Computer Architecture

Project 1

(MIPS Single Cycle CPU Implementation)

학　　과: 컴퓨터정보공학부

담당교수: 이성원 교수님

실습분반: 금34

학　　번: 2020202037

성　　명: 염정호

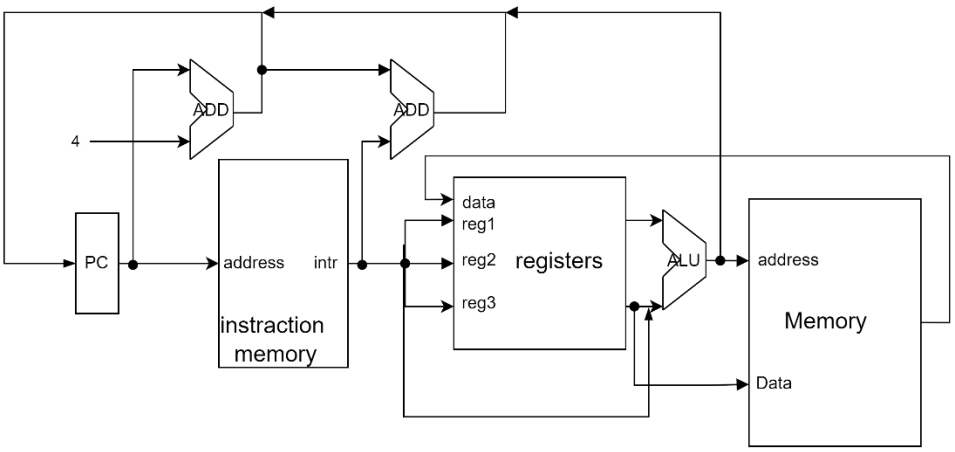
1. 프로젝트 개요

해당 프로젝트에서는 MIPS(Microprocessor without Interlocked Pipeline Stages) 아키텍처의 단일 사이클 CPU를 구현하는 것을 목표로 한다.

해당 프로젝트에서 LW, SW, ORI, ADD, SUB, J, LUI, BREAK, LLO, LHI 명령어들은 이미 구현 되어 있으며 해당 명령어의 작동 방식과 구성은 Test Banch를 통해 확인 가능하다. 여기에 추가로 XORI, SLT, SUBU, SRA, MULTU, MFLO, SB, LHU, BLEZ, JR 총 10개의 명령어를 구현하고 동작결과를 확인한다.

2. 설계 세부사항

블록 다이어그램



Mips CPU 하드웨어

module name	function
PC	명령어를 읽어들일 위치를 지정 branch나 jump명령이 입력되지 않으면 add를 통해 현재 값에 4를 더한 값 즉 다음 읽어들일 명령어의 위치를 가리키며 branch또는 jump명령어 활성화 시 다음 add와 mux를 통해 다음 pc의 주소를 지정
instruction memory	pc를 통해 입력된 명령어를 분석 및 각 모듈에 신호를 할당.
Registers	**instruction memory**로부터 받은 레지스터 주소를 이용해 연산을 수행 reg3 레지스터는 값을 저장하기 위한 목적지 레지스터의 주소를 할당. Alu로 값을 보낼 때 reg1($s)의 값은 그대로 사용하지만 reg2의 경우 명령어에 따라 Immediate value 값이 입력될 수도 있음 값의 선택은 mux를 통해 결정

Alu	레지스터로부터 받은 값을 이용해 산술 및 논리 연산 수행 연산의 결과를 메모리 또는 레지스터로 보내 값을 저장하거나 분기 명령어 사용시 조건의 만족 여부를 확인하는데 사용
Memory	프로그램이 사용하는 데이터를 저장. **instruction memory**와는 달리, 프로그램의 실행 중에 데이터를 저장하고 로드.
add	cycle cpu인 mips의 pc값을 처리하는데 사용 alu가 연산을 수행할 동안 pc의 주소를 이동하는데 사용된다.

명령어 타입

R : op(6)_$rs(5)_$rt(5)_$rd(5)_shamt(5)_func(6)

I : op(6)_$rs(5)_$rt(5)_ Immediate value (16)

J : op(6)_ Immediate value (26)

기능 및 동작

Control signal을 본 보고서에선 ctr_sig 이라 표현하겠습니다.

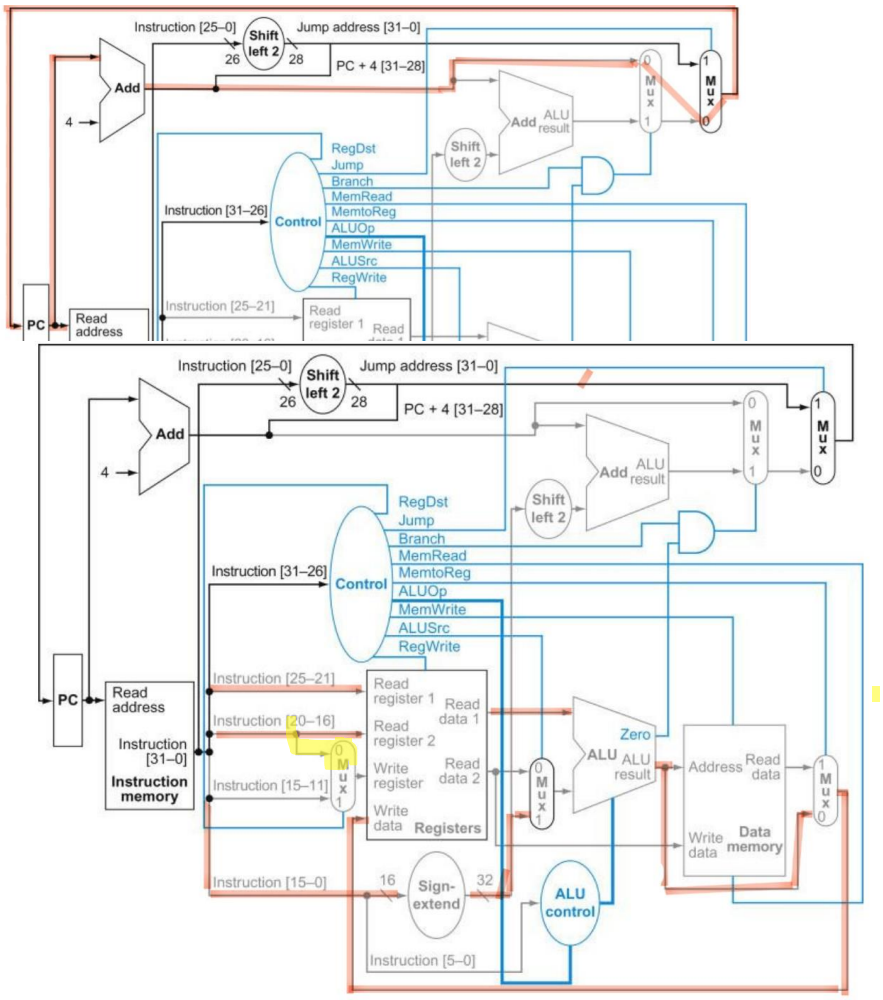
xori

$t = $s ^ ZE(i)

type : I / input condition 001110_xxxxxx_xxxxx

Control signal : 00_00_1_1_01_x0_00011_xxx_0_0_000_00_xxxxx

Port name	Bit	Func
RegDst	00	$rt : ALU의 연산 결과가 저장될 위치
RegDatSel	00	ALU의 결과값이 레지스터에 저장됨.
RegWrite	1	레지스터를 쓰기에 사용
SEUmode	1	Immediate value가 확장되어 32bit형태로 전달
ALUsrcB	01	Immediate value 사용
ALUctrl	0x	shift는 사용하지 않고, alu input은 그대로 입력
ALUop	00011	입력받은 값을 XOR연산
DataWidth	xxx	메모리를 사용하지 않기 때문에 xxx로 설정
MemWrite	0	메모리에 데이터입력 x
MemtoReg	0	alu 데이터를 reg값에 쓰기
Branch	000	pc + 4 값을 다음 주소로 설정
Jump	00	점프 x



