컴퓨터구조실험 보고서

Project #1 - MIPS Single Cycle CPU

과 목 컴퓨터구조실험

담당교수 이성원교수님

학 과 컴퓨터정보공학부

학 번 2021202058

이 름 송채영

제 출 일 2021. 04. 19

1. Introduction

이번 프로젝트는 MIPS Architecture 를 기반으로 하는 CPU 인 MIPS Single Cycle CPU 를 설계하는 것이다. 즉 한 cycle 안에 명령어를 수행하며 한 clock cycle 에 모든 명령어 실행 단계를 완료하므로 명령어의 실행 시간이 동일하다. MIPS Single Cycle CPU 는 Instruction Memory, Register File, ALU, Data Memory, Control Unit 으로 구성되어 있다. 먼저 Instruction Memory 는 명령어를 저장하는 메모리이고, Register File 은 명령어에서 사용되는 레지스터 값을 저장하는 메모리이다. 다음으로 ALU 는 산술 연산과 논리연산을 수행하고, Data Memory 는 데이터를 저장하는 메모리이다. 마지막으로 Control Unit 은 명령어 실행을 제어한다. 동작방식에 대해서도 설명해보면 다음과 같다. 먼저 명령어를 Instruction Memory 에서 가져와 Register File 에서 필요한 레지스터 값을 가져온다. 필요한 연산은 ALU 에서 수행하고 결과 값을 Register File 에 저장한다. 명령어에서 필요한 데이터를 Data Memory 에서 가져오고 필요한 데이터를 ALU 에서 처리한 후 결과 값을 Data Memory 에 저장한다. Control Unit 은 각 명령어를 실행할 때마다 다음 단계를 결정한다. 마지막으로 프로젝트에서 사용하는 MIPS Instruction 에 대해 설명해보겠다. ADDU – Add Unsigned 로, 두개의 레지스터 값인 rs 와 rt 를 더하고 결과를 레지스터 rd 에 저장한다. 부호 없는 정수를 더하는데 사용되는 명령어이다.

OR – Bitwise Or 로, 두개의 레지스터 값 rs 와 rt 에 대해 비트 OR 연산을 수행하고 결과를 레지스터 rd 에 저장한다. 두 비트열의 비트 OR 연산을 수행하는데 사용된다.

ADDIU – Add Immediate Unsigned 로, 레지스터 rs 와 16 비트 상수값의 합을 레지스터 rt 에 저장한다. rs 와 상수 값을 더하는데 사용된다.

XORI – XOR Immediate 로, 레지스터 rs 와 16 비트 상수 값의 비트 XOR 결과를 레지스터 rt 에 저장한다. rs 와 상수 값을 XOR 연산하는데 사용된다.

SLL – Shift Left Logical 로, 레지스터 rt 의 비트열을 왼쪽으로 shift 하고 shift 된 결과를 레지스터 rd 에 저장한다. 레지스터의 비트열을 왼쪽으로 shift 하는데 사용된다.

SRAV – Shift Right Arithmetic Variable 로, 레지스터 rt 의 비트열을 레지스터 rs 에서 지정한 비트 수만큼 오른쪽으로 shift 하고 shift 된 결과를 레지스터 rd 에 저장한다. 레지스터의 비트열을 오른쪽으로 shift 하는데 사용된다.

SH – Store Halfword 로, 레지스터 rt 의 하위 16bit 를 메모리 주소에서 2byte 로 저장한다. 변수의 하위 16bit 값을 메모리에 저장하는데 사용된다.

LH – Load Halfword 로, 메모리의 주소에서 2byte 값을 읽고 그 값을 레지스터 rt 에 load 한다. 메모리에서 하위 16bit 값을 읽어 레지스터에 저장하는데 사용된다.

BLTZ – Branch on Less Than Zero 로, 레지스터 rs 의 값이 0 보다 작으면 label 로 분기한다. 조건 분기를 수행하는데 사용된다.

JAL – Jump and Link 로, PC 를 목표 주소로 변경하면서 다음 명령어의 주소를 레지스터 $31 에 저장한다. 함수 호출에서 많이 사용되며 함수 호출 이전의 주소를 저장하는데 사용된다.

2. 결과 화면

우선 PLA 는 Programmable Logic Array 로 하드웨어 구성 요소 중 하나이다. PLA 는 입력을 AND gate 와 OR gate 로 구성된 matrix 에 연결하는 방식으로 구성된다.

```
000000_000000_xxxxx  // 0x00 : sll
xxxxxx_xxxxxx_xxxxx  // 0x01 : srl
xxxxxx_xxxxxx_xxxxx  // 0x02 : sra
xxxxxx_xxxxxx_xxxxx  // 0x03 : sllv
xxxxxx_xxxxxx_xxxxx  // 0x04 : srlv
000000_000111_xxxxx  // 0x05 : srav
xxxxxx_xxxxxx_xxxxx  // 0x06 : jr
xxxxxx_xxxxxx_xxxxx  // 0x07 : jalr
xxxxxx_xxxxxx_xxxxx  // 0x08 : break
xxxxxx_xxxxxx_xxxxx  // 0x09 : mfhi
xxxxxx_xxxxxx_xxxxx  // 0x0a : mthi
xxxxxx_xxxxxx_xxxxx  // 0x0b : mflo
xxxxxx_xxxxxx_xxxxx  // 0x0c   mtlo
xxxxxx_xxxxxx_xxxxx  // 0x0d   mult
xxxxxx_xxxxxx_xxxxx  // 0x0e   multu
xxxxxx_xxxxxx_xxxxx  // 0x0f   div
xxxxxx_xxxxxx_xxxxx  // 0x10   divu
xxxxxx_xxxxxx_xxxxx  // 0x11   add
000000_100001_xxxxx  // 0x12   addu
xxxxxx_xxxxxx_xxxxx  // 0x13   sub
xxxxxx_xxxxxx_xxxxx  // 0x14   subu
xxxxxx_xxxxxx_xxxxx  // 0x15   and
000000_100101_xxxxx  // 0x16   or
xxxxxx_xxxxxx_xxxxx  // 0x17   xor
xxxxxx_xxxxxx_xxxxx  // 0x18   nor
xxxxxx_xxxxxx_xxxxx  // 0x19   slt
xxxxxx_xxxxxx_xxxxx  // 0x1a   sltu
000001_xxxxxx_00000  // 0x1b   bltz
xxxxxx_xxxxxx_xxxxx  // 0x1c   bgez
xxxxxx_xxxxxx_xxxxx  // 0x1d   j
000011_xxxxxx_xxxxx  // 0x1e   jal
xxxxxx_xxxxxx_xxxxx  // 0x1f   beq
xxxxxx_xxxxxx_xxxxx  // 0x20   bne
xxxxxx_xxxxxx_xxxxx  // 0x21   blez
xxxxxx_xxxxxx_xxxxx  // 0x22   bgtz
xxxxxx_xxxxxx_xxxxx  // 0x23   addi
001001_xxxxxx_xxxxx  // 0x24   addiu
xxxxxx_xxxxxx_xxxxx  // 0x25   slti
xxxxxx_xxxxxx_xxxxx  // 0x26   sltiu
xxxxxx_xxxxxx_xxxxx  // 0x27   andi
xxxxxx_xxxxxx_xxxxx  // 0x28   ori
001110_xxxxxx_xxxxx  // 0x29   xori
xxxxxx_xxxxxx_xxxxx  // 0x2a   lui
xxxxxx_xxxxxx_xxxxx  // 0x2b   lb
100001_xxxxxx_xxxxx  // 0x2c   lh
xxxxxx_xxxxxx_xxxxx  // 0x2d   lw
xxxxxx_xxxxxx_xxxxx  // 0x2e   lbu
xxxxxx_xxxxxx_xxxxx  // 0x2f   lhu
xxxxxx_xxxxxx_xxxxx  // 0x30   sb
101001_xxxxxx_xxxxx  // 0x31   sh
xxxxxx_xxxxxx_xxxxx  // 0x32   sw
```

