710DFE0 이 $7 에 write 될 것으로 예상합니다.



pc 가 0x14, 즉 multu $3, $5 를 수행할 차례에,

next_pc = 14 + 4 = 18

MUL input1 = 0x12345678 (= $3), input2 = 0x11223344 (= $5)

MUL expected result = 0x0137E856B710DFE0

임을 확인할 수 있습니다.

따라서 다음 clock 에 HI: 0x0137E856, LO: B710DFE0 가 write 된 것을 확인할 수 있습니다.

또한 Write_reg = 0x07, RegWrite = 1, Write_data = 0xB710DFE0 임을 확인할 수 있습니다.

따라서 다음 clock 에서 $7 에 0xB710DFE0 이 write 될 것으로 예상합니다.

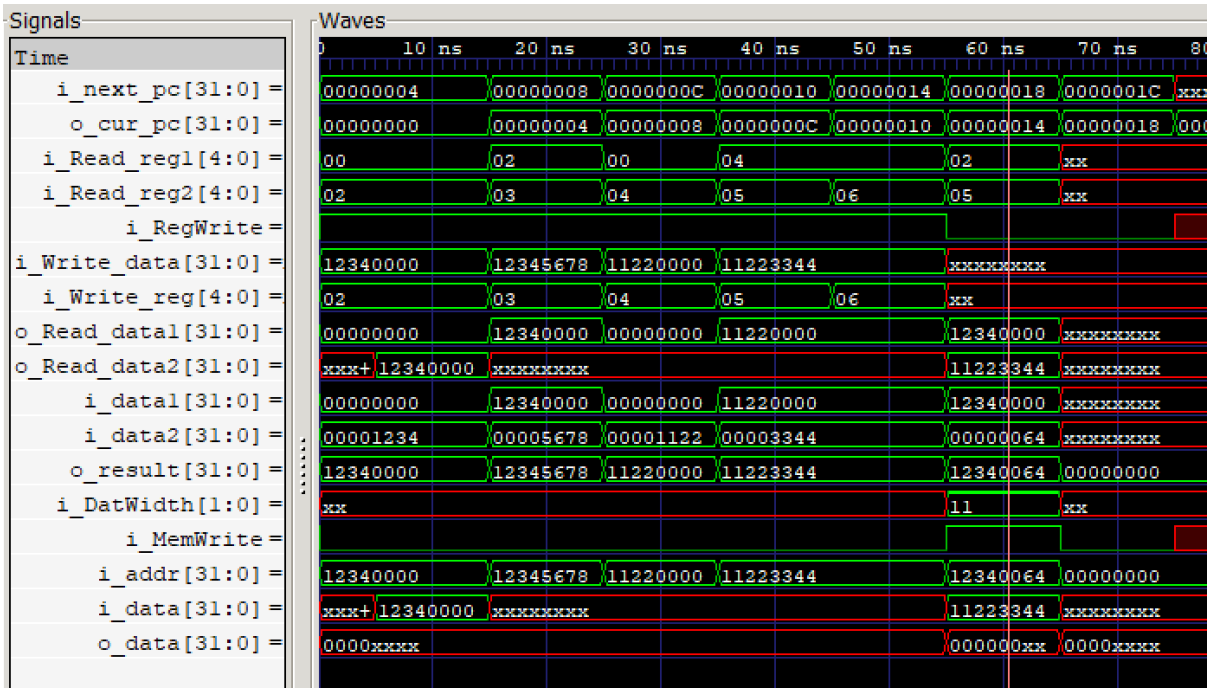
```
1 00000000 00000000 00000000 00000000 : 00000000
2 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
3 00010010 00110100 00000000 00000000 : 12340000
4 00010010 00110100 01010110 01111000 : 12345678
5 00010001 00100010 00000000 00000000 : 11220000
6 00010001 00100010 00110011 01000100 : 11223344
7 00010001 00100010 00110011 01000100 : 11223344
8 10110111 00010000 11011111 11100000 : b710dfe0
9 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
```

예상대로 $7 register 에 올바른 값 0xB710DFE0 이 write 되는 것을 확인할 수 있습니다.

