

컴퓨터구조

# Computer Architecture – Project #1

## MIPS Single Cycle CPU

### Implementation

담당교수 : 이성원 교수님 (월3 수4)

2021202003 강준우

## Introduction

저번 과제를 통해서 mips 를 구현해보았고 이번 과제에서는 명령어를 추가하여 더 많은 기능을 구현하도록 설계하는 것입니다. 이번 과제에서는 10개의 명령어에 대한 설명 및 구현현과 PLA 유닛에 대한 구현을 포함하여 정리합니다.

## 명령어 설명

고찰에서 언급한 10개의 명령어는 다음과 같습니다.

AND, NOR, ADDI, SLTU, SRL, SH, LB, BNE, BGEZ, JALR

그리고 앞서 제공된 pla\_and와 pla\_or파일의 역할은 다음과 같습니다.

pla\_and : 명령어의 opcode /funct/regimm 를 설명합니다. 17비트 2진수로 표현합니다.

pla\_or: 명령어를 기반으로 cpu 동작 제어를 설명합니다.

## and

and 명령어는 각 자리에서 두 값이 모두 1일때만 1을 반환하는 로직을 가집니다.

opcode map에서 확인시 H: 100 L: 110에 위치해 있습니다. 또한 R type으로 funct 필드 사용하고 비트 이동을 하지 않습니다. 해당 정보를 바탕으로

PLA\_AND : 000000\_100100\_00000 // 0x15 : and 를 얻어냈습니다.