PLA_AND.txt 파일이다. PLA_AND 는 입력 신호들의 AND 연산 결과를 계산한다. 입력 조건을 특정하고 이 조건이 충족되면 출력 신호를 생성해 전체 회로의 동작을 제어한다. 구현한 10 개의 Instruction 을 표로 나타내면 다음과 같다.

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

BLTZ 명령어는 레지스터 값이 0 보다 작을 경우 지정된 label 로 분기하는데, 이때 사용되는 regimm 값을 00000 으로 주어 레지스터의 값이 0 보다 작은 지 체크하기 위해서이다. 즉 0 보다 작아야 분기가 발생하기 때문에 regimm 을 00000 으로 설정해 주었다.

```
01_00_1_x_00_00_01101_xxx_0_0_000_00_xxxxx  // 0x00 : sll   $d = $t << a
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x01 : srl   $d = $t >> a
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x02 : sra   $d = $t >>> a
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x03 : sllv  $d = $t << $s
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x04 : srlv  $d = $t >> $s
01_00_1_x_00_01_01111_xxx_0_0_000_00_xxxxx  // 0x05 : srav  $d = $t >>> $s
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x06 : jr    pc = $s
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x07 : jalr  $31 = pc; pc = $s
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x08 : break
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x09 : mfhi  $d = hi
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x0a : mthi  hi = $s
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x0b : mflo  $d = lo
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x0c   mtlo  lo = $s
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x0d   mult  hi:lo = $s * $t
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x0e   multu hi:lo = $s * $t
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x0f   div   lo = $s / $t; hi = $s % $t
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x10   divu  lo = $s / $t; hi = $s % $t
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x11   add   $d = $s + $t
01_00_1_x_00_0x_00101_xxx_0_0_000_00_xxxxx  // 0x12   addu  $d = $s + $t
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x13   sub   $d = $s - $t
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x14   subu  $d = $s - $t
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x15   and   $d = $s & $t
01_00_1_x_00_0x_00001_xxx_0_0_000_00_xxxxx  // 0x16   or    $d = $s | $t
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x17   xor   $d = $s ^ $t
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x18   nor   $d = ~($s | $t)
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x19   slt   $d = ($s < $t)
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x1a   sltu  $d = ($s < $t)
xx_xx_0_1_10_0x_10000_xxx_0_x_101_00_xxxxx  // 0x1b   bltz  if ($s < 0) pc += i << 2
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x1c   bgez  if ($s >= 0) pc += i << 2
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x1d   j     pc = pc4 | i26 << 2
10_11_1_x_xx_xx_xxxxx_xxx_0_x_xxx_01_xxxxx  // 0x1e   jal   $31 = pc; pc = pc4 | i26 << 2
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x1f   beq   if ($s == $t) pc += i << 2
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x20   bne   if ($s != $t) pc += i << 2
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x21   blez  if ($s <= 0) pc += i << 2
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x22   bgtz  if ($s > 0) pc += i << 2
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x23   addi  $t = $s + SE(i)
00_00_1_1_01_0x_00101_xxx_0_0_000_00_xxxxx  // 0x24   addiu $t = $s + SE(i)
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x25   slti  $t = ($s < SE(i))
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x26   sltiu $t = ($s < ZE(i))
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x27   andi  $t = $s & ZE(i)
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x28   ori   $t = $s | ZE(i)
00_00_1_0_01_0x_00011_xxx_0_0_000_00_xxxxx  // 0x29   xori  $t = $s ^ ZE(i)
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x2a   lui   $t = i << 16
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x2b   lb    $t = SE (MEM [$s + i]:1)
00_00_1_1_01_0x_00100_010_0_1_000_00_xxxxx  // 0x2c   lh    $t = SE (MEM [$s + i]:2)
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x2d   lw    $t = MEM [$s + i]:4
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x2e   lbu   $t = ZE (MEM [$s + i]:1)
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x2f   lhu   $t = ZE (MEM [$s + i]:2)
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x30   sb    MEM [$s + i]:1 = LB ($t)
xx_xx_0_1_01_0x_00100_010_1_x_000_00_xxxxx  // 0x31   sh    MEM [$s + i]:2 = LH ($t)
xx_xx_x_x_xx_xx_xxxxx_xxx_x_x_xxx_xx_xxxxx  // 0x32   sw    MEM [$s + i]:4 = $t
```

다음으로 PLA_OR.txt 파일이다. PLA_OR 는 입력 신호들의 OR 연산 결과를 계산한다. 입력 신호를 OR 연산해서 여러 조건 중 하나 이상이 참일 경우 출력 신호를 생성해 전체 회로의 동작을 제어한다. 구현한 10 개의 Instruction 을 표로 나타내면 다음과 같다.

