Project #1

MIPS Single Cycle CPU Implementation

과목명	컴퓨터구조
담당교수	이성원 교수님
학과	컴퓨터정보공학부
학년	3학년
학번	2019202009
이름	서여지
제출일	2021.04.12(월)

KWANGWOON UNIVERSITY

1. 문제의 해석 및 해결 방향

주어진 single cycle CPU에 대해 제시된 12종류의 명령어에 따른 data path를 구성하는 control signal을 적절하게 구하는 프로젝트이다. 주어진 CPU는 32개의 32bit register를 갖는 register file과 ALU, MULT 등의 module을 갖고있으며, 이것을 제어하는 signal을 MyControl 에 작성한다.

A. 12개의 명령어에 대한 기능과 동작 상세 설명

제시된 명령어는 NOR, SLT, SLTI, SRAV, ADDI, BEQ, BGTZ, MULT, MFLO, XORI, JAL, JR로 총 12개이다. 각 명령어의 기능에 대해 설명하고, Sample Assembly Program에서 수행하는 동작에 대한 설명은 B. 실험 내용에 대한 설명에서 다룬다.

(1) NOR

syntax	f $d, $s, $t	operation	$d = ~($s|$t)

nor 명령어는 $s와 $t로 전달된 register에 저장된 정보를 nor연산하여 그 결과를 $d로 전달된 register에 저장하는 동작을 한다. ALU의 Bitwise NOR연산을 이용하여 계산한 뒤, RF에 저장한다.

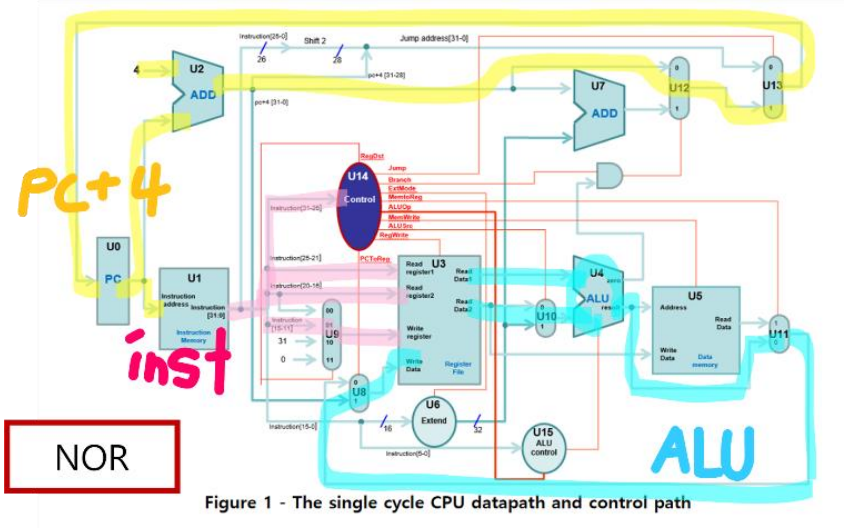


Figure 1 - The single cycle CPU datapath and control path

(2) SLT

syntax	f $d, $s, $t	operation	$d = ($s<$t)

slt 명령어는 $s와 $t로 전달된 register에 저장된 값의 크기를 비교하여 $s가 더 작은 경우 1을 $d에 저장하고, 아닌 경우 0을 저장한다. ALU의 Set Less Than을 이용하여 계산한 뒤, RF에 저장한다.