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

r타입 ,일반연산,값저장,r타입, data2사용, and, 자료참고, 접근x, 사용x , alu결과, 분기x ,점프x

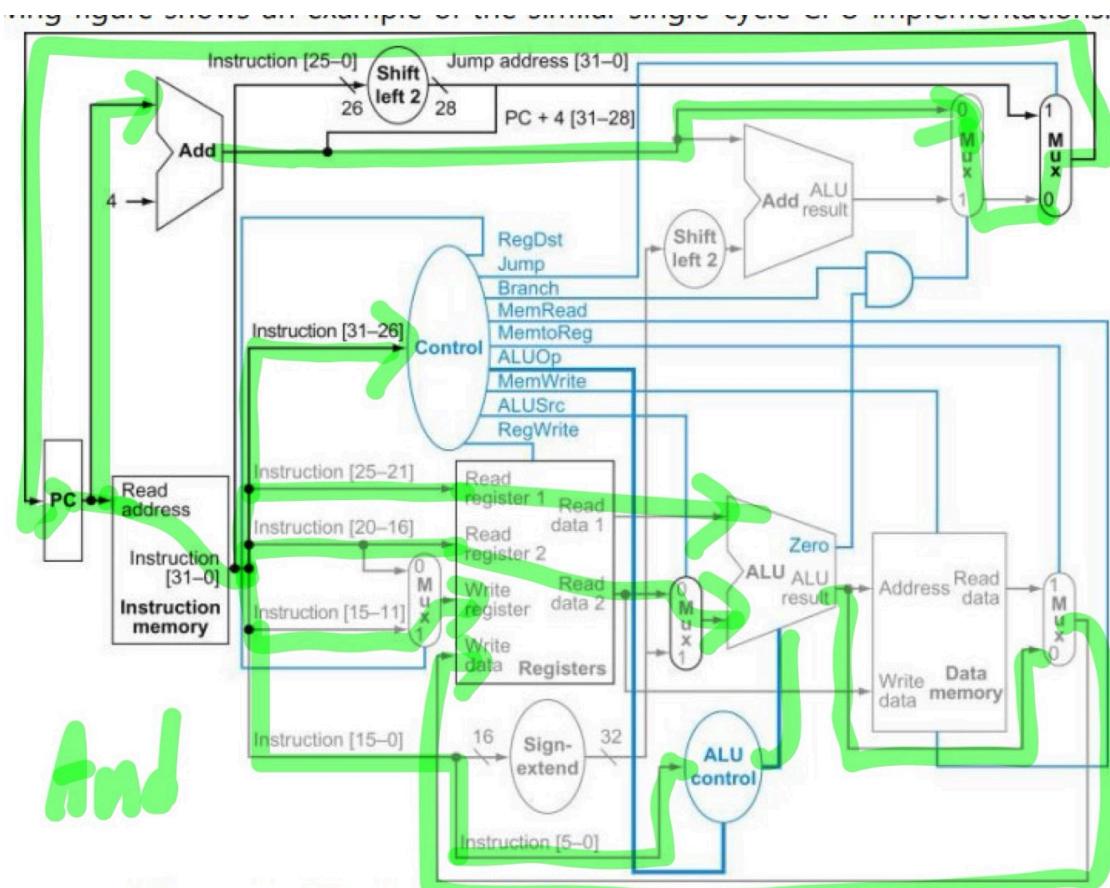


Figure 1 - The single cycle CPU datapath and control path

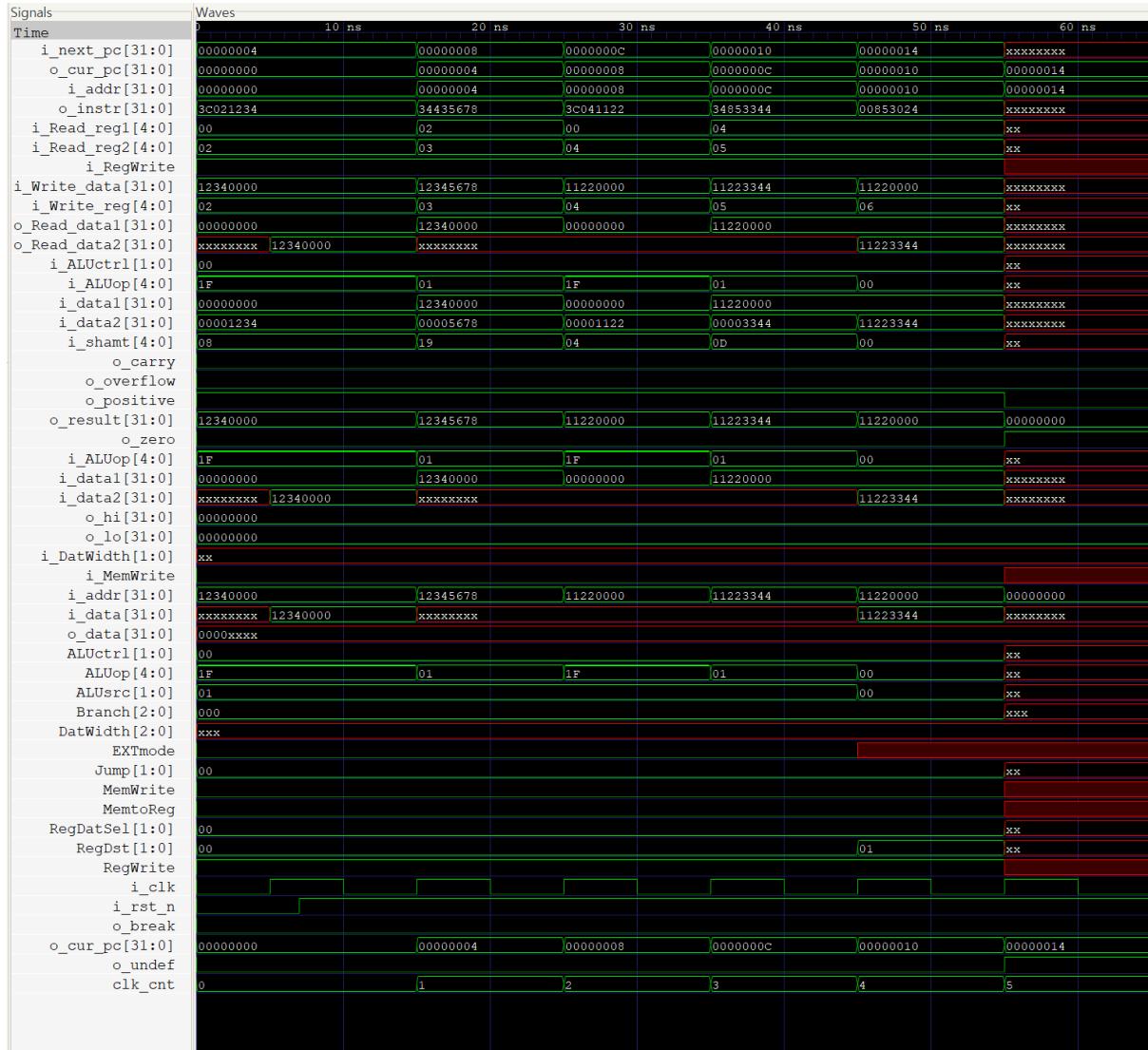
```

00111100_00000010_00010010_00110100 // lui $2, 0x1234
00110100_01000011_01010110_01111000 // ori $3, $2, 0x5678 ($3 = 0x12345678)
00111100_00000100_00010001_00100010 // lui $4, 0x1122
00110100_10000101_00110011_01000100 // ori $5, $4, 0x3344 ($5 = 0x11223344)

00000000_10000101_00110000_00100100 // and $6, $4, $5

```

해당 명령을 수행하면 4번&5번를 통해 값이 계산됨을 예상할 수 있습니다.



00000000\_10000101\_00110000\_00100100 명령어로 4번째와 5번째 레지스터 값을 AND  
통해서 4번 레지스터의 11220000&11223344 = 11220000을 구현한 모습입니다.

## NOR

NOR 명령어는 두값이 모두 1일때만 0을 반환하는 명령어입니다. H: 111 L: 100이고, 또한 R type으로 funct 필드 사용하고 비트 이동을 하지 않습니다. 해당 정보를 바탕으로 000000\_100111\_xxxxx // 0x18 : nor 를 얻어냅니다.

| RegDst | RegDatSel | RegWrite | SEUmod e | ALUsrcB | ALUctrl | ALUop  | DataWidt h | MemWrit e | MemoRe g | Branch | Jump |
|--------|-----------|----------|----------|---------|---------|--------|------------|-----------|----------|--------|------|
| 01     | 00        | 1        | x        | 00      | 00      | 000010 | xxx        | 0         | 0        | 000    | 00   |