Pc

Branch 명령어가 0이고 jump를 사용하지 않기 때문에 pc는 현재 주소값에 4가 더해진 주소로 이동

Registers

Reg1과 reg2에 대상 레지스터 할당. 쓰기 주소는 mux를 통해 $t주소로 설정됨. ALUB로 들어가는 값은 MUX를 통해 immdata와 reg2중에 imm데이터가 선택됨

Alu

연산의 결과를 저장 Aluctrl을 통해 어떠한 연산을 진행할지 정해지고 연산의 결과값은 register의 write data로 이동해 $t 주소에 저장된다.

slt

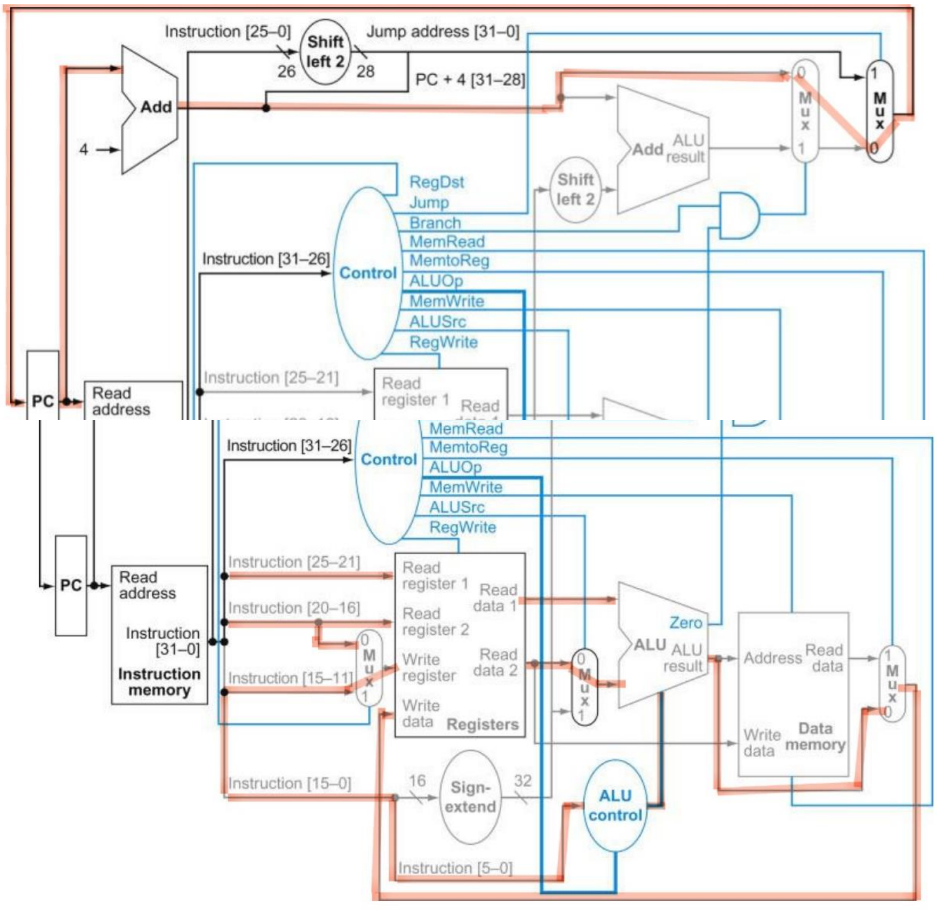
$d = ($s < $t) 해당 연산산이 참인지 거짓인지 저장

type : R / input condition 000000_101010_xxxxx

Control signal : 01_00_1_x_00_0x_10000_xxx_0_0_000_00_xxxxx

Port name	Bit	Func

RegDst	01	$d 목적지 레지스터에 값 저장.
RegDatSel	00	ALU의 결과값이 레지스터에 저장됨.
RegWrite	1	레지스터를 쓰기에 사용
SEUmode	x	확장 사용x
ALUsrcB	00	B주소 값을 사용
ALUctrl	0x	shift는 사용하지 않고, alu input은 그대로 입력
ALUop	10000	set less than
DataWidth	xxx	메모리를 사용하지 않기 때문에 xxx로 설정
MemWrite	0	메모리에 입력 x
MemtoReg	0	alu 데이터를 reg값에 쓰기
Branch	000	pc + 4 값을 다음 주소로 설정
Jump	00	점프 사용 x



PC

Branch 명령어가 0이고 jump를 사용하지 않기 때문에 pc는 현재 주소값에 4가 더해진 주소로 이동

Register

명령어로부터 각 연산에 사용될 레지스터 주소들을 입력받음. Write register는 mux

를 통해 $d값으로 설정됨.

Alu

Alu control에서 func와 Alu op를 이용해 alu에서 수행할 연산을 지정 연산 결과는 mux를 거쳐 registers의 write data로 이동해 $d에 값이 입력된다.

subu

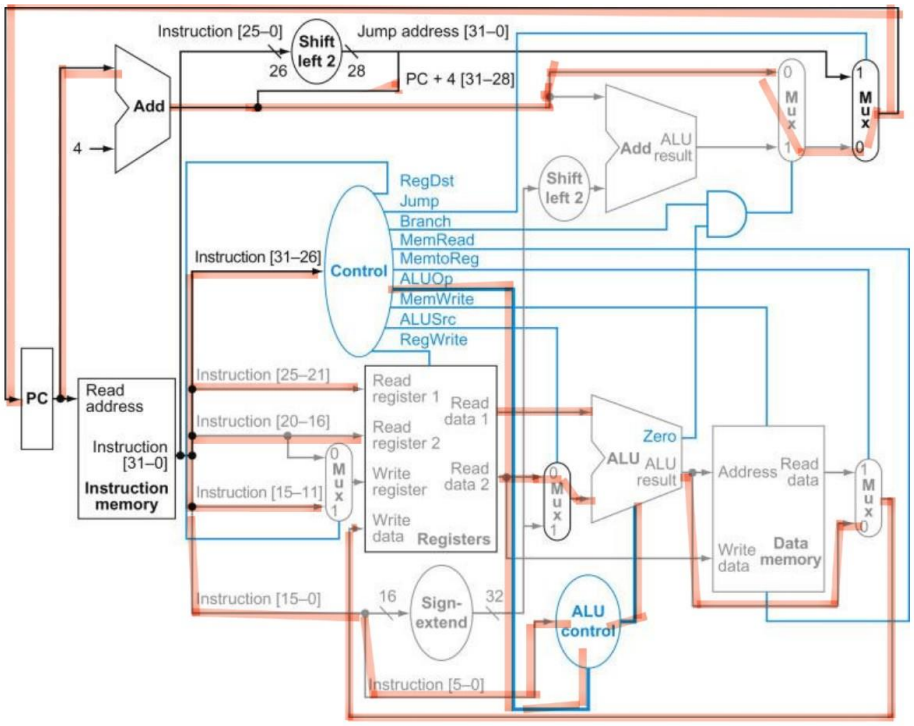
$d = $s - $t

$s - $t값을 $d레지스터에 저장

type : R / input condition 000000_100011_xxxxx

Control signal : 01_00_1_x_00_0x_00111_xxx_0_0_000_00_xxxxx

Port name	Bit	Func
RegDst	01	$rd 목적지 주소에 값 저장.
RegDatSel	00	ALU의 결과값이 레지스터에 저장됨.
RegWrite	1	레지스터를 쓰기에 사용
SEUmode	x	확장 사용x
ALUsrcB	00	B주소 값을 사용
ALUctrl	0x	shift는 사용하지 않고, alu input은 그대로 입력
ALUop	00111	부호를 고려하지 않고 a -b 수행
DataWidth	xxx	메모리에 데이터를 저장하지 않아 xxx로 설정
MemWrite	0	메모리 입력 x
MemtoReg	0	alu 데이터를 reg값에 쓰기
Branch	000	pc + 4 값을 다음 주소로 설정
Jump	00	점프 사용 x



PC

Branch 명령어가 0이고 jump를 사용하지 않기 때문에 pc는 현재 주소값에 4가 더해진 주소로 이동

Register

입력받은 명령어를 이용해 연산에 사용할 데이터가 존재하는 레지스터와 연산 값을 저장할 레지스터 지정.