Instruction /Port name	ADDU	OR	ADDIU	XORI	SLL	SRAV	SH	LH	BLTZ	JAL
RegDst	01	01	00	00	01	01	xx	00	xx	10
RegDatSel	00	00	00	00	00	00	xx	00	xx	11
RegWrite	1	1	1	1	1	1	0	1	0	1

SEUmode	X	X	1	0	X	X	1	1	1	X
ALUsrcB	00	00	01	01	00	00	01	01	10	xx
ALUctr	0x	0x	0x	0x	00	01	0x	0x	0x	xx
ALUop	00101	00001	00101	00011	01101	01111	00100	00100	10000	xxxxx
DataWidth	xxx	xxx	xxx	xxx	xxx	xxx	010	010	xxx	xxx
MemWrite	0	0	0	0	0	0	1	0	0	0
MemtoReg	0	0	0	0	0	0	X	1	X	X
Branch	000	000	000	000	000	000	000	000	101	xxx
Jump	00	00	00	00	00	00	00	00	00	01

다음으로 각 명령어 별 기능과 동작에 대해 설명하겠다.

ADDU

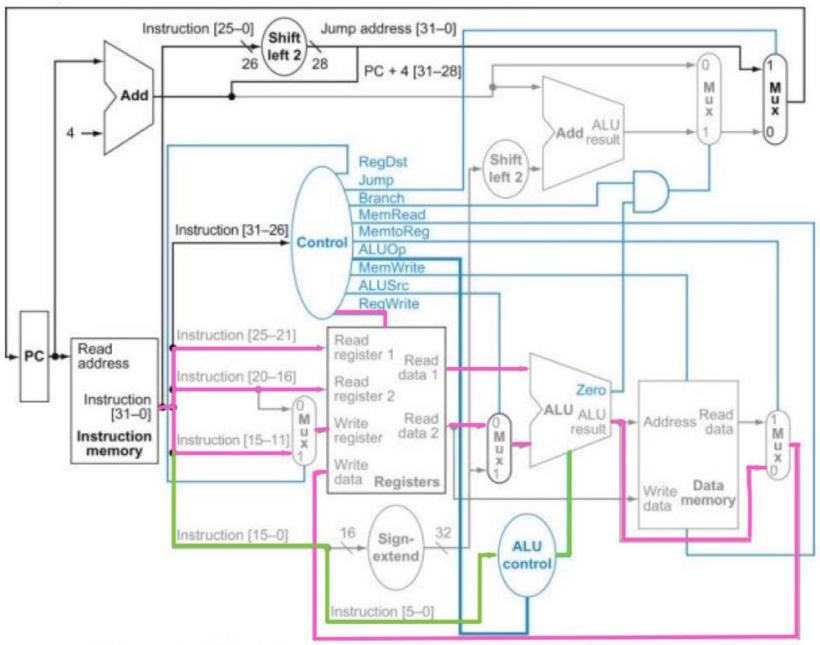


Figure 1 - The single cycle CPU datapath and control path

먼저 ADDU 명령어로, 부호 없는 두 개의 레지스터 값을 더하는 연산을 수행한다. addu $rd, $rs, $rt 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rd 는 결과를 저장할

대상, rt 와 rs 는 더할 값이 들어 있는 레지스터이다. 동작방식은 다음과 같다. 명령어 fetch 단계에서 명령어 메모리에서 해당 명령어를 가져온다. rtype 의 명령어이므로, ALU control 에서 function 의 값을 해석한다. 여기서 초록색 선은 function 을 의미한다. 명령어 decode 단계에서 rs 와 rt 는 각각 read register1, read register 2 로 입력되며 rd 는 mux 에 의해 선택된다. ALU 의 입력으로 rs 와 rt 가 들어가며, 결과를 다시 Register 에 write 해주었다. 따라서 reg write 도 1 이 된다.

OR

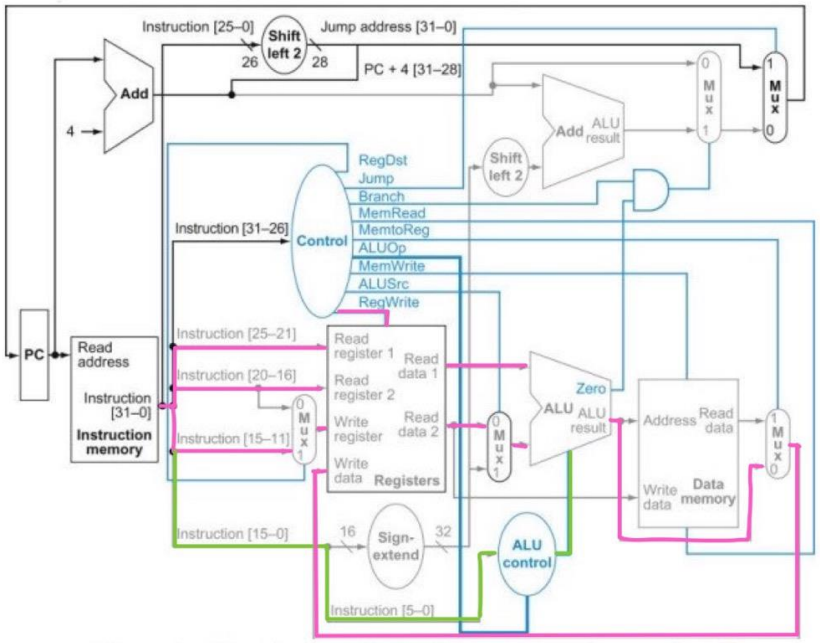


Figure 1 - The single cycle CPU datapath and control path

OR 명령어로, 두 개의 레지스터 값을 bit 별로 or 연산을 수행한다. or $rd, $rs, $rt 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rd 는 결과를 저장할 대상, rt 와 rs 는 더할 값이 들어 있는 레지스터이다. 동작방식은 다음과 같다. 명령어 fetch 단계에서 명령어 메모리에서 해당 명령어를 가져온다. rtype 의 명령어이므로, ALU control 에서 function 의 값을 해석한다. 여기서 초록색 선은 function 을 의미한다. 명령어 decode 단계에서는 rs 와 rt 는 각각 read register1, read register 2 로 입력되며 rd 는 mux 에 의해 선택된다. ALU 의 입력으로 rs 와 rt 가 들어가며, 결과를 다시 Register 에 write 해주었다. 따라서 reg write 도 1 이 된다. OR 명령어는 ADDU 명령어와 실행 흐름이 동일하다.