r타입 ,일반연산,값저장,r타입, data2사용, and, 자료참고, 접근x,사용x , alu결과,분기x ,점프x

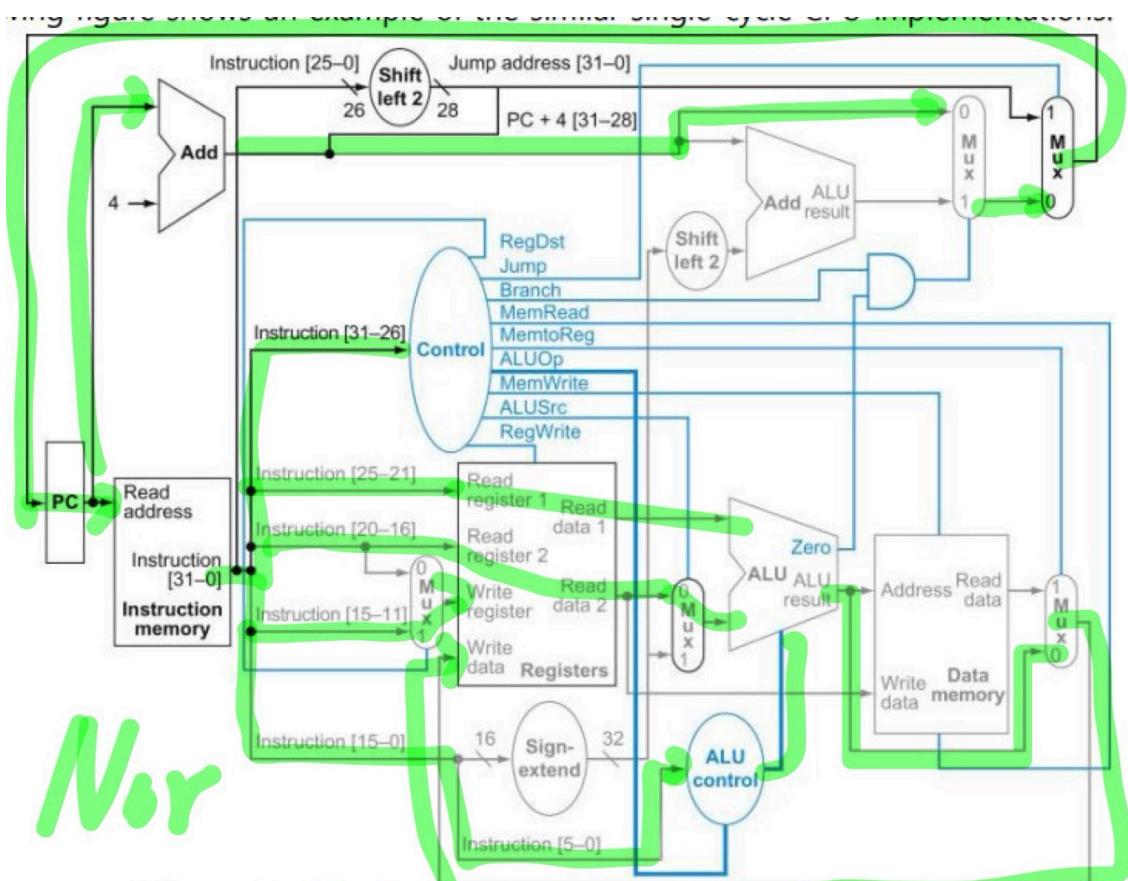


Figure 1 - The single cycle CPU datapath and control path

```

00111100_00000010_00010010_00110100 // lui $2, 0x1234
00110100_01000011_01010110_01111000 // ori $3, $2, 0x5678 ($3 = 0x12345678)
00111100_00000100_00010001_00100010 // lui $4, 0x1122
00110100_10000101_00110011_01000100 // ori $5, $4, 0x3344 ($5 = 0x11223344)

```

00000000\_10000101\_00110000\_00100111 // nor \$6, \$4, \$5

00010001\_00100010\_00000000\_00000000

00010001\_00100010\_00110011\_01000100

이때 nor은 둘다 1일때가 아닌 경우 모두 1처리하므로

11101110\_11011101\_11001100\_10111011를 예상할 수 있다.

| Time               | 10 ns    | 20 ns    | 30 ns    | 40 ns    | 50 ns    | 60 ns    | 70 ns    |
|--------------------|----------|----------|----------|----------|----------|----------|----------|
| i_next_pc[31:0]    | 00000004 | 00000008 | 0000000C | 00000010 | 00000014 | xxxxxxxx |          |
| o_cur_pc[31:0]     | 00000000 | 00000004 | 00000008 | 0000000C | 00000010 | 00000014 | xxxxxxxx |
| i_addr[31:0]       | 00000000 | 00000004 | 00000008 | 0000000C | 00000010 | 00000014 | xxxxxxxx |
| o_instr[31:0]      | 3C021234 | 34435678 | 3C041122 | 34553344 | 00853027 | xxxxxxxx |          |
| i_Read_Reg1[4:0]   | 00       | 02       | 00       | 04       | 05       | xx       |          |
| i_Read_Reg2[4:0]   | 02       | 03       | 04       | 05       | xx       |          |          |
| i_RegWrite         |          |          |          |          |          |          |          |
| i_Write_Data[31:0] | 12340000 | 12345678 | 11220000 | 11223344 | EEDDCCB  | xxxxxxxx |          |
| i_Write_Reg[4:0]   | 02       | 03       | 04       | 05       | 06       | xx       |          |
| o_Read_Data1[31:0] | 00000000 | 12340000 | 00000000 | 11220000 |          | xxxxxxxx |          |
| o_Read_Data2[31:0] | xxxxxxxx | 12340000 | xxxxxxxx |          | 11223344 | xxxxxxxx |          |
| i_ALUctrl[1:0]     | 00       |          |          |          |          | xx       |          |
| i_ALUOp[4:0]       | 1F       | 01       | 1F       | 01       | 02       | xx       |          |
| i_data1[31:0]      | 00000000 | 12340000 | 00000000 | 11220000 |          | xxxxxxxx |          |
| i_data2[31:0]      | 00001234 | 00005678 | 00001122 | 00003344 | 11223344 | xxxxxxxx |          |
| i_shamt[4:0]       | 08       | 19       | 04       | 0D       | 00       | xx       |          |
| i_carry            |          |          |          |          |          |          |          |
| i_overflow         |          |          |          |          |          |          |          |
| i_positive         |          |          |          |          |          |          |          |
| o_result[31:0]     | 12340000 | 12345678 | 11220000 | 11223344 | EEDDCCB  | 00000000 |          |
| o_zero             |          |          |          |          |          |          |          |
| i_ALUOp[4:0]       | 1F       | 01       | 1F       | 01       | 02       | xx       |          |
| i_data1[31:0]      | 00000000 | 12340000 | 00000000 | 11220000 |          | xxxxxxxx |          |
| i_data2[31:0]      | xxxxxxxx | 12340000 | xxxxxxxx |          | 11223344 | xxxxxxxx |          |
| o_hi[31:0]         | 00000000 |          |          |          |          |          |          |
| o_lo[31:0]         | 00000000 |          |          |          |          |          |          |
| i_DatWidth[1:0]    | xx       |          |          |          |          |          |          |
| i_MemWrite         |          |          |          |          |          |          |          |
| i_addr[31:0]       | 12340000 | 12345678 | 11220000 | 11223344 | EEDDCCB  | 00000000 |          |
| i_data[31:0]       | xxxxxxxx | 12340000 | xxxxxxxx |          | 11223344 | xxxxxxxx |          |
| o_data[31:0]       | 00000000 |          |          |          |          |          |          |
| ALUctrl[1:0]       | 00       |          |          |          |          | xx       |          |
| ALUOp[4:0]         | 1F       | 01       | 1F       | 01       | 02       | xx       |          |
| ALUsrc[1:0]        | 01       |          |          |          | 00       | xx       |          |
| Branch[2:0]        | 000      |          |          |          |          | xxx      |          |
| DatWidth[2:0]      | xxx      |          |          |          |          |          |          |
| EXTmode            |          |          |          |          |          |          |          |
| Jump[1:0]          | 00       |          |          |          |          | xx       |          |
| MemWrite           |          |          |          |          |          |          |          |
| MemtoReg           |          |          |          |          |          |          |          |
| RegDatSel[1:0]     | 00       |          |          |          |          |          |          |
| RegDst[1:0]        | 00       |          |          |          | 01       | xx       |          |
| RegWrite           |          |          |          |          |          |          |          |
| i_clk              |          |          |          |          |          |          |          |
| i_rst_n            |          |          |          |          |          |          |          |
| o_break            |          |          |          |          |          |          |          |
| o_cur_pc[31:0]     | 00000000 | 00000004 | 00000008 | 0000000C | 00000010 | 00000014 | xxxxxxxx |
| o_undef            |          |          |          |          |          |          |          |
| clk_cnt            | 0        | 1        | 2        | 3        | 4        | 5        | 6        |