ALU

Func과 ALUop를 이용해 ALU가 수행할 명령어 명시, $s - $t연산 결과를 register의 writedata로 전송 – 레지스터에 연산 결과 저장.

SRA

$d = $t >>> a

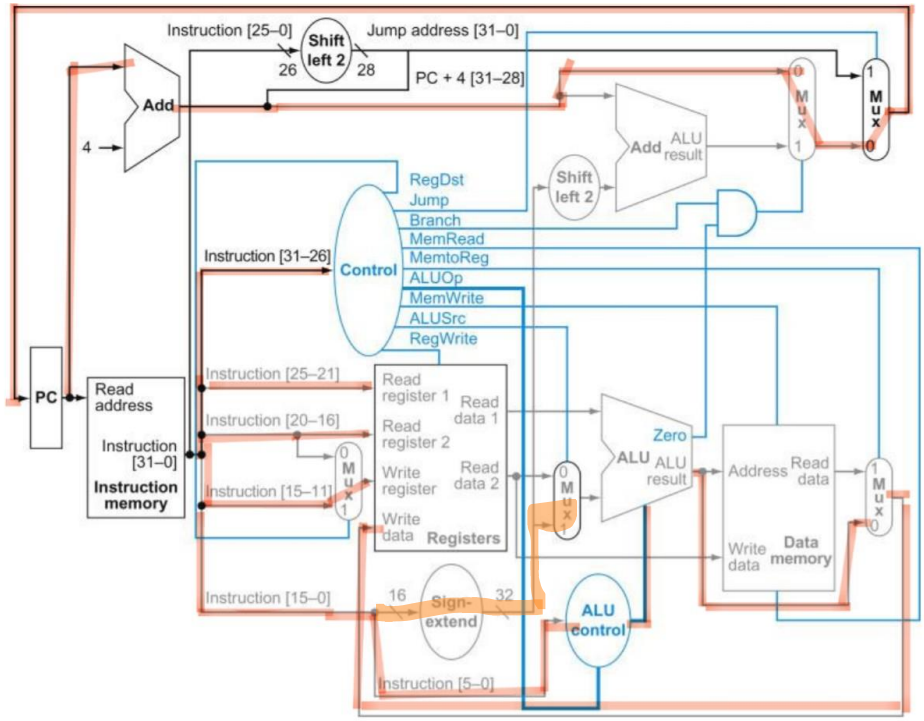
t값을 a값 만큼 logical shift

type : R / input condition 000000_000011_xxxxx

Control signal : 01_00_1_x_00_00_01111_xxx_0_0_000_00_xxxxx

Port name	Bit	Func
RegDst	01	$rd 목적지 레지스터에 연산 결과 저장
RegDatSel	00	ALU의 결과값이 레지스터에 저장됨.
RegWrite	1	레지스터를 쓰기에 사용
SEUmode	x	확장 사용x

ALUsrcB	00	B주소 값을 사용
ALUctrl	00	AB값을 그대로 사용, amout만큼 shift
ALUop	01111	Shift Right Arithmetic
DataWidth	xxx	메모리를 사용하지 않기 때문에 xxx로 설정
MemWrite	0	메모리 입력 x
MemtoReg	0	alu 데이터를 reg값에 쓰기
Branch	000	pc + 4 값을 다음 주소로 설정
Jump	00	점프 사용 x



PC

Branch 명령어가 0이고 jump를 사용하지 않기 때문에 pc는 현재 주소값에 4가 더해진 주소로 이동

Register

입력받은 명령어를 이용해 연산에 사용할 데이터가 존재하는 레지스터를 설정하고 mux를 이용해 $d를 쓰기에 이용할 레지스터로 설정 해당 명령어에선 $s주소가 사용되지 않는다.

ALU

Func과 ALUop를 이용해 ALU가 수행할 명령어 명시, $t값을 입력받은 명령어의 shamt값 만큼 shift right를 진행 연산 결과를 register의 write data로 전송 – 레지스터에 연산 결과 저장. Shamt값은 mux를 통해 선택된 후 alu로 입력 즉 reg2 값은 상관 없다.

MULTU

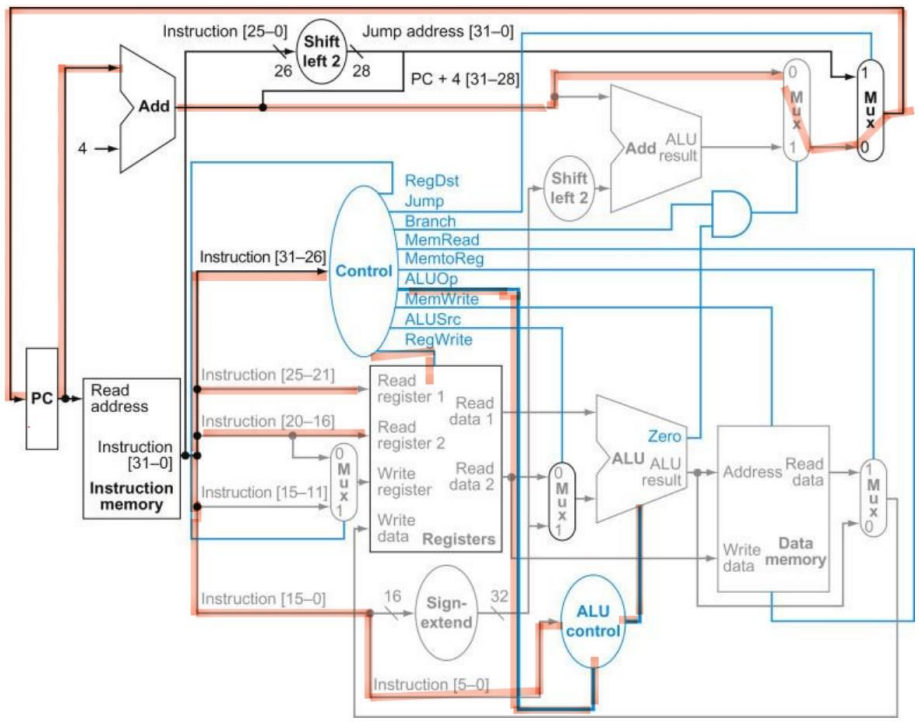
hi:lo = $s * $t

$s와 $t를 곱한 값을 hi와 lo에 저장.

type : R / input condition 000000_011001_xxxxx

Control signal : xx_00_0_x_00_00_01001_xxx_0_0_000_00_xxxxx

Port name	Bit	Func
RegDst	xx	레지스터에 값 저장 x
RegDatSel	00	ALU의 결과값이 곱셈기 내부 레지스터에 저장됨.
RegWrite	0	레지스터에 값 저장x
SEUmode	x	확장 x
ALUsrcB	00	B값을 alu 데이터로 받음
ALUctrl	0x	shift는 사용하지 않고, alu인풋은 그대로 사용
ALUop	01001	a * b
DataWidth	xxx	메모리를 사용하지 않기 때문에 xxx로 설정
MemWrite	0	메모리 입력x
MemtoReg	0	alu 데이터를 reg값에 쓰기
Branch	000	pc + 4 값을 다음 주소로 설정
Jump	00	점프 사용 x



PC

Branch 명령어가 0이고 jump를 사용하지 않기 때문에 pc는 현재 주소값에 4가 더해진 주소로 이동

Register

두 개의 레지스터를 읽어와 해당 값을 alu로 넘겨준다. 목적지. 레지스터에 연산결과를 저장하지 않고 MULU에 포함되어 있는 HI와 LO에 연산 결과값을 저장하기 때문에 write는 Enable하나 목적지 레지스터는 따로 지정하지 않는다. 테스트 벤치를 통해 값을 확인가능하다.