ADDIU

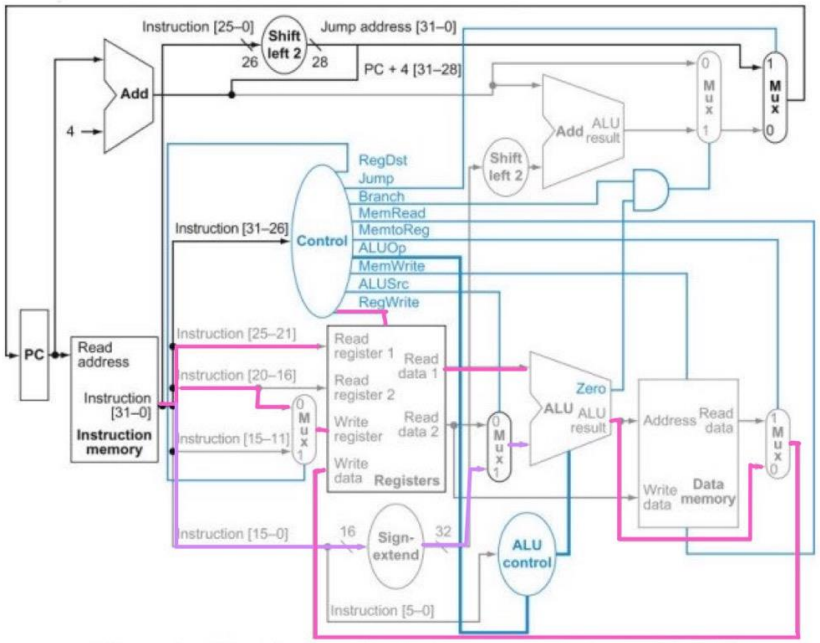


Figure 1 - The single cycle CPU datapath and control path

ADDIU 명령어로, 레지스터와 16bit 의 상수 값을 더한다. addiu $rt, $rs, imm 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rt 는 결과를 저장할 대상 레지스터, rs 는 더해질 값이 들어 있는 레지스터이며 imm 은 immediate value 를 말한다. 동작방식은 다음과 같다. 명령어 fetch 단계에서 명령어 메모리에서 해당 명령어를 가져온다. immediate value 를 sign-extend 하여 32bit 로 만들어야 하기 때문에 sign-extend 를 1 로 준다. 이때 보라색 선이 immediate value 에 해당한다. 명령어 decode 단계에서는 rs 와 rt 의 값을 읽어온 후, ALU 의 입력으로 rs 와 imm 이 들어가며, 결과를 다시 Register 에 write 해주었다. 따라서 reg write 도 1 이 된다.

XORI

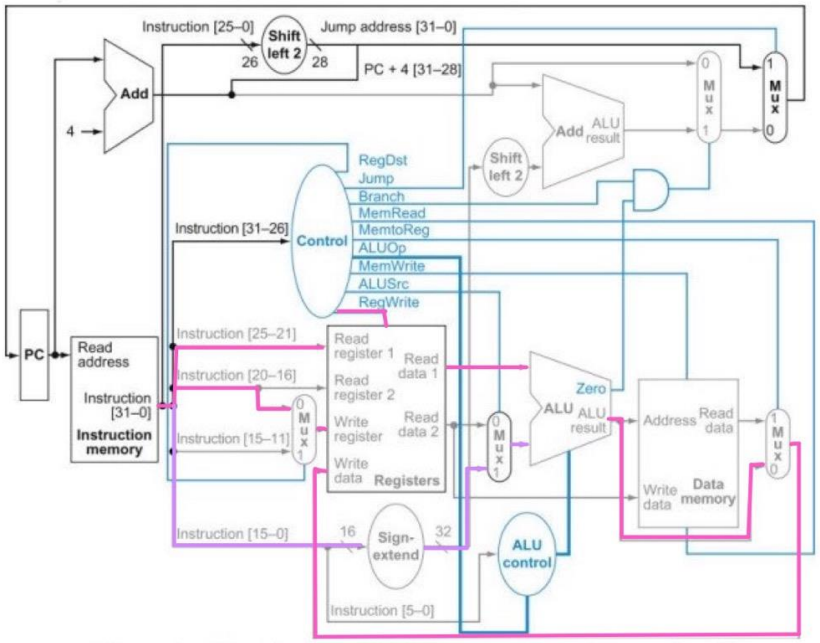


Figure 1 - The single cycle CPU datapath and control path

XORI 명령어로, 레지스터와 16bit 의 상수 값을 xor 한다. xor $rt, $rs, imm 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rt 는 결과를 저장할 대상, rs 는 더해질 값이 들어 있는 레지스터이며 imm 은 immediate value 를 말한다. 동작방식은 다음과 같다. 명령어 fetch 단계에서 명령어 메모리에서 해당 명령어를 가져온다. immediate value 를 sign-extend 하여 32bit 로 만들어야 하기 때문에 sign-extend 를 1 로 준다. 이때 보라색 선이 immediate value 에 해당한다. 명령어 decode 단계에서는 rs 와 rt 의 값을 읽어온 후, ALU 의 입력으로 rs 와 imm 이 들어가며, 결과를 다시 Register 에 write 해주었다. 따라서 reg write 도 1 이 된다. ADDIU 명령어와 실행 흐름이 동일하다.