결과에서 o\_result에서 EEDDCCB = 11101110\_11011101\_11001100\_10111011을 확인했습니다.

## ADDI

ADDI 명령어는 즉시 덧셈을 구현합니다. 오피코드 001000이고, 또한 I type과 이후 데이터에 의존적임을 바탕으로 001000\_xxxxxx\_xxxxx // 0x23 : addi 를 얻어냅니다.

| RegDst | RegDatSel | RegWrite | SEUmod | ALUsrcB | ALUctrl | ALUop | DataWidth | MemWrit | MemtoRe | Branch | Jump |
|--------|-----------|----------|--------|---------|---------|-------|-----------|---------|---------|--------|------|
| 00     | 00        | 1        | 1      | 01      | 00      | 00100 | xxx       | 0       | 0       | xxx    | xx   |

i타입, 일반연산, 값저장, i타입, signExt, and, 자료참고, 접근x, 사용x, alu결과, 분기x, 점프x

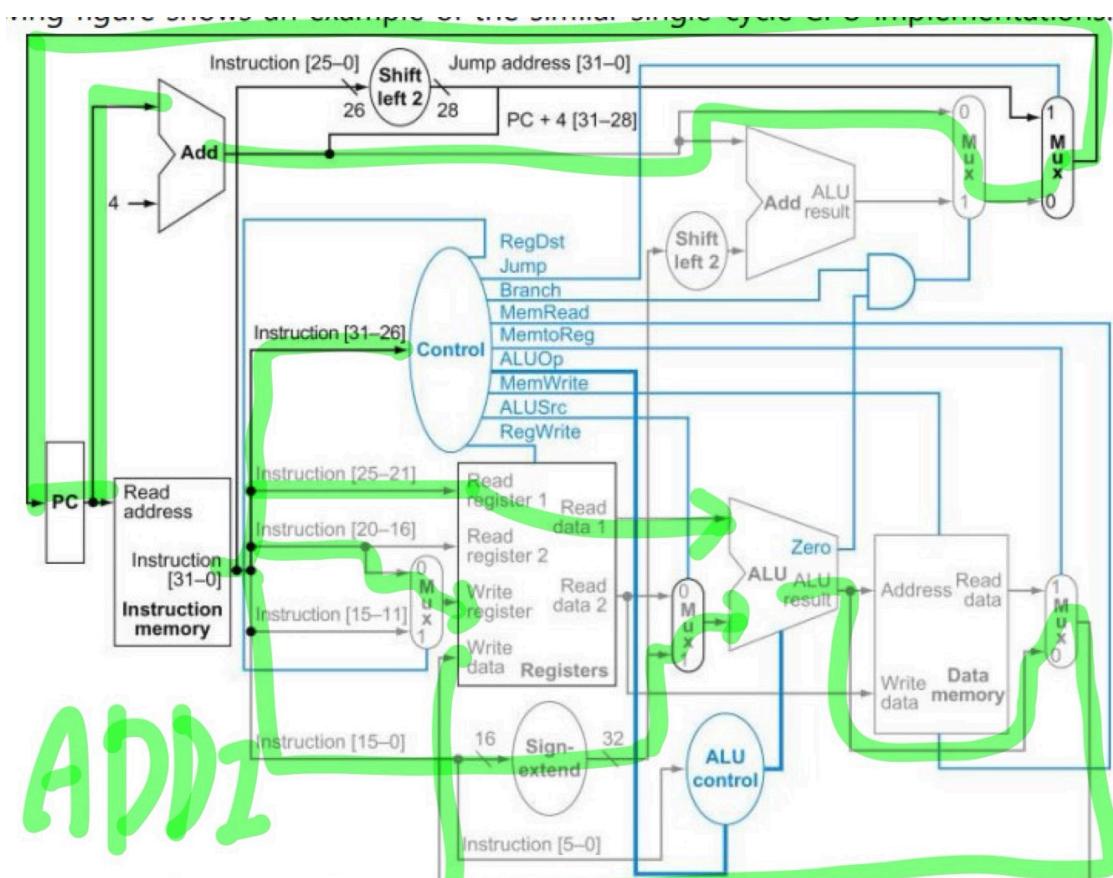


Figure 1 - The single cycle CPU datapath and control path

```

00111100_00000010_00010010_00110100 // lui $2, 0x1234
00110100_01000011_01010110_01111000 // ori $3, $2, 0x5678 ($3 = 0x12345678)
00111100_00000100_00010001_00100010 // lui $4, 0x1122
00110100_10000101_00110011_01000100 // ori $5, $4, 0x3344 ($5 = 0x11223344)