ALU

입력값 A B를 곱연산 한 뒤 해당 결과값을 결과를 각각 HI LO 레지스터에 저장한다.

MFLO

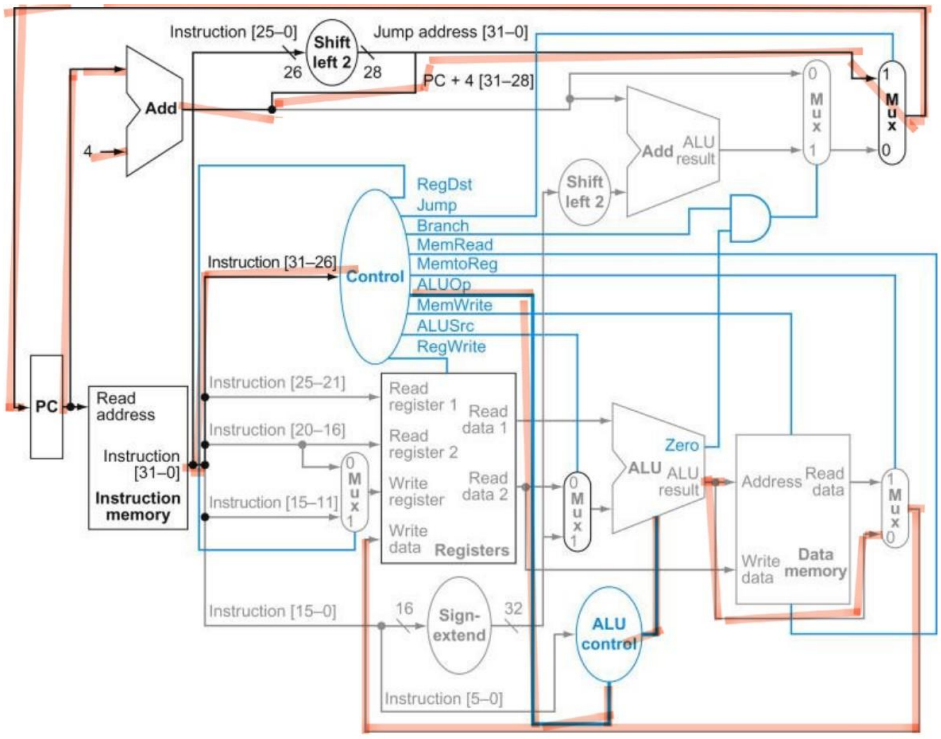
$d = lo

$d레지스터에 이전 곱셈 결과의 low값을 입력

type : R / input condition 000000_010010_xxxxx

Control signal : 01_01_1_x_xx_xx_10011_xxx_0_0_000_00_xxxxx

Port name	Bit	Func
RegDst	01	$d레지스터에 값 저장
RegDatSel	01	ALU의 결과값이 곱셈기 내부 레지스터에 저장됨.
RegWrite	1	레지스터에 값 저장
SEUmode	x	확장 x
ALUsrcB	xx	이전 결과를 사용하기 때문에 현재 연산 x
ALUctrl	xx	alu연산 x
ALUop	10011	lo값을 읽어옴
DataWidth	xxx	메모리를 사용하지 않기 때문에 xxx로 설정
MemWrite	0	메모리 입력x
MemtoReg	0	alu 데이터를 reg값에 쓰기
Branch	000	pc + 4 값을 다음 주소로 설정
Jump	00	점프 사용 x



PC

Branch 명령어가 0이고 jump를 사용하지 않기 때문에 pc는 현재 주소값에 4가 더해진 주소로 이동

Register

곱셈기내부 레지스터인 lo의 값을 읽어와 $d주소에 저장한다.

ALU

이전 곱연산에서 저장된 결과값 중 LO부분을 가져와 레지스터에 저장한다.

SB

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

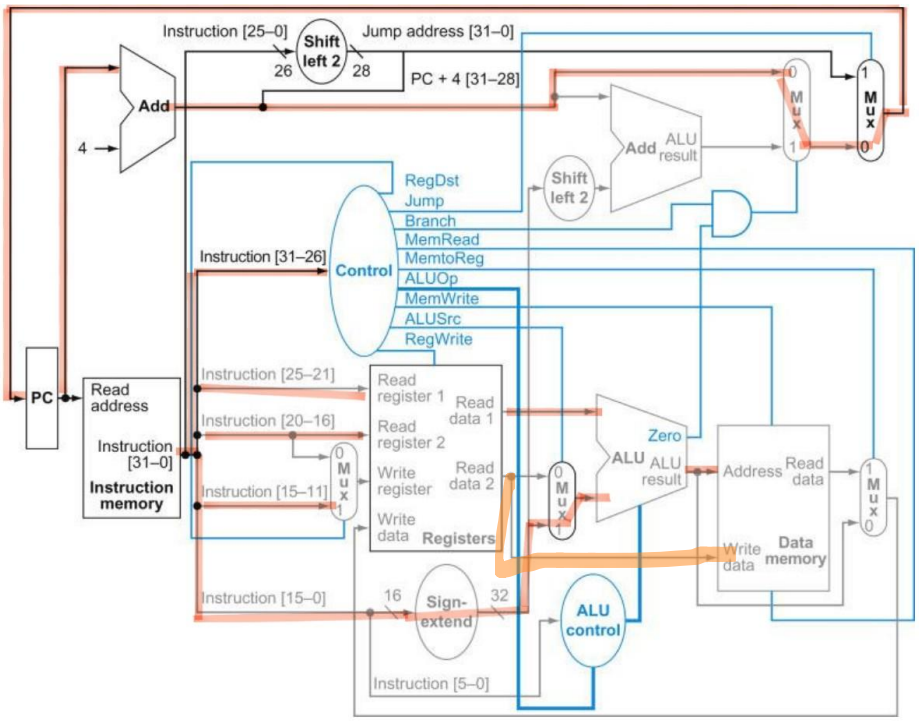
$t레지스터에서 1byte크기의 데이터를 읽어와 메모리에 저장

type : i / input condition 101000_xxxxxx_xxxxx

Control signal : xx_xx_0_1_01_0x_00100_011_1_x_000_00_xxxxx

Port name	Bit	Func
RegDst	xx	레지스터에 값 저장x
RegDatSel	xx	레지스터에 값을 쓰지 않는다.
RegWrite	0	레지스터 입력 x
SEUmode	1	Immediate value 32bit로 확장 후 전달
ALUsrcB	01	Immediate value 사용
ALUctrl	0x	shift는 사용하지 않고, alu input은 그대로 입력
ALUop	00100	a+b 메모리 주소 계산에 이용

DataWidth	011	8bit데이터 저장
MemWrite	1	메모리 입력
MemtoReg	x	연산 결과를 저장하지 않음
Branch	000	pc + 4 값을 다음 주소로 설정
Jump	00	점프 사용 x



PC

Branch 명령어가 0이고 jump를 사용하지 않기 때문에 pc는 현재 주소값에 4가 더해진 주소로 이동

Register

$s메모리 주소에 $t1값을 byte만큼 저장하는 명령어로 reg1 과 reg2에는 각각 메모리 주소가 담긴 레지스터가 들어가게 된다. 레지스터에 값을 입력하지 않기 때문에 reg wirte는 0이되고 목적지 주소 또한 따로 지정하지 않는다.