SLL

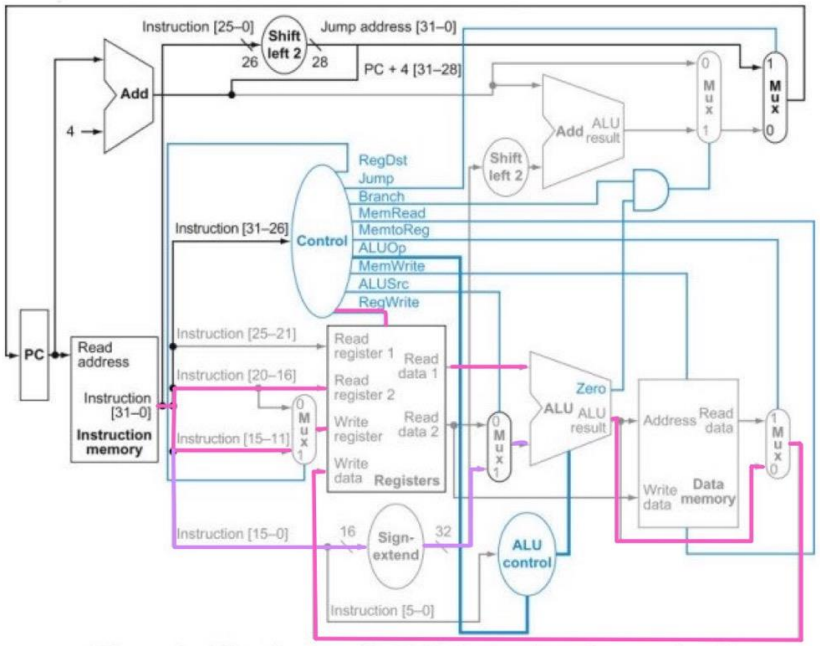


Figure 1 - The single cycle CPU datapath and control path

SLL 명령어로, 레지스터 값의 비트를 왼쪽으로 amount 만큼 shift 한다. sll $rd, $rt, sa 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rd 는 결과를 저장할 대상, rt 는 시프트할 값이 들어 있는 레지스터이며 sa 는 shift amount 를 의미한다. 동작방식은 다음과 같다. 명령어 fetch 단계에서 명령어 메모리에서 해당 명령어를 가져온다. sa 는 5bit 이기 때문에 sign-extend 를 거쳐 32bit 로 확장해준다. 여기서 보라색 선은 shift amount 를 의미한다. 명령어 decode 단계에서는 rt 와 rd 의 값을 읽어온 후, ALU 의 입력으로 rt 와 sa 가 들어가며, 결과를 다시 Register 에 write 해주었다. 따라서 reg write 도 1 이 된다.

SRAV

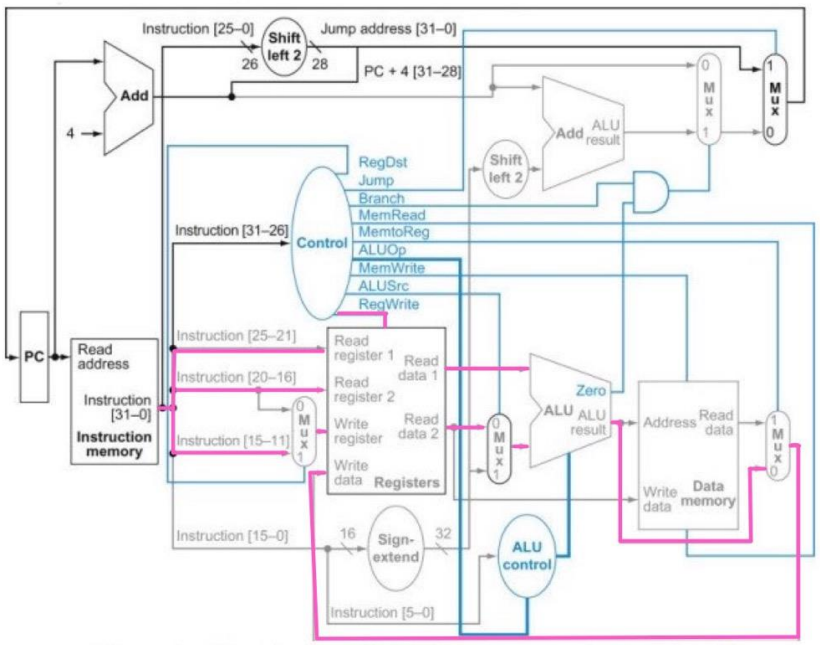


Figure 1 - The single cycle CPU datapath and control path

SRAV 명령어로, 레지스터 값의 비트를 오른쪽으로 레지스터 rs 만큼 shift 한다. srav $rd, $rt, $rs 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rd 는 결과를 저장할 대상, rt 는 더해질 값이 들어 있는 레지스터이며 rs 은 shift 할 값을 말한다. 동작방식은 다음과 같다. 명령어 fetch 단계에서 명령어 메모리에서 해당 명령어를 가져온다. 명령어 decode 단계에서는 rs 와 rt, rd 의 값을 읽어온 후, ALU 의 입력으로 rs 와 rt 가 들어가며, 결과를 다시 Register 에 write 해주었다. 따라서 reg write 도 1 이 된다.

SH

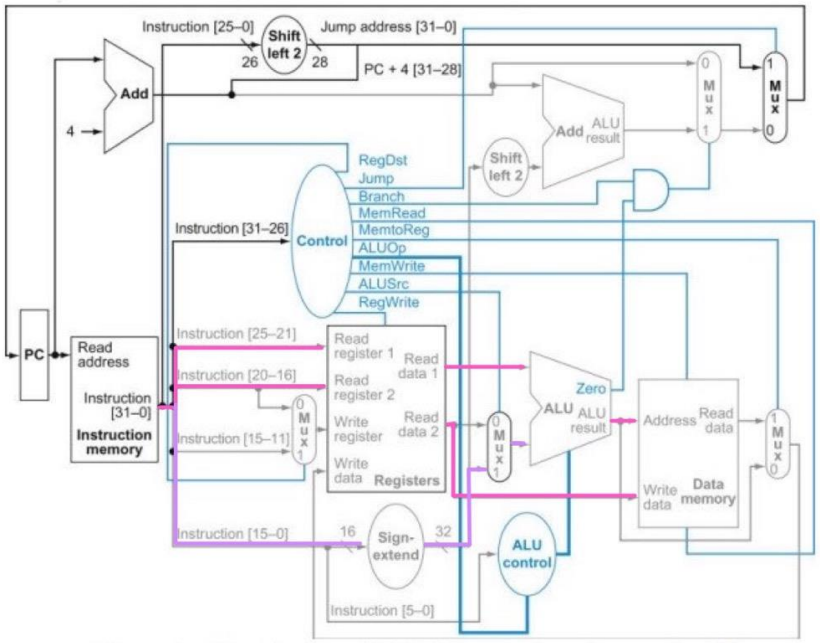


Figure 1 - The single cycle CPU datapath and control path

SH 명령어로, 하위 16bit 를 메모리에 저장한다. sh $rt, imm 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rt 는 결과를 저장할 대상, imm 은 immediate value 를 말한다. 동작방식은 다음과 같다. 명령어 fetch 단계에서 명령어 메모리에서 해당 명령어를 가져온다. immediate value 를 sign-extend 하여 32bit 로 만들어야 하기 때문에 sign-extend 를 1 로 준다. 이때 보라색 선이 immediate value 에 해당한다. 명령어 decode 단계에서는 rs 와 rt 의 값을 읽어온 후, ALU 의 입력으로 rs 와 imm 이 들어간다. 메모리에 값을 저장해야 하기 때문에 write data 가 1 이다.