```

00100000\_10100110\_00000000\_00010001 // addi \$6, \$5, 10001

해당 명령어를 통해 11223344를 11223355로 더해지기를 예상합니다.

| Time                 | 10 ns    | 20 ns    | 30 ns    | 40 ns    | 50 ns    | 60 ns    |
|----------------------|----------|----------|----------|----------|----------|----------|
| i_next_pc[31:0] =    | 00000004 | 00000008 | 0000000C | 00000010 | xxxxxxxx |          |
| o_cur_pc[31:0] =     | 00000000 | 00000004 | 00000008 | 0000000C | 00000010 | xxxxxxxx |
| i_addr[31:0] =       | 00000000 | 00000004 | 00000008 | 0000000C | 00000010 | xxxxxxxx |
| o_instr[31:0] =      | 3C021234 | 34435678 | 3C041122 | 34853344 | 20A60011 | xxxxxxxx |
| i_Read_reg1[4:0] =   | 00       | 02       | 00       | 04       | 05       | xx       |
| i_Read_reg2[4:0] =   | 02       | 03       | 04       | 05       | 06       | xx       |
| i_RegWrite =         |          |          |          |          |          |          |
| i_Write_data[31:0] = | 12340000 | 12345678 | 11220000 | 11223344 | 11223355 | xxxxxxxx |
| i_Write_reg[4:0] =   | 02       | 03       | 04       | 05       | 06       | xx       |
| o_Read_data1[31:0] = | 00000000 | 12340000 | 00000000 | 11220000 | 11223344 | xxxxxxxx |
| o_Read_data2[31:0] = | xxxxxxxx | 12340000 | xxxxxxxx |          |          |          |
| i_ALUctrl[1:0] =     | 00       |          |          |          |          | xx       |
| i_ALUop[4:0] =       | 1F       | 01       | 1F       | 01       | 04       | xx       |
| i_data1[31:0] =      | 00000000 | 12340000 | 00000000 | 11220000 | 11223344 | xxxxxxxx |
| i_data2[31:0] =      | 00001234 | 00005678 | 00001122 | 00003344 | 00000011 | xxxxxxxx |
| i_shamt[4:0] =       | 08       | 19       | 04       | 0D       | 00       | xx       |
| o_carry =            |          |          |          |          |          |          |
| o_overflow =         |          |          |          |          |          |          |
| o_positive =         |          |          |          |          |          |          |
| o_result[31:0] =     | 12340000 | 12345678 | 11220000 | 11223344 | 11223355 | 00000000 |
| o_zero =             |          |          |          |          |          |          |
| i_ALUop[4:0] =       | 1F       | 01       | 1F       | 01       | 04       | xx       |
| i_data1[31:0] =      | 00000000 | 12340000 | 00000000 | 11220000 | 11223344 | xxxxxxxx |
| i_data2[31:0] =      | xxxxxxxx | 12340000 | xxxxxxxx |          |          |          |
| o_hi[31:0] =         | 00000000 |          |          |          |          |          |
| o_lo[31:0] =         | 00000000 |          |          |          |          |          |
| i_DatWidth[1:0] =    | xx       |          |          |          |          |          |
| i_MemWrite =         |          |          |          |          |          |          |
| i_addr[31:0] =       | 12340000 | 12345678 | 11220000 | 11223344 | 11223355 | 00000000 |
| i_data[31:0] =       | xxxxxxxx | 12340000 | xxxxxxxx |          |          |          |
| o_data[31:0] =       | 0000xxxx |          |          |          |          |          |
| ALUctrl[1:0] =       | 00       |          |          |          |          | xx       |
| ALUop[4:0] =         | 1F       | 01       | 1F       | 01       | 04       | xx       |
| ALUsrc1[1:0] =       | 01       |          |          |          |          | xx       |
| Branch[2:0] =        | 000      |          |          |          |          | xxx      |
| DatWidth[2:0] =      | xxx      |          |          |          |          |          |
| EXTmode =            |          |          |          |          |          |          |
| Jump[1:0] =          | 00       |          |          |          |          | xx       |
| MemWrite =           |          |          |          |          |          |          |
| MemtoReg =           |          |          |          |          |          |          |
| RegDatSel[1:0] =     | 00       |          |          |          |          | xx       |
| RegDst[1:0] =        | 00       |          |          |          |          | xx       |
| RegWrite =           |          |          |          |          |          |          |
| i_clk =              |          |          |          |          |          |          |
| i_rst_n =            |          |          |          |          |          |          |
| o_break =            |          |          |          |          |          |          |
| o_cur_pc[31:0] =     | 00000000 | 00000004 | 00000008 | 0000000C | 00000010 | xxxxxxxx |
| o_undef =            |          |          |          |          |          |          |
| clk_cnt =            | 0        | 1        | 2        | 3        | 4        | 5        |

결과에서 o\_result에서 11223355를 확인했습니다.

## SLTU

SLTU 명령어는 두 레지스터 값을 비교하여 \$rs < \$rt일 때 결과를 1로, 아니면 0으로 설정합니다.