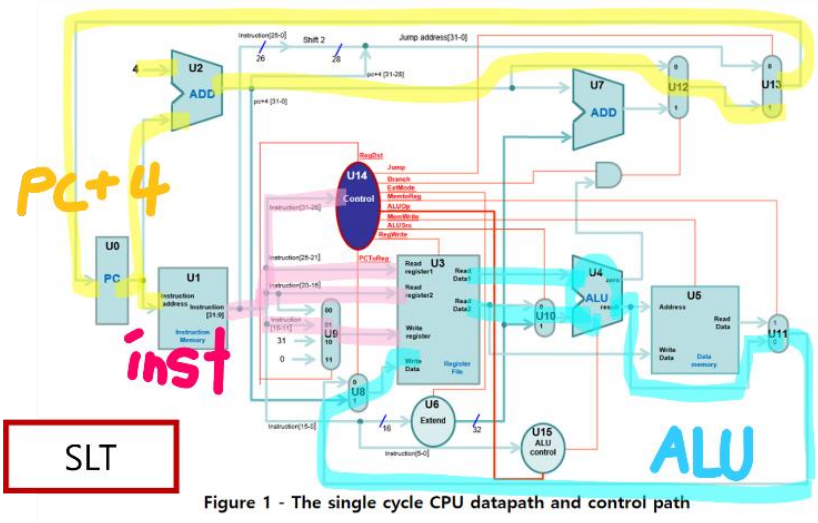


Figure 1 - The single cycle CPU datapath and control path

(3) SLTI

syntax	o $d, $s, i	operation	$d = ($s<SE(i))

slti 명령어는 slt명령어와 유사하게 인자로 전달된 두 값의 크기를 비교한다. 이때 slt명령어와의 차이점은 비교하는 값으로 immediate를 이용하는 것이다. 값을 구하는 방법은 slt와 동일하게 ALU의 Set Less Than을 이용하였다.

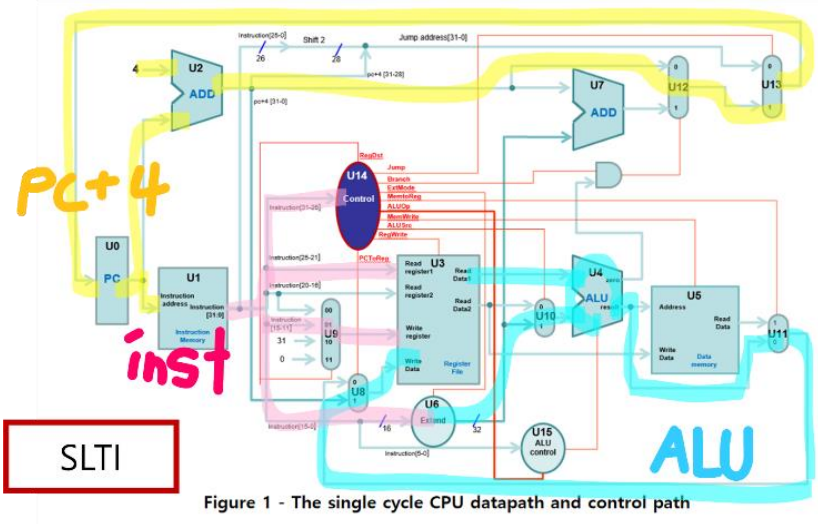


Figure 1 - The single cycle CPU datapath and control path

(4) SRAV

syntax	f $d, $t, $s	operation	$d = $t>>>$s

srav 명령어는 $t로 전달된 register에 저장된 값을 $s로 전달된 register의 값만큼 arithmetic right shift한다. 그리고 그 결과를 $d에 저장한다. ALU의 Shift Right Arithmetic을 이용하여 계산한 뒤, RF에 저장한다.

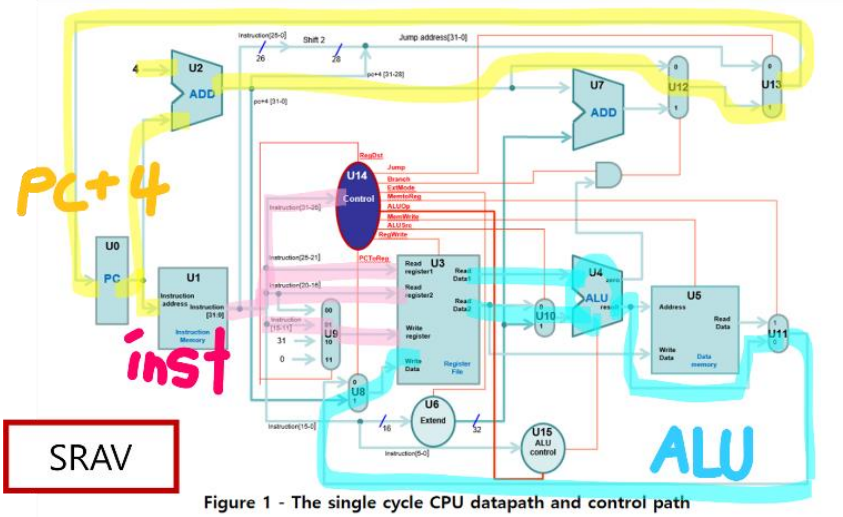


Figure 1 - The single cycle CPU datapath and control path

(5) ADDI

syntax	o $d, $s, i	operation	$d = $s + SE(i)

addi 명령어는 $s로 전달된 register에 저장된 값과 immediate를 더하고, 그 결과를 $d에 저장한다. ALU의 덧셈을 이용하여 계산한 뒤, RF에 저장한다.