LH

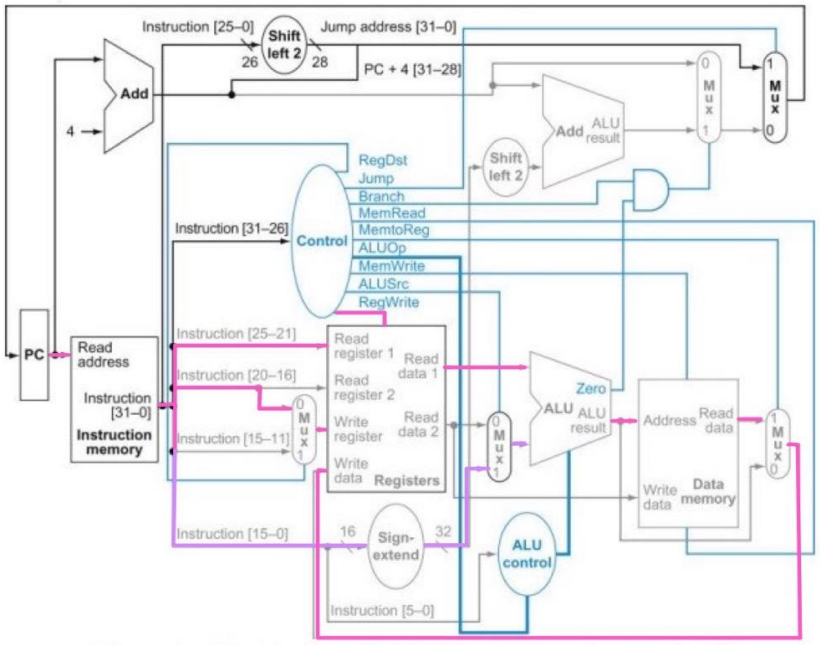


Figure 1 - The single cycle CPU datapath and control path

LH 명령어로, 하위 16bit 를 load 한다. lh $rt, imm 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rt 는 결과를 저장할 대상, imm 은 immediate value 를 말한다.
동작방식은 다음과 같다. 명령어 fetch 단계에서 명령어 메모리에서 해당 명령어를 가져온다. immediate value 를 sign-extend 하여 32bit 로 만들어야 하기 때문에 sign-extend 를 1 로 준다. 이때 보라색 선이 immediate value 에 해당한다. 명령어 decode 단계에서는 rs 와 rt 의 값을 읽어온 후, ALU 의 입력으로 rs 와 imm 이 들어가며, 결과를 다시 Register 에 write 해주었다. 따라서 reg write 도 1 이 된다.

BLTZ

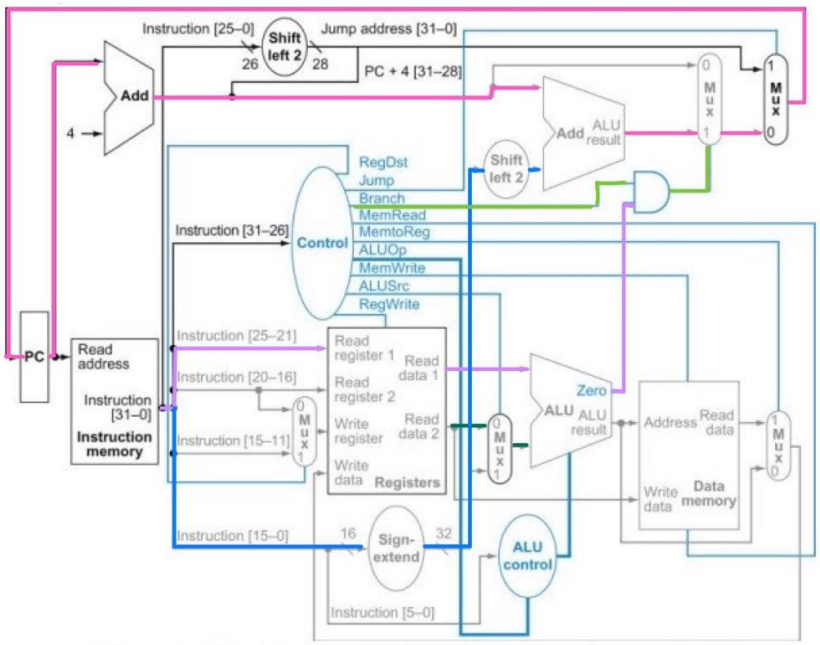


Figure 1 - The single cycle CPU datapath and control path

BLTZ 명령어로, 레지스터의 값이 0 보다 작으면 label 로 분기한다. bltz $rs, label 의 syntax 를 가지고, 각각 레지스터의 번호를 나타내며 rs 는 분기 조건으로 사용될 레지스터이며, label 은 분기할 목적지 주소를 말한다. 동작방식은 다음과 같다. 명령어 fetch 단계에서 명령어 메모리에서 해당 명령어를 가져온다 label 을 sign-extend 하여 32bit 로 만들어야 하기 때문에 sign-extend 를 1 로 준다. 이때 파란색 선이 label 에 해당한다. 명령어 decode 단계에서는 rs 의 값을 읽어온 후, ALU 의 입력으로 rs 와 0 이 들어간다. rs 가 0 보다 작으면 alu 는 1 을 반환하기 위해 보라색 선처럼 zero 로 나온다. 핑크색 선은 pc + 4 + label 을 표현하였다.