-sb

```
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
001101_00100_00110_0011001101000100 // ori $6, $4, 0x3344
// $2: 0x12340000    $3: 0x12345678    $4: 0x11220000    $5: 0x11223344    $6: 0x11223344

101000_00010_00101_0000000001100100// sb $5, 100($2)
// MEM[0x12340064] = 0x44
```

$2 register 값 0x12340000 + imm 값 0x00000064 = 0x12340064가 Memory의 address에 입력되고, Memory 의 0x12340064 번지에 $5 의 LSB 1byte 0x44 가 write 될 것으로 예상합니다.



pc가 0x14, 즉 sb $5, 100($2)를 수행할 차례에,

next_pc = 14 + 4 = 18

ALU input1 = 0x12340000 (= $2), input2 = 0x00000064 (= imm)

ALU result = 0x12340064

RegWrite = 0

MemWrite = 1

MemAddr = 0x12340064

MemData = 11223344 (= $5)

임을 확인할 수 있습니다.

따라서 다음 clock 에 Mem[0x12340064]에 0x44 가 write 될 것을 예상할 수 있습니다.

```
1 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
2 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
3 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
4 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
5 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
6 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
7 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
8 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
9 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
10 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
11 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
12 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
13 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
14 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
15 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
16 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
17 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
18 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
19 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
20 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
21 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
22 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
23 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
24 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
25 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
26 xxxxxxxxxxxxxxxxxxxxxxxx01000100 xxxxxx44
```

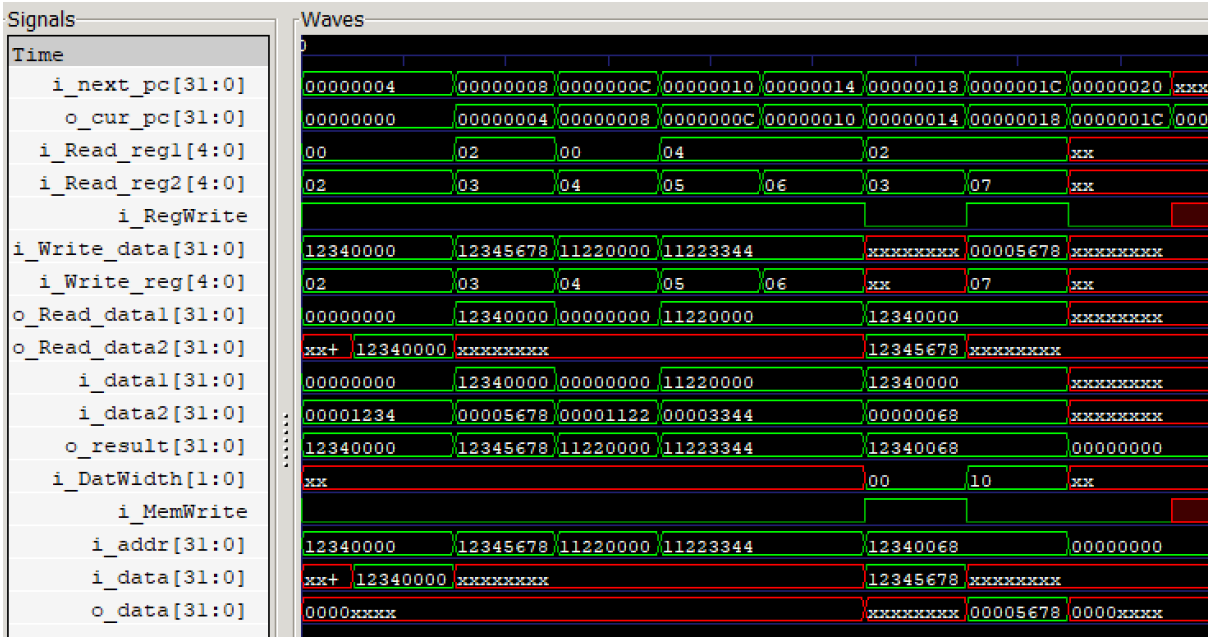
line 한 줄 당 4byte, 왼쪽이 더 큰 주소 값을 가지므로, 0x44 는 정확히 0x64 번지에 저장돼 있음을 확인할 수 있습니다.

-lhu

```
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
001101_00100_00110_0011001101000100 // ori $6, $4, 0x3344
// $2: 0x12340000   $3: 0x12345678    $4: 0x11220000    $5: 0x11223344    $6: 0x11223344

101011_00010_00011_0000000001101000// sw $3, 104($2)
// MEM[0x12340068] = 0x78
// MEM[0x12340069] = 0x56
// MEM[0x1234006a] = 0x34
// MEM[0x1234006b] = 0x12

100101_00010_00111_0000000001101000 // lhu $7, 104($2)
// $7: 00005678

000000_xxxxx_xxxxx_xxxxx_xxxxx_001101 // break
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
```

$2 register 값 0x12340000 + imm 값 0x00000068 = 0x12340068 가 Memory 의 address 에 입력되고, Little Endian 방식을 따라, Memory 의 0x12340068 번지부터 차례로 $5 의 값이 0x78, 0x56, 0x34, 0x12 가 write 될 것으로 예상합니다.