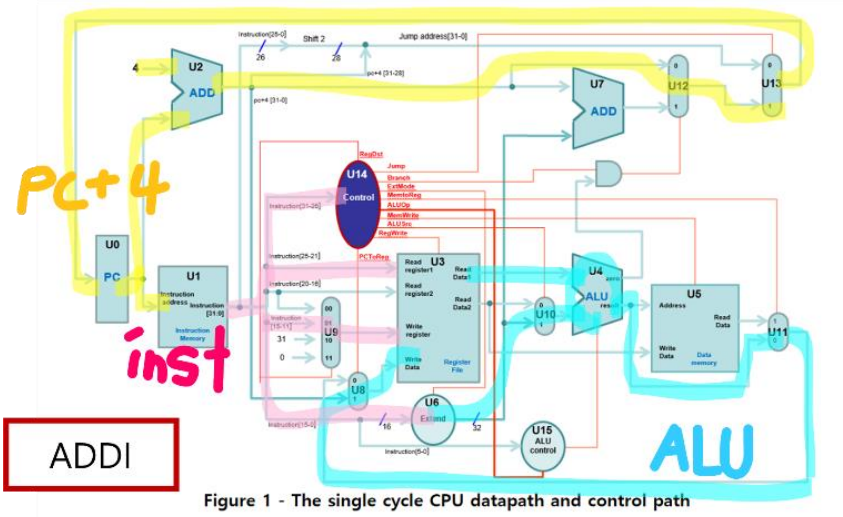


Figure 1 - The single cycle CPU datapath and control path

(6) BEQ

syntax	o $s, $t, label	operation	if($s==$t) pc + = i <<2

beq 명령어는 $s와 $t로 전달된 register의 값이 같은 경우 pc를 label로 이동시킨다. ALU module은 $s와 $t의 값을 비교하는 것에 사용된다. 값의 비교는 ALU의 뺄셈계산 이후 그 결과가 0인 경우에 두 값이 같음을 이용하였다. 제안서에 제시된 그림에서 U7로 나타난 adder에서 jump할 pc값을 계산한다.

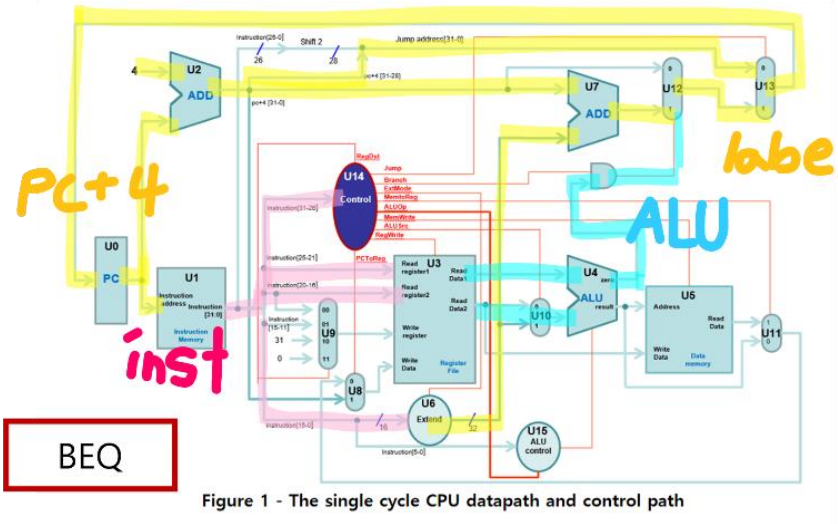


Figure 1 - The single cycle CPU datapath and control path

(7) BGTZ

syntax	o $s, label	operation	if($s>0) pc+ =i<<2

bgtz 명령어는 $s로 전달된 register의 값이 0보다 큰 경우 pc를 label로 이동시킨다. ALU module은 Set Less Than을 이용하여 $s의 값과 0을 비교하고, 이 결과가 0이 아닌 경우 jump한다.