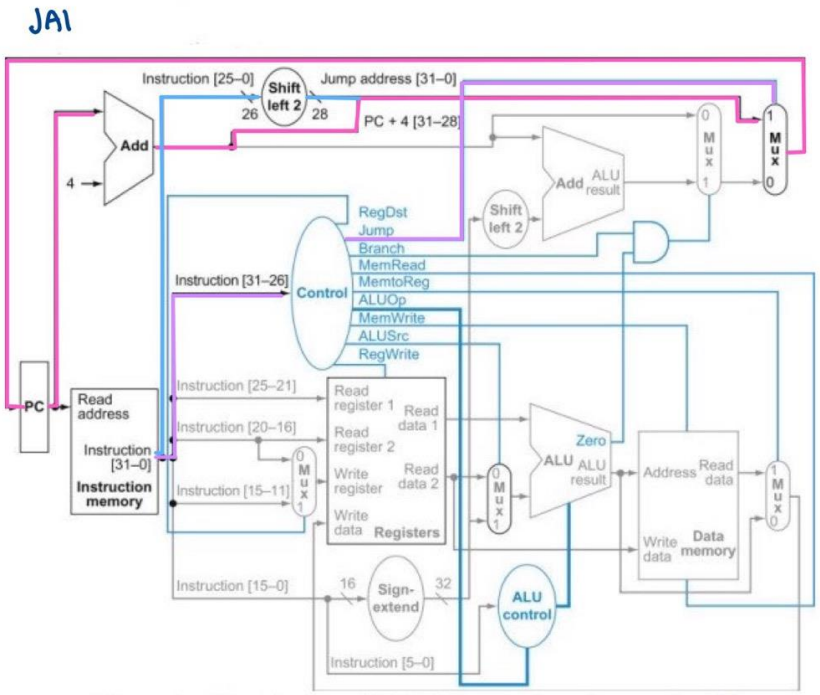


Figure 1 - The single cycle CPU datapath and control path

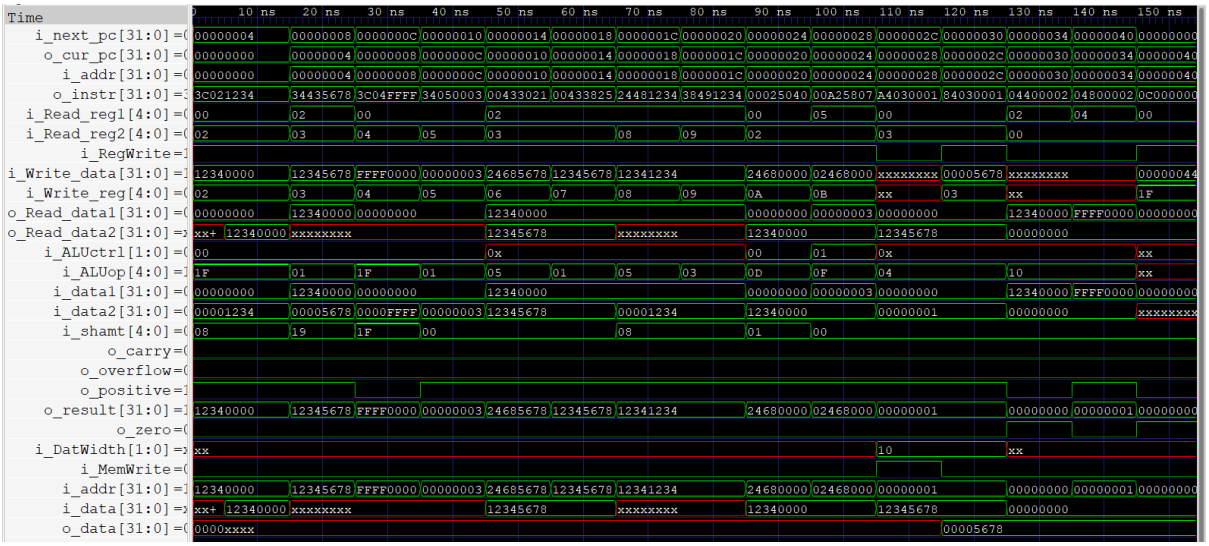
JAL 명령어로, 현재 pc 값을 레지스터에 저장하고 지정된 주소로 분기한다. jar label 의 syntax 를 가지고, label 은 분기할 목적지 주소를 말한다. 동작방식은 다음과 같다. 명령어 fetch 단계에서 명령어 메모리에서 해당 명령어를 가져온다. 핑크색 선과 파란색 선은 pc + 4 + shift left 2 label 을 표현하였다. 보라색 선은 jump 를 하는 것을 의미한다.
회로도에는 표현할 수 없지만, write register 에 $31 을 연결해 주어야 하고, write data 에 pc+4 도 연결해주어야 한다.

다음으로 구현한 내용에 대해서 설명해보겠다.

```
001111_00000_00010_0001001000110100 //lui r2 0x1234 -> r2,0x12340000
001101_00010_00011_0101011001111000 //ori r2 r3 0x5678 -> r3,0x12345678
001111_00000_00100_1111111111111111 //lui r4 0xffff -> r4,0xffff0000
001101_00000_00101_0000000000000011 //ori r0 r5 0x0003 -> r5,0x00000003

000000_00010_00011_00110_00000_100001 //addu r2 r3 r6 -> r6,24686578
000000_00010_00011_00111_00000_100101 //or r2 r3 r7 -> r7,24685678
001001_00010_01000_0001001000110100 //addiu r2 r8 0x1234 -> r8, 0x12341234
001110_00010_01001_0001001000110100 //xori r2 r9 0x1234 -> r9, 0x12341234
000000_00000_00010_01010_00001_000000 //sll r2 r10 shift 1 -> r10, 0x24680000
000000_00101_00010_01011_00000_000111 //srav r5 r2 r11 -> r11, 0x24680000
101001_00000_00011_0000000000000001 //sh r3 1 해당 주소 값 넣음
100001_00000_00011_0000000000000001 //lh r3 1 해당 주소 값을 가져옴
000001_00010_00000_0000_0000_0000_0010 //bltz r2 branch -> pc + 4
000001_00100_00000_0000_0000_0000_0010 //bltz r4 branch -> pc+4 + label
001111_00000_00010_0001001000110100 //lui r2 0x1234 -> r2,0x12340000
001101_00010_00011_0101011001111000 //ori r2 r3 0x5678 -> r3,0x12345678
000011_0000_0000_0000_0000_0000_000000 //jal
```

먼저 위의 사진은 M_TEXT_SEG.txt 파일로 작성한 명령어가 잘 동작하는지 확인하기 위한 txt 파일이다. 사용한 레지스터와 명령어, 그리고 최종 값이 어떻게 나오는지 주석으로 작성해두었다.