$2 register 값 0x12340000 + imm 값 0x00000068 = 0x12340068 가 Memory 의 address 에 입력되고, Memory 의 0x12340068 번지에서부터 2byte 의 값을 Zero Extend 한 값을 $7 에 write 할 것으로 예상합니다.



pc 가 0x14, 즉 sw $3, 104($2)를 수행할 차례에,

next_pc = 14 + 4 = 18

ALU input1 = 0x12340000 (= $2), input2 = 0x00000068 (= imm)

ALU result = 0x12340068

RegWrite = 0

MemWrite = 1

MemAddr = 0x12340068

MemInputData = 12345678 (= $3)

임을 확인할 수 있습니다.

따라서 다음 clock 에 Mem[0x12340068]에 0x78, 0x56, 0x34, 0x12 가 write 될 것을 예상할 수 있습니다.

pc 가 0x18, 즉 lhu $7, 104($2)를 수행할 차례에,

next_pc = 18 + 4 = 1c

ALU input1 = 0x12340000 (= $2), input2 = 0x00000068 (= imm)

ALU result = 0x12340068

RegWrite = 1

MemWrite = 0

MemAddr = 0x12340068

MemOutputData = 0x00005678

임을 확인할 수 있습니다.

따라서 다음 clock 에 $7 에 0x00005678 이 write 될 것을 예상할 수 있습니다.

```
1 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
2 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
3 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
4 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
5 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
6 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
7 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
8 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
9 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
10 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
11 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
12 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
13 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
14 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
15 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
16 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
17 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
18 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
19 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
20 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
21 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
22 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
23 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
24 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
25 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
26 xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx
27 00010010001101000101011001111000 12345678
```

```
1 00000000 00000000 00000000 00000000 : 00000000
2 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
3 00010010 00110100 00000000 00000000 : 12340000
4 00010010 00110100 01010110 01111000 : 12345678
5 00010001 00100010 00000000 00000000 : 11220000
6 00010001 00100010 00110011 01000100 : 11223344
7 00010001 00100010 00110011 01000100 : 11223344
8 00000000 00000000 01010110 01111000 : 00005678
```

예상과 같이, $7 에 0x00005678 이 write 된 것을 확인할 수 있습니다.

-blez, jr

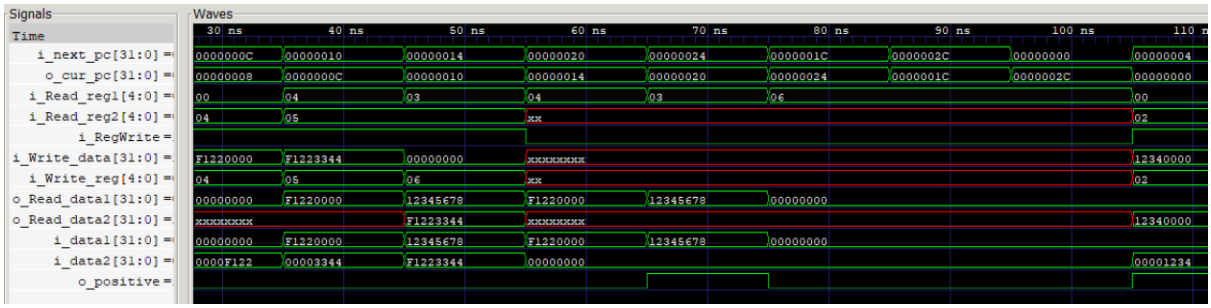
```
001111_00000_00010_0001001000110100 // lui $2, 0x1234
001101_00010_00011_0101011001111000 // ori $3, $2, 0x5678
001111_00000_00100_0001000100100010 // lui $4, 0xF122
001101_00100_00101_0011001101000100 // ori $5, $4, 0x3344
000000_00011_00101_00110_xxxxx_101010 // slt $6, $3, $5
// $2: 0x12340000   $3: 0x12345678    $4: 0xF1220000    $5: 0xF1223344    $6: 0x00000000

000110_00100_xxxxx_0000000000000010 // blez $4, +2
// pc += (4 + 2 << 2) : 14 -> 20

000000_xxxxx_xxxxx_xxxxx_xxxxx_001101 // break

000110_00110_xxxxx_0000000000000011 // blez $6, 3
// pc += (4 + 3 << 2) : 1c -> 2c

000110_00011_xxxxx_1111111111111111 // blez $3, -1
// pc += 4 : 20 -> 24

000110_00110_xxxxx_1111111111111101 // blez $6, -3
// pc += (4 + -3 << 2) : 24 -> 1c

000000_xxxxx_xxxxx_xxxxx_xxxxx_001101 // break

000000_00110_xxxxx_xxxxx_xxxxx_001000 // jr $6
// PC = 0

xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx
```

register $3 값 0x12345678 > 0 이므로 pc 가 +4 됩니다.

register $4 값 0xF1220000 <= 0 이므로 pc 가 branch 됩니다.

register $6 값 0x00000000 <= 0 이므로 pc 가 branch 됩니다.