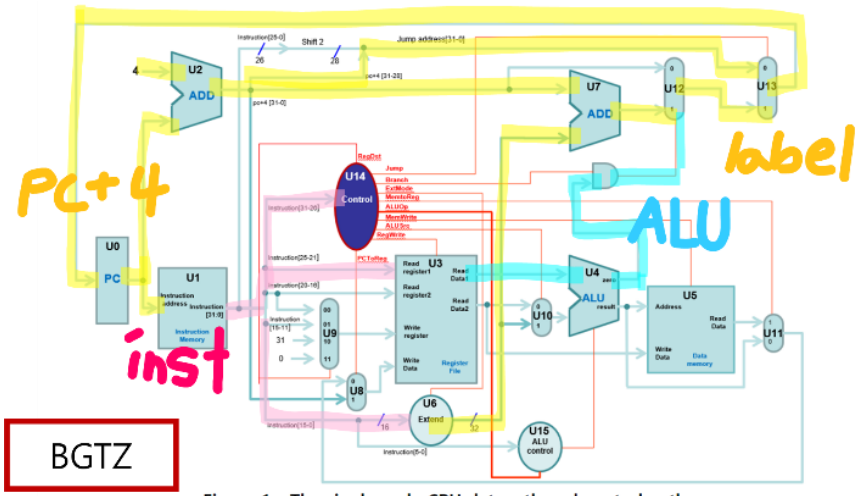


Figure 1 - The single cycle CPU datapath and control path

(8) MULT

syntax	f $s, $t	operation	hi:lo = $s * $t

mult 명령어는 $s와 $t로 전달된 register의 값을 곱하여 그 결과를 hi와 lo에 저장한다. 32bit * 32bit 계산으로 64bit의 결과를 얻는다. 이때 hi에는 상위 32bit, lo에는 하위 32bit를 저장한다.

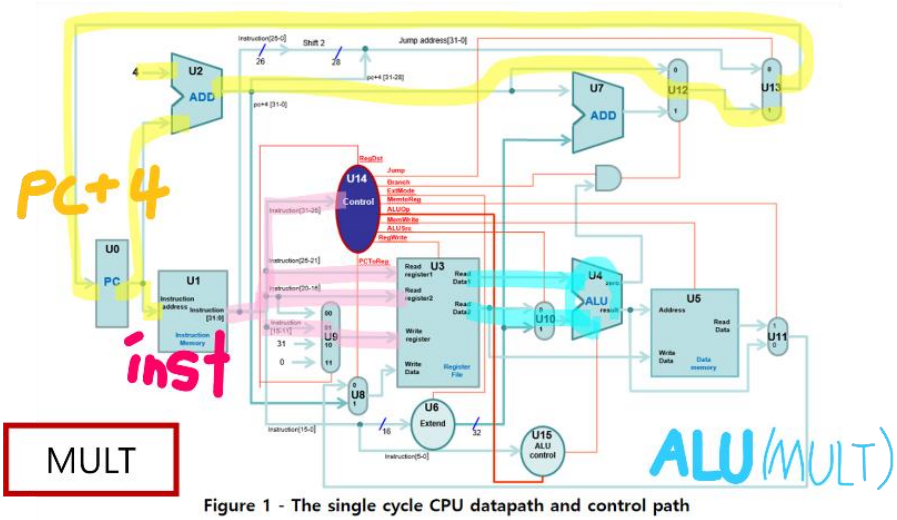


Figure 1 - The single cycle CPU datapath and control path

(9) MFLO

syntax	f $d	operation	$d = lo

mflo 명령어는 $d로 전달된 registser에 lo의 내용을 저장하는 동작을 한다. lo는 mult계산을 통해 얻은 값의 하위 32bit가 저장되어있다.