ALU

레지스터로부터 받은 주소값에 I Immediate value 값을 더해 메모리 주소를 지정하기 때문에 mux에서 extend를 거친 Immediate value 값이 들어오며 해당 값은 alu에서 합 연산을 통해 메모리의 adderss로 들어간다.

Memory

입력받은 메모리 주소값(reg1 + Immediate value)에 reg2 주소로부터 읽어들인 값을 저장한다.

LHU

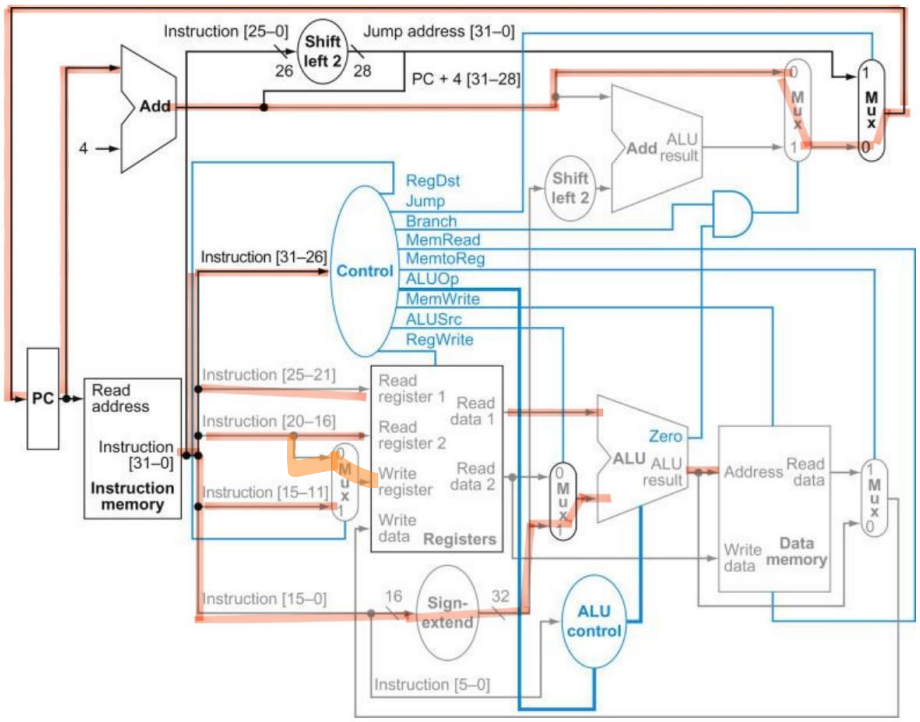
$t = ZE (MEM [$s + i]:2)

$s + i 메모리로부터 데이터를 읽어와 $t에 저장.

type : i / input condition 100101_xxxxxx_xxxxx

Control signal : 00_00_1_1_01_0x_00100_010_0_1_000_00_xxxxx

Port name	Bit	Func
RegDst	00	$rt : ALU의 연산 결과가 저장될 위치
RegDatSel	00	ALU의 결과값이 레지스터에 저장됨.
RegWrite	1	레지스터를 쓰기에 사용
SEUmode	x	Immediate value 32bit로 확장
ALUsrcB	01	Immediate value 사용
ALUctrl	0x	shift는 사용하지 않고, alu input은 그대로 입력
ALUop	00100	a + b 메모리 주소 계산에 사용
DataWidth	010	half word데이터
MemWrite	0	메모리에 값 저장 x
MemtoReg	1	메모리 데이터를 레지스터에 저장
Branch	000	pc + 4 값을 다음 주소로 설정
Jump	00	점프 사용 x



PC

Branch 명령어가 0이고 jump를 사용하지 않기 때문에 pc는 현재 주소값에 4가 더해진 주소로 이동

Register

레지스터에 $s $t번지 데이터를 읽어서 연산 수행 $t레지스터는 write를 위해 사용하기 때문에 mux로 들어가 src에 의해 선택된다. $레지스터의 값은 메모리 주소를 찾는데 이용한다.

ALU

Register로부터 들어온 $s레지스터 주소에 Immediate value를 더해 메모리 주소를 선택하고 해당 메모리에서 half word만큼의 데이터를 불러와 레지스터의 data wirte로 넘겨주어 $t에 값이 저장되는 역할을 수행한다.

BLEZ

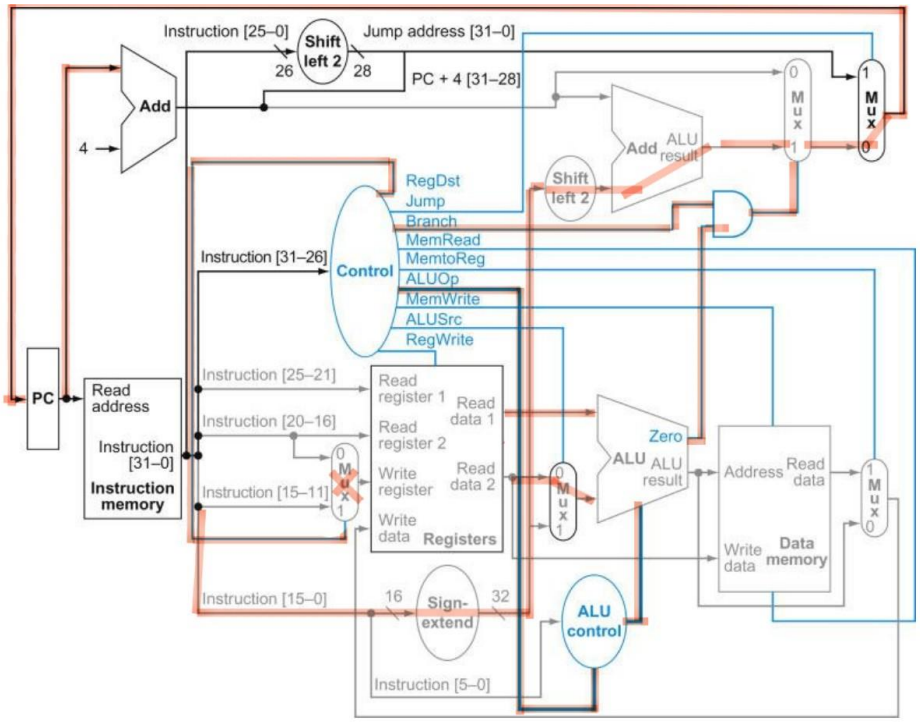
if ($s <= 0) pc += i << 2

$s값이 0보다 작거나 같을 경우 pc의 주소를 Immediate value 값 + 4 만큼 이동

type : i / input condition 000110_xxxxxx_xxxxx

Control signal : xx_xx_0_1_10_00_00100_xxx_x_0_110_00_xxxxx

Port name	Bit	Func
RegDst	xx	레지스터에 값 저장 x
RegDatSel	xx	레지스터에 값 저장 x
RegWrite	0	레지스터에 값 저장 x
SEUmode	0	immediate value 32 bit로 확장
ALUsrcB	10	a+0을 통해 부호를 확인하기 때문에 0 설정
ALUctrl	00	shift사용 ,alu연산에 ab 값 그대로 이용
ALUop	00100	a + b 수행
DataWidth	xxx	메모리를 사용하지 않기 때문에 xxx로 설정
MemWrite	x	메모리에 데이터를 쓰지 않는다.
MemtoReg	0	alu 데이터를 reg값에 쓰기
Branch	110	값이 참이 아닌 경우 데이터 이동
Jump	00	점프 사용 x



PC

ALU의 연산 결과에 따라 다음 pc 의 주소를 지정 alu로부터 A + 0이 연산되는데 해당 값이 0보다 작은 경우 mux로 1이 들어가 Immediate value 값 만큼 이동된 pc 주소가 입력 값이 0보다 큰 경우 0이 선택되어 pc + 4값이 pc로 load Immediate value 값에 <<shift를 2번 하게 되는데 byte단위로 pc주소를 변경하기 위해서 이다.