오피코드는 000000이고, Funct 필드는 101011이므로, 이후 데이터와 결합하여 000000\_101011\_xxxx // 0x1a : sltu

| RegDst | RegDatSel | RegWrite | SEUmod | ALUsrcB | ALUctrl | ALUop | DataWidth | MemWrite | MemtoReg | Branch | Jump |
|--------|-----------|----------|--------|---------|---------|-------|-----------|----------|----------|--------|------|
| 01     | 00        | 1        | 0      | 00      | 00      | 10001 | xxx       | 0        | 0        | xxx    | xx   |

r타입 ,일반연산,값저장,r타입,data2사용, and, 자료참고, 접근x,사용x ,alu결과,분기x ,점프x

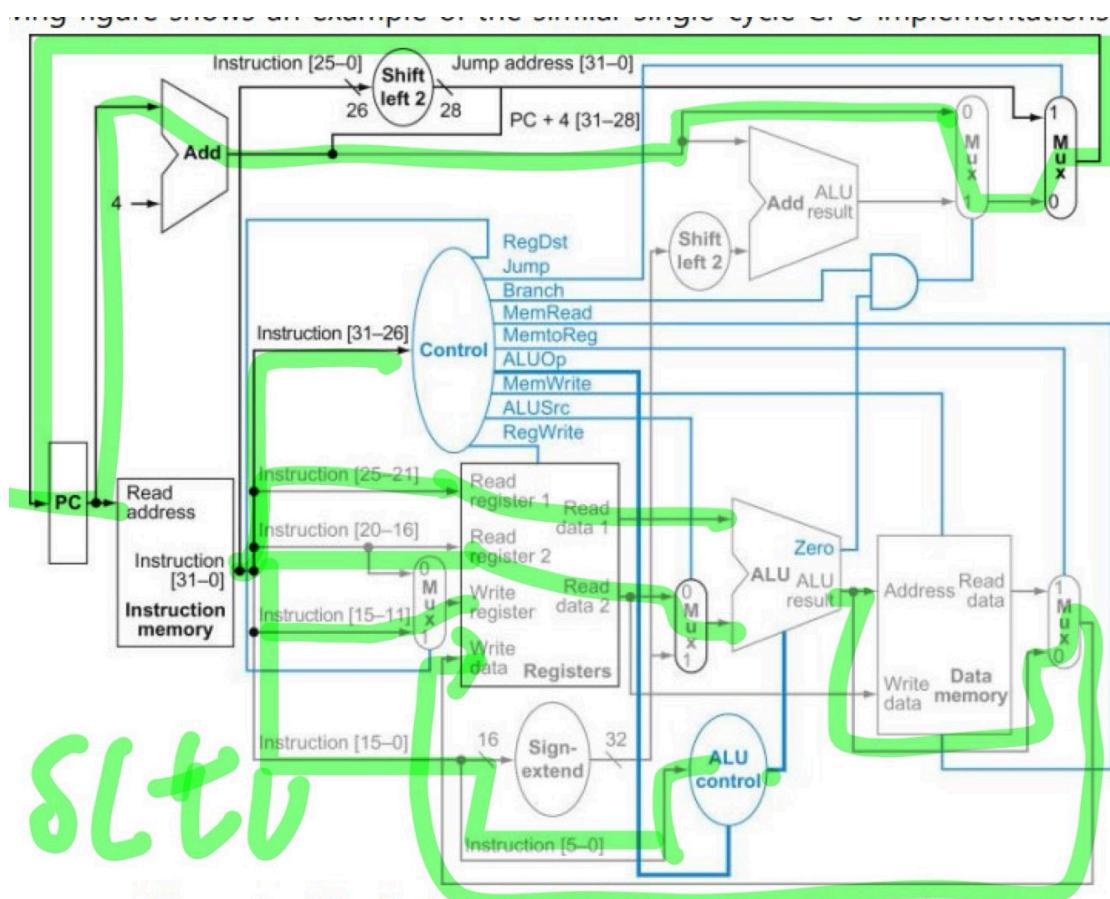


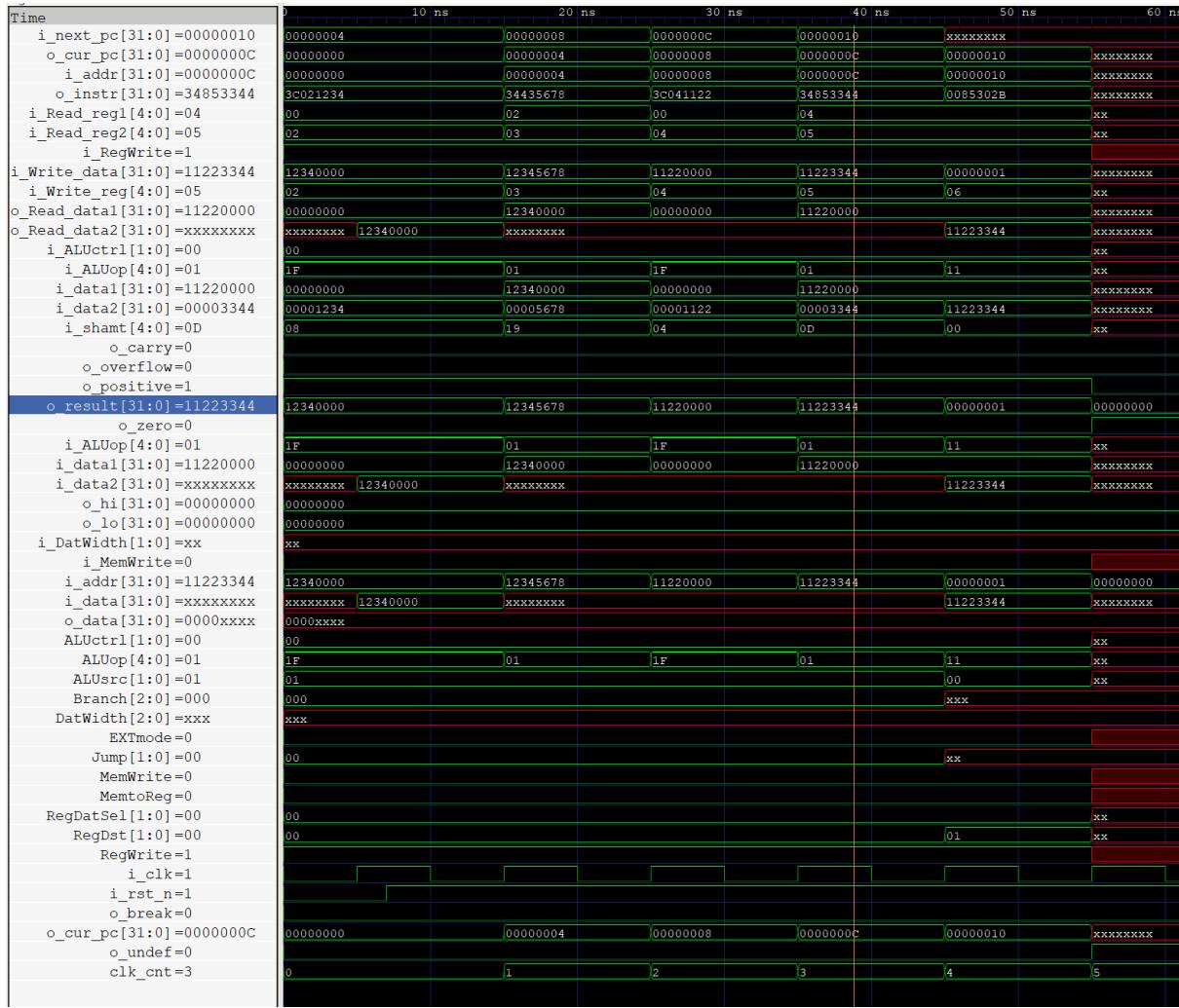
Figure 1 - The single cycle CPU datapath and control path

```

00111100_00000010_00010010_00110100 // lui $2, 0x1234
00110100_01000011_01010110_01111000 // ori $3, $2, 0x5678 ($3 = 0x12345678)
00111100_00000100_00010001_00100010 // lui $4, 0x1122
00110100_10000101_00110011_01000100 // ori $5, $4, 0x3344 ($5 = 0x11223344)

```

00000000\_10000101\_00110000\_00101011 // sltu \$4, \$5 \$6  
\$4 <\$5라면 1을 반환하게 설계했습니다. 11220000<11223344로 1을 예측합니다.



o\_result에서 1을 확인했습니다.

## SRL

SRL 명령어는 레지스터를 오른쪽으로 시프트 합니다. 오피코드는 000000이고, funct 필드는 0000100이므로, 이후 데이터와 결합하여 000000\_000010\_xxxxx // 0x01 : srl 을 얻어냅니다.

| RegDst | RegDatSel | RegWrite | SEUmode | ALUsrcB | ALUctrl | ALUop | DataWidth | MemWrite | MemtoReg | Branch | Jump |
|--------|-----------|----------|---------|---------|---------|-------|-----------|----------|----------|--------|------|
| 01     | 00        | 1        | x       | 00      | 00      | 01110 | xxx       | 0        | 0        | xxx    | xx   |