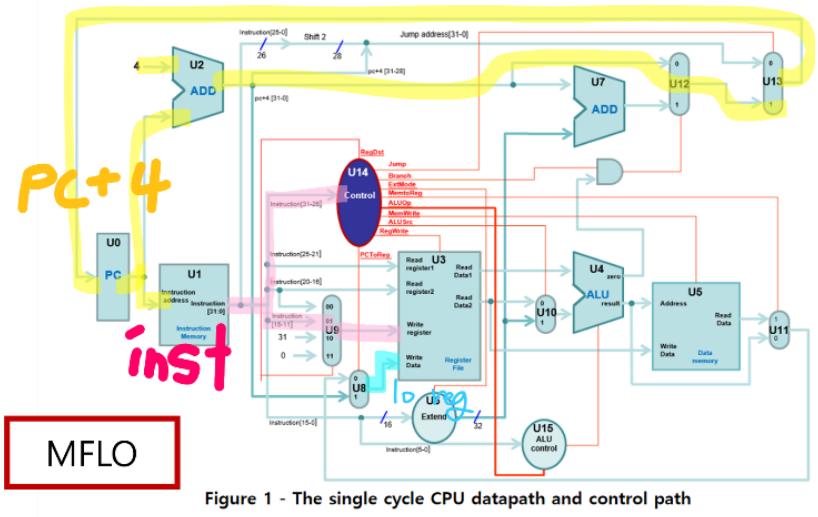


Figure 1 - The single cycle CPU datapath and control path

(10)XORI

syntax	o $t, $s, i	operation	$t = $s^ZE(i)

xori 명령어는 $s로 전달된 register에 저장된 값과 immediate값을 xor계산하여 $t에 저장하는 동작을 한다. ALU의 bitwise xor을 이하여 결과를 구하고, RF에 저장한다.

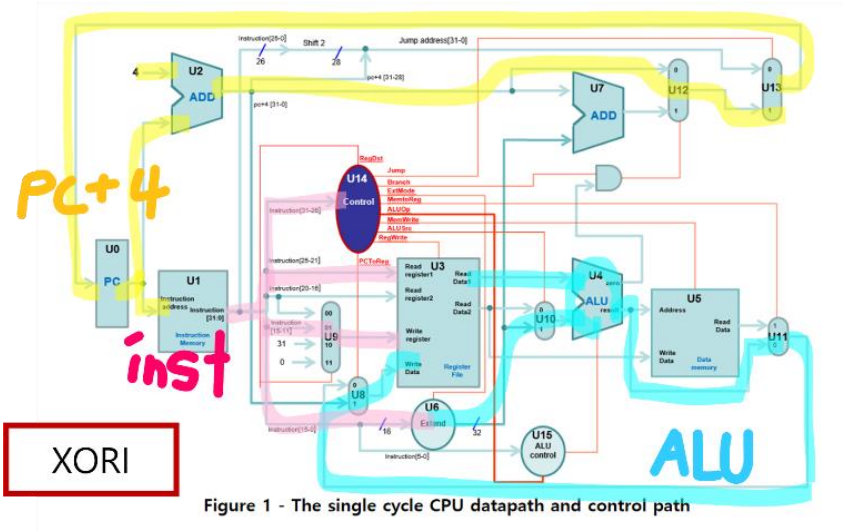


Figure 1 - The single cycle CPU datapath and control path

(11)JAL

syntax	o label	operation	$31=pc; pc + =i<<2

jal 명령어는 현재 pc값을 31번 register에 저장하고, pc를 label로 이동시킨다.

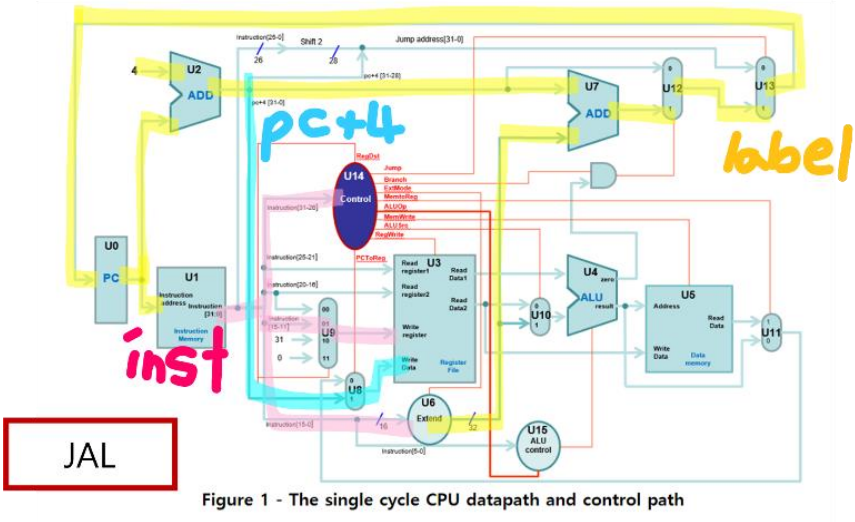


Figure 1 - The single cycle CPU datapath and control path

(12)JR

syntax	f labelR	operation	pc = $s

jr 명령어는 인자로 전달된 register의 값을 pc에 저장하여 해당 label의 위치로 이동한다.