Register

연산에 사용될 레지스터 주소만을 가져옴 $s해당 값을 Alu로 보내 0보다 작거나 같 같은 확인.

ALU

$s레지스터 값과 0의 합연산을 수행 해당 결과 값이 0<=인 경우 positive값인 0이 입력되며 branch에서 not positive일 때 pc를 Immediate value 만큼 이동하도록 한다.

Jr

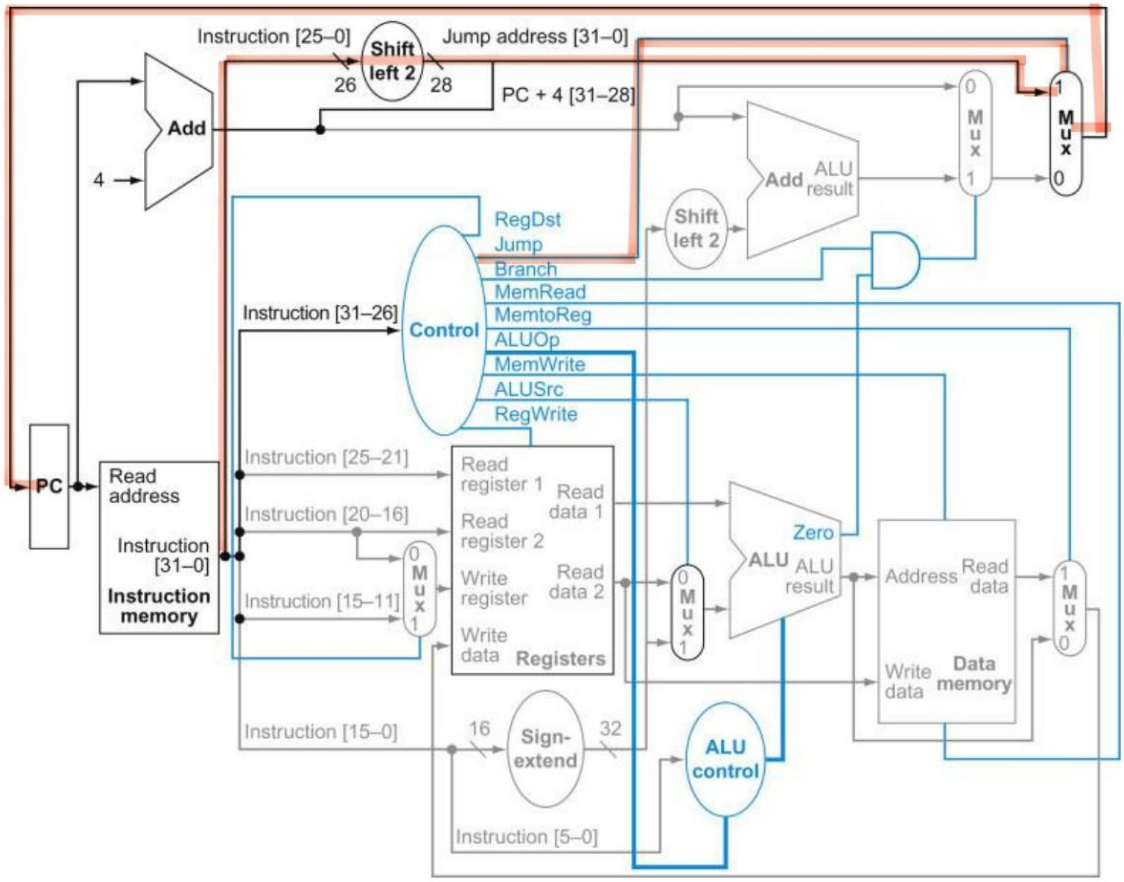
pc = $s pc주소를 레지스터 값으로 변경

type : R / input condition 000110_xxxxxx_xxxxx

Control signal : xx_xx_x_1_xx_xx_xxxxx_xxx_x_x_000_10_xxxxx

Port name	Bit	Func
RegDst	xx	레지스터 사용 x
RegDatSel	xx	레지스터 사용 x
RegWrite	x	레지스터 사용 x
SEUmode	1	부호 확장 필요
ALUsrcB	xx	alu연산을 사용하지 않기 때문에 xx

ALUctrl	xx	alu연산을 사용하지 않기 때문에 xx
ALUop	xxxxx	alu연산을 사용하지 않기 때문에 xx
DataWidth	xxx	메모리를 사용하지 않기 때문에 xxx로 설정
MemWrite	x	메모리를 사용하지 않기 때문에 xxx로 설정
MemtoReg	x	메모리를 사용하지 않기 때문에 xxx로 설정
Branch	xxx	
Jump	10	



PC

$s레지스터로부터 읽어온 값을 이용해 pc주소를 변경

Register

$s의 데이터 만을 필요로 하기 때문에 하나의 레지스터만 읽으면 된다.

ALU

기능 수행 x

3. 결과

괄호는 예상 결과값 입니다.

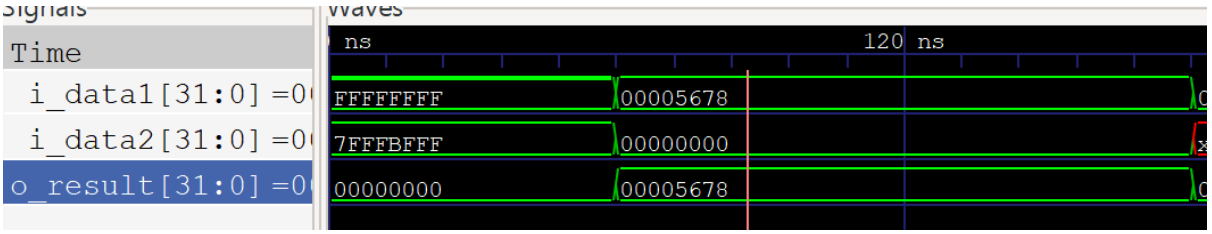
```
//i opcode 6 rs 5 rt 5 im 16
//r opcpde 6 rd 5 rs 5 rt 5 shamt 5 func 6
00110100_00000010_00000000_00000100 // ori $2 | 0x0004 -> 0000004
00110100_00000011_01010110_01111000 // ori $3 | 0x5678 -> 00005678
00111100_00000101_11111111_11111111 // lui $5, 0xFFFF0000 삽입
00110100_10100101_11111111_11111111 // lui $5| 0xFFFF -> FFFFFFFF
00111100_00000110_01111111_11111111 // ori $6, 0x7FFF // 상수 삽입
00110100_11000110_10111111_11111111 // ori $6, 0x0000BFFF // 상수 삽입

//초기 레지스터 세팅
//$2 : 0x0000_0004
//$3 : 0x0000_5678
//$5 : 0xFFFF_FFFF
//$6 : 0x7FFF_BFFF
```