따라서 pc 값은 14->20->24->1c->2c->0 ...이 됩니다.



pc 가 0x14, 즉 blez $4, 2 를 수행할 차례에,

ALU input1 = 0xF1220000 (= $4), input2 = 0x00000000

ALU Positive = 0

RegWrite = 0

MemWrite = 0

ALU Positive = 0 이므로 branch 를 수행하게 된다.

next_pc = 14 + c = 20

따라서 다음 clock 에 pc 는 20 이 될 것을 예상할 수 있습니다.

pc 가 0x20, 즉 blez $3, -1 를 수행할 차례에,

ALU input1 = 0x12345678 (= $3), input2 = 0x00000000

ALU Positive = 1

RegWrite = 0

MemWrite = 0

ALU Positive = 1 이므로 branch 를 수행하지 않는다.

next_pc = 20 + 4 = 24

따라서 다음 clock 에 pc 는 24 이 될 것을 예상할 수 있습니다.

pc 가 0x24, 즉 blez $6, -3 를 수행할 차례에,

ALU input1 = 0x00000000 (= $6), input2 = 0x00000000

ALU Positive = 0

RegWrite = 0

MemWrite = 0

ALU Positive = 0 이므로 branch 를 수행하게 된다.

next_pc = 24 + -8 = 1c

따라서 다음 clock 에 pc 는 1c 이 될 것을 예상할 수 있습니다.

pc 가 0x1c, 즉 blez $6, 3 를 수행할 차례에,

ALU input1 = 0x00000000 (= $6), input2 = 0x00000000

ALU Positive = 0

RegWrite = 0

MemWrite = 0

ALU Positive = 0 이므로 branch 를 수행하게 된다.

next_pc = 1c + 10 = 2c

따라서 다음 clock 에 pc 는 2c 이 될 것을 예상할 수 있습니다.

pc 가 0x2c, 즉 jr $6 를 수행할 차례에,

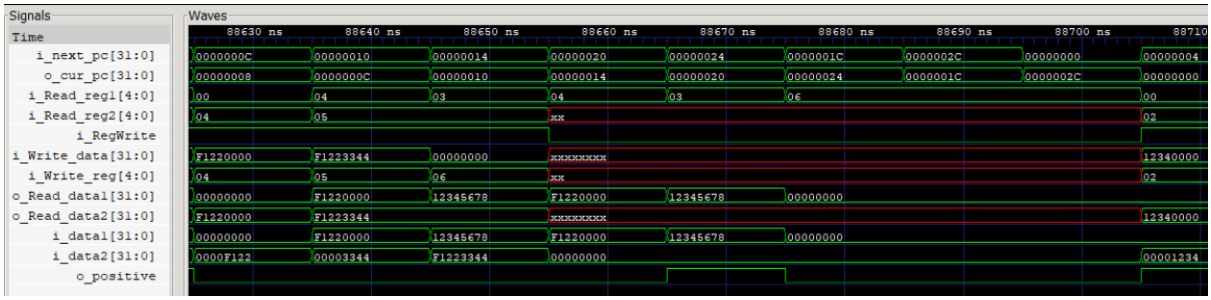
RegWrite = 0

MemWrite = 0

$6 = 0x00000000 이므로

next_pc = 0

따라서 다음 clock 에 pc 는 0 이 되고, 첫 번째 명령어부터 다시 실행하게 됩니다. 결과적으로 무한루프가 발생합니다.



(무한루프)

고찰

이론시간에 배운 MIPS Single Cycle CPU Implementation datapath 를 실습을 통해 직접 Control unit 의 signal 들을 지정해주는 경험을 함으로써 어떠한 명령어의 type 과 기능에 따라 알맞은 signal 이 무엇인지를 더 명확하게 알게 되었습니다. 다음 실습 내용인 MIPS Multi-Cycle CPU Implementation 를 구현하는데 있어서 이번 프로젝트 1 이 밑거름이 될 것 같다고 생각합니다.