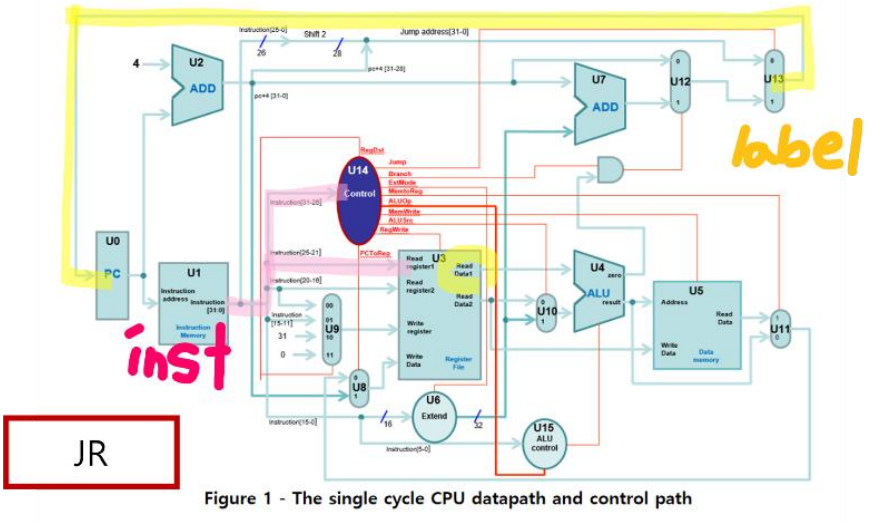


Figure 1 - The single cycle CPU datapath and control path

B. 실험 내용에 대한 설명

주어진 자료를 이용해 작성한 코드를 확인하였다. Sample Assembly Program은 다음과 같이 동작한다.

0000: ori $r10, $r0, 0x110
$r0의 값과 immediate를 or 연산하여 그 결과를 $r10에 저장한다. 이때 $r0의 값은 0이므로 $r10에는 0x110이 저장된다.
0004: add $r9, $r10, $r10
$r10의 값을 자기자신에 더하여 $r9에 저장한다. $r10에 0x110이 저장되어 있었으므로 $r9 = 0x220이다.
0008: sub $r8, $r9, $r10
$r9의 값에서 $r10의 값을 뺀 후 결과를 $r8에 저장한다. $r10 = 0x110, $r9 = 0x220이므로 $r8 = 0x110이다.
000C: lw $r1, -256($r8)
$r1에 메모리에서 읽은 값을 저장한다. 읽으려는 메모리의 주소는 $r8에서 immediate값을 더한 결과인 0x10이다. 000C: 10001101 00000001 11111111 00000000 op: 100011/ $rs: 01 000/$rt: 00001/ imm16 11111111 00000000 <- (-256) INSTR_ROM.txt의 해당 줄을 살펴보면 immediate로 전달된 값이 제안서의 표기인 -128와 다른 -256임을 확인할 수 있다. 1 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx // 0x00 2 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx // 0x04 3 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx // 0x08 4 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx // 0x0c 5 6 00010010 00110100 01010110 01111000 // 0x10 7 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx DATA_RAM.txt의 0x10에 저장된 내용이 저장되어 $r1 = 0x12345678이다.
0010: sw $r1, 4($r0)
메모리에 $r1의 값을 저장한다. 이때 $r0의 값은 0이므로 0x4의 위치에 $r1의 값인 0x12345678이 저장된다.