초기 레지스터 값 세팅

XORI

```
00111000_01101011_00000000_00000000 // $3 xori 0 -> $11(00005678)
```

ALU



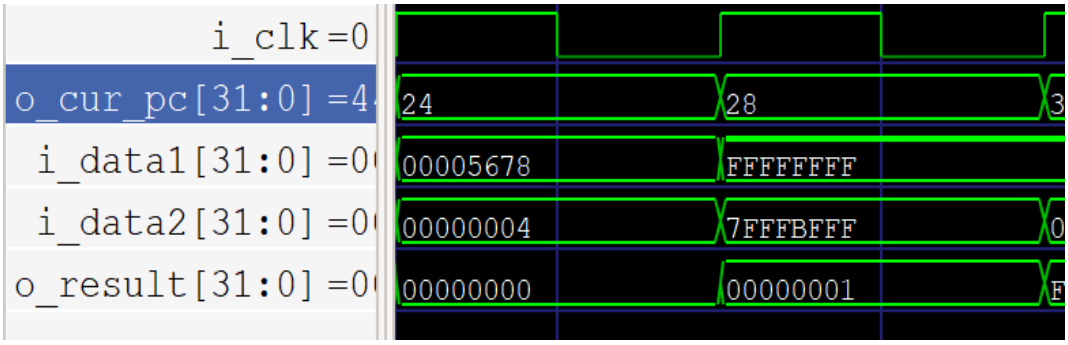
레지스터 저장 값

```
00000000 00000000 00000000 00000000 : 00000000
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
00000000 00000000 00000000 00000100 : 00000004
00000000 00000000 01010110 01111000 : 00005678
00000000 00000000 00000000 00000000 : 00000000
11111111 11111111 11111111 11111111 : ffffffff
01111111 11111111 10111111 11111111 : 7fffbfff
00000000 00000000 00000000 00000001 : 00000001
11111111 11111111 10101001 10000111 : ffffa987
00000000 00000000 00001010 11001111 : 00000acf
10000000 00000000 01000000 00000001 : 80004001
00000000 00000000 01010110 01111000 : 00005678
```

SLT

```
000000_00011_00010_00100_00000_101010 // $3 < $2? => $4(0)
000000_00101_00110_00111_00000_101010 // $5 < $6? => $7(1)
```

ALU연산 결과



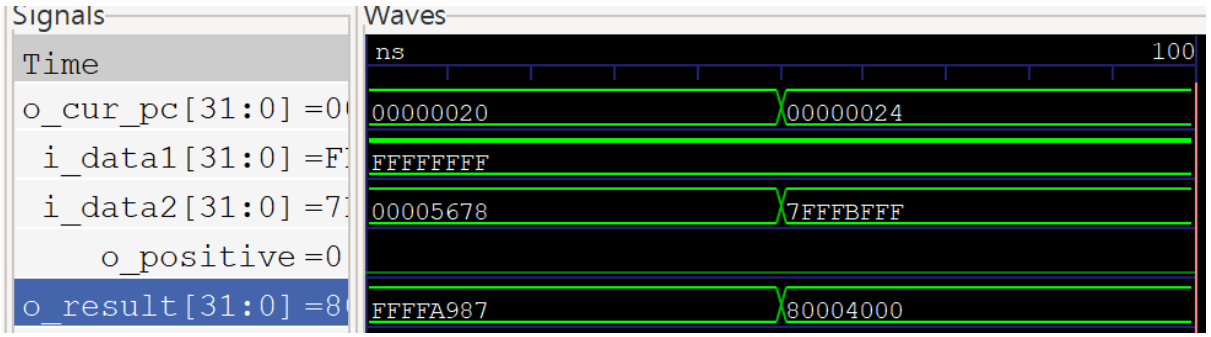
레지스터 저장 값

```
F   E   V
00000000 00000000 00000000 00000000 : 00000000
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
00000000 00000000 00000000 00000100 : 00000004
00000000 00000000 01010110 01111000 : 00005678
00000000 00000000 00000000 00000000 : 00000000
11111111 11111111 11111111 11111111 : ffffffff
01111111 11111111 10111111 11111111 : 7fffbfff
00000000 00000000 00000000 00000001 : 00000001
```

SUBU

```
000000_00101_00011_01000_00000_100011 // $5-$3 = $8(FFFF A987)
000000_00101_00110_01000_00000_100011 // $5-$6 = $8(8000 4000)
```

ALU gtk wave



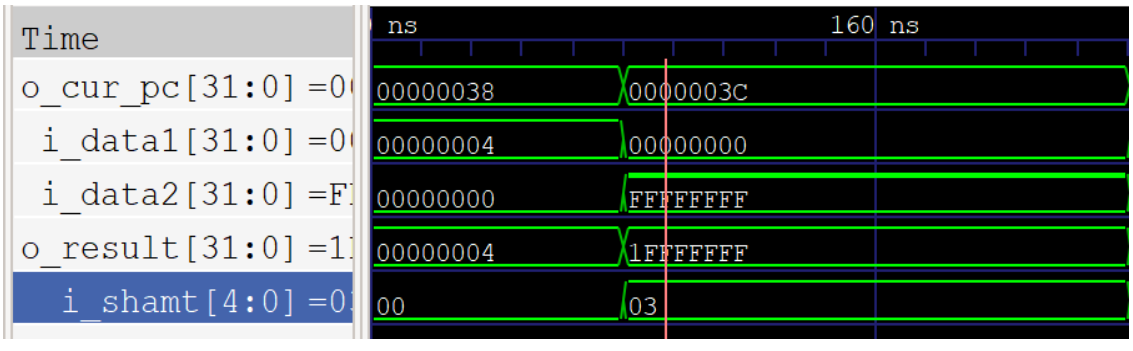
레지스터 저장값 2번째 결과 값만 저장함

```
00000000 00000000 00000000 00000000 : 00000000
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
00000000 00000000 00000000 00000100 : 00000004
00000000 00000000 01010110 01111000 : 00005678
00000000 00000000 00000000 00000000 : 00000000
11111111 11111111 11111111 11111111 : ffffffff
01111111 11111111 10111111 11111111 : 7fffbfff
00000000 00000000 00000000 00000001 : 00000001
10000000 00000000 01000000 00000000 : 80004000
```

SRA

```
00000000_00000011_01001000_11000011 // shift $5>>>3 = 9$(1FFFFFFF)
```

ALU gtk wave



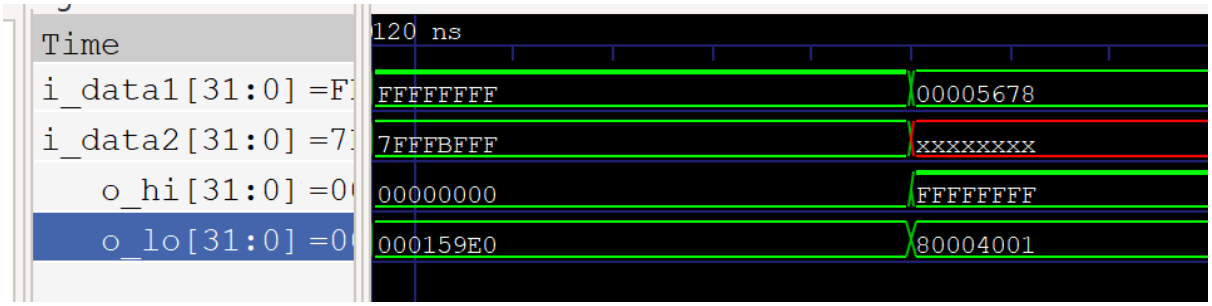
레지스터 저장값

```
00000000 00000000 00000000 00000000 : 00000000
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
00000000 00000000 00000000 00000100 : 00000004
00000000 00000000 01010110 01111000 : 00005678
00000000 00000000 00000000 00000000 : 00000000
11111111 11111111 11111111 11111111 : ffffffff
01111111 11111111 10111111 11111111 : 7fffbfff
00000000 00000000 00000000 00000001 : 00000001
10000000 00000000 01000000 00000000 : 80004000
00011111 11111111 11111111 11111111 : 1fffffff
```

MULTU

```
00000000_01000011_00000000_00011001// $2 * $3 ( hi: 0 lo: 0x000159E0)
00000000_10100110_00000000_00011001// $5 * $6 (hi: 7FFF lo: 80004001)
```

MULT동작



레지스터에 저장 X

MFLO

```
00000000_00000000_01010000_00010010 // FFFFFFF*7FFFBFFF result of low $10(8001)
```