실행 결과로 하나씩 살펴보겠다.

먼저 0x00000000 부터 0x0000000c 까지는 lui 명령어와 ori 명령어를 사용하여 r2, r3, r4, r5 레지스터에 각각 0x12340000, 0x12345678, 0xffff0000, 0x00000003 을 넣어주었다.

다음으로 0x00000010 에서 addu 명령어를 수행하였는데, r2 와 r3 을 더해 r6 에 저장하였으므로, r6 에 0x24686578 이 저장된 것을 확인할 수 있다.

다음으로 0x00000014 에서 or 명령어를 수행하였는데, r2 와 r3 을 or 연산을 하여 r7 에 저장하였으므로 r7 에 0x12345678 이 저장된 것을 확인할 수 있다.

다음으로 0x00000018 에서 addiu 명령어를 수행하였는데, r2 와 0x1234 를 더해 r8 에 저장하였으므로, r8 에 0x12341234 가 저장된 것을 확인할 수 있다.

다음으로 0x0000001c 에서 xori 명령어를 수행하였는데, r2 와 0x1234 를 xor 연산을 하여 r9 에 저장하였으므로, r9 에 0x12341234 가 저장된 것을 확인할 수 있다.

다음으로 0x00000020 에서 sll 명령어를 수행하였는데, r2 를 shift 1 해서 r10 에 저장하였으므로, r10 에 0x24680000 이 저장된 것을 확인할 수 있다.

```
i_data2[31:0]=00010010001101000000000000000000
 i_shamt[4:0]=01
     o_carry=0
  o_overflow=0
  o_positive=1
o_result[31:0]=00100100011010000000000000000000
```

위의 연산을 binary 로 확인한 것이다.

다음으로 0x00000024 에서 srav 명령어를 수행하였는데 r2 를 r5 만큼 shift 해서 r11 에 저장하였으므로, r11 에 0x02468000 이 저장된 것을 확인할 수 있다.

```
 i_data2[31:0]=00010010001101000000000000000000
  i_shamt[4:0]=00
       o_carry=0
    o_overflow=0
    o_positive=1
o_result[31:0]=00000010010001101000000000000000
```

위의 연산을 binary 로 확인한 것이다.

다음으로 0x00000028 에서 sh 명령어를 수행하였는데 r3 에 저장된 값을 1 에 해당하는 주소의 메모리에 넣으므로 하위 16bit 에 5678 이 저장된 것을 알 수 있다.

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

다음으로 0x0000002c 에서 lh 명령어를 수행하였는데 1 에 해당하는 메모리에서 값을 가져와 r3 에 저장한다. 따라서 0x00005678 이 저장된 것을 확인할 수 있다.

다음으로 0x00000030 에서 bltz 명령어를 수행하는데 r2 와 branch 값을 비교해서 r2 가 0 보다 크므로 pc+4 만 진행하므로 next_pc 가 0x00000034 가 된다..

다음으로 0x00000034 에서 bltz 명령어를 수행하는데 r4 와 branch 값을 비교해서 r4 가 0 보다 작으므로 pc+4+label 을 진행해하므로 next_pc 가 0x00000040 이 된다.

```
00000000 00000000 00000000 00000000 : 00000000
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
00010010 00110100 00000000 00000000 : 12340000
00010010 00110100 01010110 01111000 : 12345678
11111111 11111111 00000000 00000000 : ffff0000
00000000 00000000 00000000 00000011 : 00000003
00100100 01101000 01010110 01111000 : 24685678
00010010 00110100 01010110 01111000 : 12345678
00010010 00110100 00010010 00110100 : 12341234
00010010 00110100 00010010 00110100 : 12341234
00100100 01101000 00000000 00000000 : 24680000
00000010 01000110 10000000 00000000 : 02468000
xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx : xxxxxxxx
```

0 부터 11 까지의 레지스터이다. 명령어를 수행한 후의 값이 저장된 것을 확인할 수 있다.

3. 고찰

실습 시간에는 직접 cmd 창에 명령어를 써 gtkwave 를 실행시켰는데, run 실행파일을 통해 바로 waveform 을 볼 수 있다는 것이 신기했다. 또한 M_TEXT_SEG.txt 파일에 값을 넣어준 후 gtkwave 를 통해 10 개의 명령어에 대한 기능과 동작을 설명하는 부분에서 결과를 시각화 하여 명령어가 잘 구현됐는지 확인할 수 있는 점이 편했고 도움이 많이 되었다. 컴퓨터 구조 강의 시간에 이론으로만 배웠던 MIPS 에서의 single cycle CPU 를 이번 프로젝트를 통해 직접 구현하고 직접 흐름을 파악하고 신호에 값을 넣어주는 과정을 통해 single cycle path 의 동작 방식에 대해 잘 이해가 된 것 같다. 이번 프로젝트를 통해 다양한 명령어를 구현해 볼 수 있었지만, 아직은 명령어를 구현하는 부분이 어색하고 헷갈린 점도 많은 것 같다. 하지만 동작 방식을 설명하기 위해 datapath 그림 1 을 활용하였는데, 구현한 부분과 주어진 회로도가 다른 부분이 있어 동작을 그림으로 표현하기가 까다로웠던 것 같다. sll 과 srav 명령어가 잘 구현됐는지 gtkwave 를 통해 확인하면서, 처음에는 값이 왜 이렇게 나오는지 몰랐고, 이해가 가지 않아 이해하는데 있어 시간이 오래 걸렸지만, binary 로 바꾸어 볼 생각을 하지 못해 시간을 많이 뺏긴 점이 아쉬웠다. 다음 프로젝트인 multi cycle 을 구현하는데 있어 이번 프로젝트가 도움이 많이 될 것 같다.

4. Reference

컴퓨터구조실험 강의자료
컴퓨터구조 강의자료