0014: lw $r2, 4($r0)
$r2에 메모리에서 읽은 값을 저장한다. 직전의 sw명령어에서 사용한 것과 같은 주소를 이용하므로 $r1 = $r2 = 0x12345678이 된다.
0018: j 0x020
pc를 0x20으로 이동한다.
0020: lhi $r3, 0x8765
$r3의 상위 16bit에 immediate로 전달된 값인 0x8765를 저장한다.
0024: llo $r3, 0x4321
$r3의 하위 16bit에 immediate로 전달된 값인 0x4321을 저장한다. lhi명령어와 llo의 수행결과 $r3에는 0x87654321이 저장된다.
0028: srav $r4, $r10, $r3
$r10의 값을 $r3의 값만큼 right arithmetic shift 하여 그 결과를 $r4에 저장한다. $r10의 값은 0x110이고, $r3의 값은 0x87654321이므로 $r4에는 0이 저장된다.
002C: nor $r5, $r4, $r0
$r4와 $r0의 nor 계산 결과를 $r5에 저장한다. 이때 $r0과 $r4의 값은 모두 0이므로 $r5에는 모든 비트가 1인 값이 저장된다.
0030: beq $r1, $r2, 0x38
$r1과 $r2에 저장된 값이 같은 경우 pc를 0x38로 이동한다. $r1 = $r2 = 0x12345678이므로 pc = 0x38이다.
0038: addi $r6, $r9, 1
$r9에 저장된 값에 immediate로 전달된 1을 더한 뒤 결과를 $r6에 저장한다. $r9 = 0x220이므로 $r6에는 0x221이 저장된다.
003C: jal 0x44
현재 pc값을 31번 register에 저장하고, pc에 0x44를 저장한다. 31번 register에는 0040이 저장된다.
0044: slt $r7, $r9, $r6
$r9와 $r6에 저장된 값의 크기를 비교하고 결과를 $r7에 저장한다. $r9 = 0x220, $r6 = 0x221이므로 $r9의 크기가 더 작다. 따라서 $r7에는 1이 저장된다.
0048: bgtz $r7, 0x50
$r7에 저장된 값이 0보다 큰 경우 pc에 0x50을 저장한다. $r7=1이므로 pc = 0x50이다.
0050: lui $r2, 0x78ff
immediate로 전달된 0x78ff를 16bit left shift하여 $r2에 저장한다. lhi 명령어의 동작과 유사하지만 shift결과를 이용하기 때문에 하위 16bit가 모두 0이된다는 차이가 있다. $r2 = 0x78ff0000이 된다.
0054: xori $r2, $r2, 0x4321
$r2의 값과 immediate로 전달된 0x4321를 xor계산하여 다시 $r2에 저장한다. $r2의 값은 0x78ff0000에서 0x78ff4321이 된다.
0058: mult $r2, $r8
$r2와 $r8의 값을 곱하여 hi, lo register에 저장한다. $r2 = 0x78ff4321, $r8 = 0x110이므로 곱셈의 결과는 64bit 0x808f375310이고, 32bit씩 나누어 hi, lo에 저장된다. hi = 0x80, lo = 0x8f375310
005C: mflo $r7
lo register의 값을 $r7에 저장한다. $r7 = 0x8f375310이다.
0060: slti $r10, $r7, 0x1
$r7의 값과 immediate값인 1을 비교하여 결과를 $r10에 저장한다.

$r7 = 0x8f375310이고, 이때 sign bit가 1이므로 음수이다. 따라서 $r10=1이다.
0064: beq $r1, $r2, 0x6c
$r1과 $r2의 값이 같으면 pc에 0x6c를 저장한다. $r1 = 0x12345678이고, $r2 = 0x78ff4321이다. 두 register에 저장된 값이 다르므로 pc는 pc+4인 0x68이 저장된다.
0x68: jr $r31
$r31의 값을 pc에 저장한다. jal명령어를 이용하여 $31에 0x40을 저장해두었으므로 pc = 40이 된다.
0x40: j 0x0
pc에 0을 저장한다. 프로그램이 처음부터 반복된다.

이를 바탕으로 코드를 0x68까지 한 번씩 실행했을 때 register에 저장되는 예상 결과를 구할 수 있다.