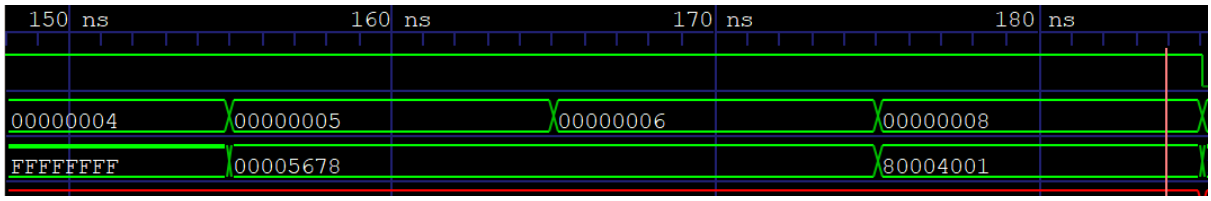
레지스터 저장 값

```
00000000 00000000 00000000 00000000 : 00000000
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
00000000 00000000 00000000 00000100 : 00000004
00000000 00000000 01010110 01111000 : 00005678
00000000 00000000 00000000 00000000 : 00000000
11111111 11111111 11111111 11111111 : ffffffff
01111111 11111111 10111111 11111111 : 7fffbfff
00000000 00000000 00000000 00000001 : 00000001
10000000 00000000 01000000 00000000 : 80004000
00011111 11111111 11111111 11111111 : 1fffffff
10000000 00000000 01000000 00000001 : 80004001
00000000 00000000 01010110 01111000 : 00005678
```

SB

```
10100000_01000101_00000000_00000000 // FF -> MEM$(4)
10100000_01000011_00000000_00000001 // 78 -> MEM$(4+1
10100000_01000011_00000000_00000010 // 78 -> MEM$(4+2
10100000_01001010_00000000_00000100 // FF -> MEM$(4 +4)
```

MEM 동작 i_data중 하위 1byte를 읽어와 메모리의 i_addr주소에 저장한다.



메모리 상태

메모리 주소는 오른쪽부터 2bit씩 0 1 2 3 순이다.

```
xxxxxxxx01111000011110001111111 xx7878ff
xxxxxxxxxxxxxxxxxxxxxxxx00000001 xxxxxx01
```

`

LHU

```
10100000_01000101_00000000_00000000 // FF -> MEM$(4)
10100000_01000011_00000000_00000001 // 78 -> MEM$(4+1
10100000_01000011_00000000_00000010 // 78 -> MEM$(4+2
10100000_01001010_00000000_00000100 // FF -> MEM$(4 +4)

00000000_00000101_01001000_11000011 // shift $5>>>3 = 9$(1FFFFFFF)

10010100_01001101_00000000_00000000//MEM$2(4) ->$13
10010100_01001110_00000000_00000010//MEM$2(4+2)->$14
10010100_01001111_00000000_00000100//MEM$2(4+4)->$15
```

mem동작

메모리 저장 값

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxxxxxxx 0
xxxxxxxx01111000011110001111111 1xx7878ff 4
xxxxxxxxxxxxxxxxxxxxxxxx00000001 xxxxxx01 8
```

레지스터 저장 결과

```
00000000 00000000 01111000 11111111 : 000078ff
00000000 00000000 xxxxxxxx 01111000 : 0000xx78
00000000 00000000 xxxxxxxx 00000001 : 0000xx01
```

BLEZ

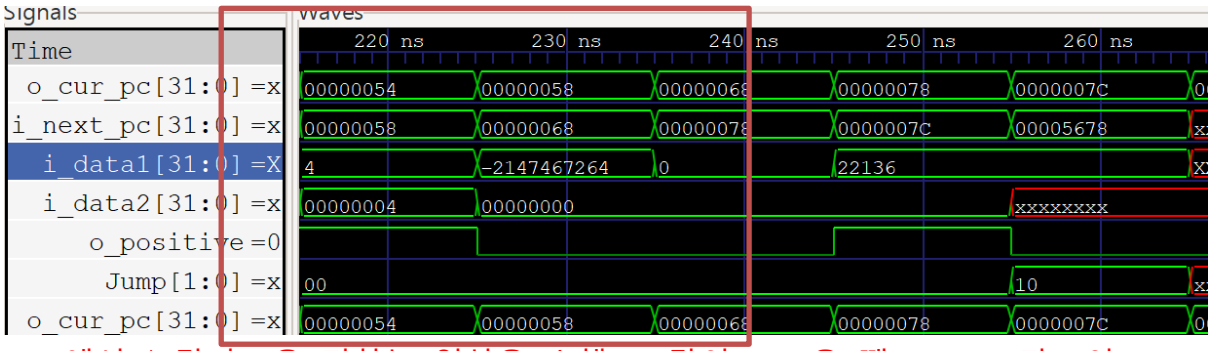
명령어

00011001_00000000_00000000_00000011// if($8<0)pc = pc+4+3<<2
XXXXXXXX_XXXXXXXX_XXXXXXXX_XXXXXXXX
XXXXXXXX_XXXXXXXX_XXXXXXXX_XXXXXXXX
XXXXXXXX_XXXXXXXX_XXXXXXXX_XXXXXXXX

00011000_10000000_00000000_00000011// if($4<0)pc = pc+4+3<<2
XXXXXXXX_XXXXXXXX_XXXXXXXX_XXXXXXXX
XXXXXXXX_XXXXXXXX_XXXXXXXX_XXXXXXXX
XXXXXXXX_XXXXXXXX_XXXXXXXX_XXXXXXXX

00011001_01100000_00000000_00000011// if($11<0)pc = pc+4+3<<2

PC 동작



ALU에서 $s값과 0을 더하는 연산을 수행 alu값이 0<=을 때 positive가 0이 되고 positive가 0일 때 branch명령어가 작동. -> pc값 pc +=3<<2

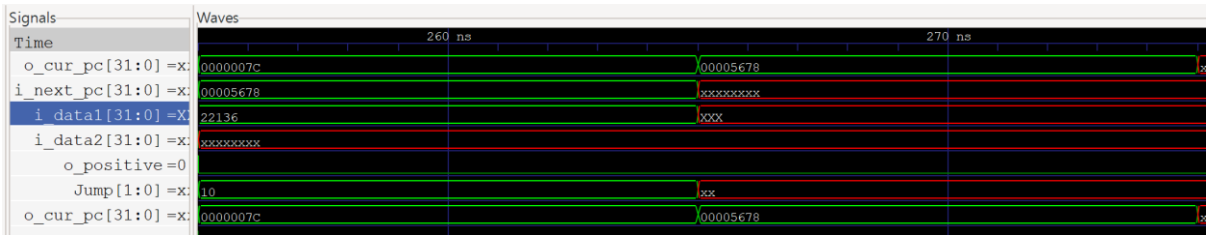
아닌 경우 pc + 4수행

,

JR

00000001_01100000_00000000_00001000// pc = $11

PC 동작



Pc의 값이 $s값인 5678로 변화

4. 고찰

본 프로젝트에선 single cycle cpu인 mips의 명령어들을 구현해 보았다.

And pla의 경우 각 op코드와 명령어 타입에 따라 값을 적으면 되기 때문에 비교적 쉬웠지만. Or pla에서 어려움을 겪었다. 먼저 해당 이진코드가 어떤 모듈을 제어하는지 알아야 했고 모듈 내에서 이진코드가 어떻게 동작되는지 이해해야 완성할 수 있기 때문이다. 명령어 별로 사용하지 않는 모듈들이 했다. 예를들어 xori 명령어의 경우 memory는 사용하지 않기 때문에 memory관련 명령어들은 0으로 채웠는데 opcode를 제대로 읽지 못하는 결과가 발생했다. 해당 결과의 원인으로는 0으로 채운 명령어에서 0을 읽어들이고 또 하나의 동작을 했기 때문에 신호가 섞여 불분명한 값이 출력된 것을 확인했다. 따라서 사용하지 않는 모듈의 명령어는 x로 값을 입력해 해당 모듈이 동작하지 않도록 설정해 해결 할 수 있었다.