$r0	0x 0000 0000	$r1	0x 1234 5678
$r2	0x 78ff 4321	$r3	0x 8765 4321
$r4	0x 0000 0000	$r5	0x ffff ffff
$r6	0x 0000 0221	$r7	0x 8f37 5310
$r8	0x 0000 0110	$r9	0x 0000 0220
$r10	0x 0000 0001	$r31	0x 0000 0040
hi	0x 0000 0080	lo	0x 8f37 5310

C. 문제점 및 개선점

Single Cycle CPU는 모든 명령어를 1 clock cycle에 처리하기 때문에 clock 주기를 시간이 가장 오래 걸리는 명령어의 길이에 따라 충분히 길게 결정해야한다. 짧은 시간에 모든 연산을 끝내는 동작의 경우 연산을 종료한 뒤 다음 cycle까지의 대기시간이 길어진다. 따라서 Single cycle이 아닌 multi cycle을 이용하도록 개선하면 CPU가 모든 명령어를 실행하는데 걸리는 시간을 단축할 수 있을 것이다.

2. 설계 의도와 방법

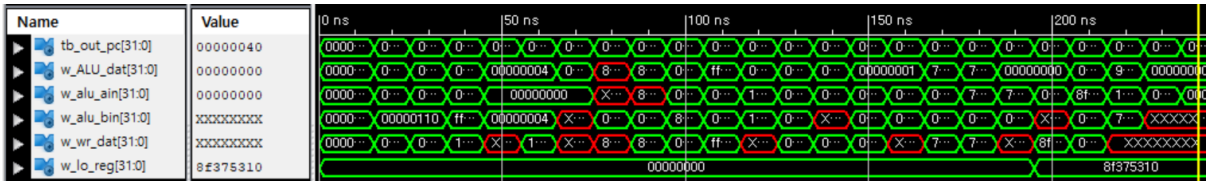
A. 구현한 Single Cycle CPU 블록도

제시된 SingleCycle.v를 변형없이 이용하였다.

B. 전체 testbench 비교분석

제공된 tb_SingleCycle.v의 tb_SingleCycle module을 수정하여 결과를 확인하였다. 해당 모듈의 output에 w_alu_ain, w_alu_bin, w_wr_dat, w_lo_reg를 추가하여 waveform을 확인하였다. 또한 프로그램의 pc가 0으로 되돌아가 반복실행되므로 register에 쓰인 결과를 확인하기 위해 simulation 시간을 모든 명령어가 한 번씩 수행된 이후인 240ns로 변경하여 결과를 확인하였다.

C. 시뮬레이션 결과와 예상 결과 비교분석

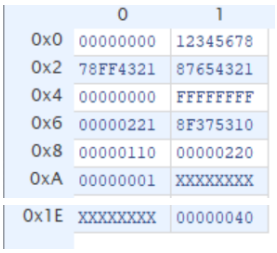


시뮬레이션 결과 pc의 값이 jump, branch등의 명령어에서도 예상대로 변경되는 것을 확인할 수 있었다. 또한 추가한 output을 이용하여 single cycle CPU의 동작을 자세히 살펴볼 수 있었다.

240ns 동안 프로그램을 실행한 뒤 register에 저장된 값은 다음과 같았다.

$r0	0x 0000 0000	$r1	0x 1234 5678
$r2	0x 78ff 4321	$r3	0x 8765 4321
$r4	0x 0000 0000	$r5	0x ffff ffff
$r6	0x 0000 0221	$r7	0x 8f37 5310
$r8	0x 0000 0110	$r9	0x 0000 0220
$r10	0x 0000 0001	$r31	0x 0000 0040
hi	0x 0000 0080	lo	0x 8f37 5310

<예상 값>



<실제 값>

예상한 값과 일치하는 결과를 얻을 수 있었다. 따라서 해당 프로젝트의 목표를 정상적으로 달성한 것을 확인할 수 있다.

3. 고찰

명령어의 동작에 따라 제시된 single cycle CPU의 data path를 적절하게 구성하는 문제였다. MIPS 명령어의 구조와 실제 쓰임을 확인할 수 있었다. 동작에 따라 원하는 data path를 구성하는 것은 큰 어려움이 없었으나 몇 가지 module과 signal에 대한 정확한 이해가 이루어지지 못한 점이 아쉽다. DatWidth signal을 이용하는 DM module이 어떤 동작을 하는지 정확하게 이해하지 못했고, Jump signal의 01 (Use Pseudo Jump address format)또한 w_jump_addr를 구하는 과정을 이해하는 것이 어려웠다.

4. 참고자료

이성원교수님, 컴퓨터구조 강의자료, 광운대학교 컴퓨터정보공학과 2021