r타입, 일반연산, 값저장, r타입, data2사용, and, 자료참고, 접근x, 사용x, alu결과, 분기x, 점프x

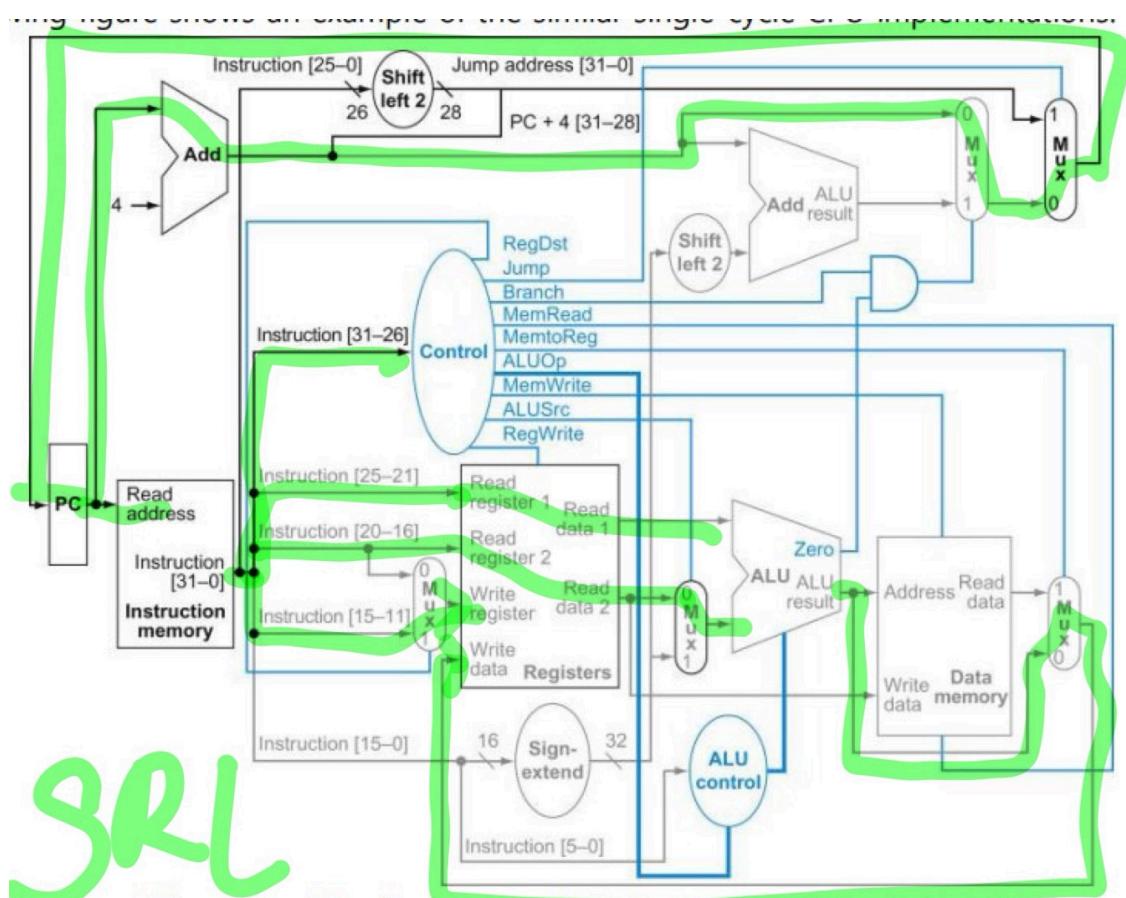


Figure 1 - The single cycle CPU datapath and control path

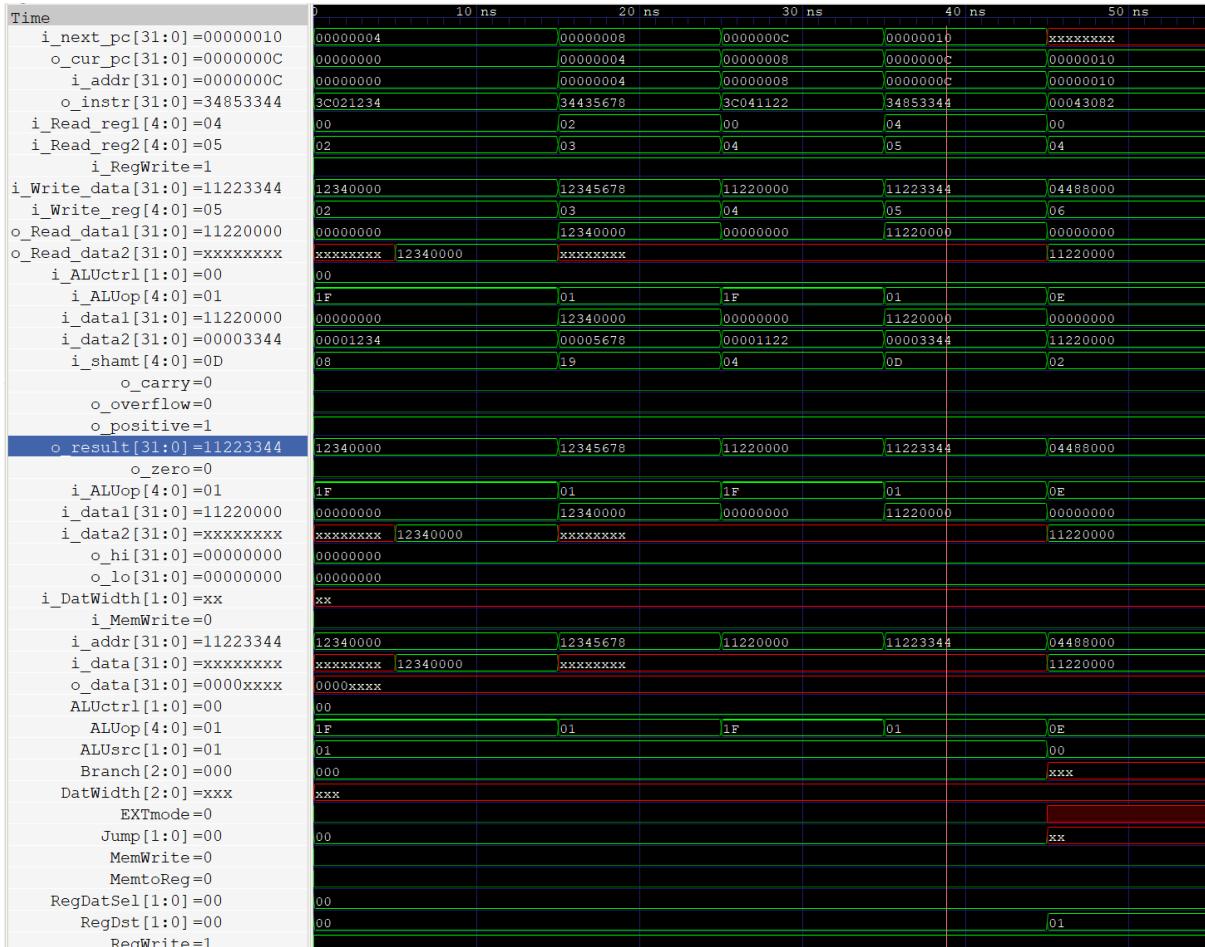
```

00111100_00000010_00010010_00110100 // lui $2, 0x1234
00110100_01000011_01010110_01111000 // ori $3, $2, 0x5678 ($3 = 0x12345678)
00111100_00000100_00010001_00100010 // lui $4, 0x1122
00110100_10000101_00110011_01000100 // ori $5, $4, 0x3344 ($5 = 0x11223344)

```

00000000\_00000100\_00110000\_10000010 // srl \$6, \$4, 2

1122인 4번 레지스터를 2번 시프트 즉 4배처리해서 4488을 예상합니다.



결과적으로 시프트 과정에서 상위0 추가를 통해서 044880이 구현되었다.

## SH

SH 명령어는 rt에 저장된 하프워드(16비트) 값을 메모리 주소 rs + immediate에 저장하는 명령어입니다. I-type 명령어이며, 오피코드는 101001이므로 101001\_xxxxxx\_xxxxx // 0x31 : sh

을 얻을 수 있습니다.

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

i타입, 일반연산, 값저장, i타입, signExt, and, 자료참고, 작성허용, 사용, alu결과, 분기x, 점프x

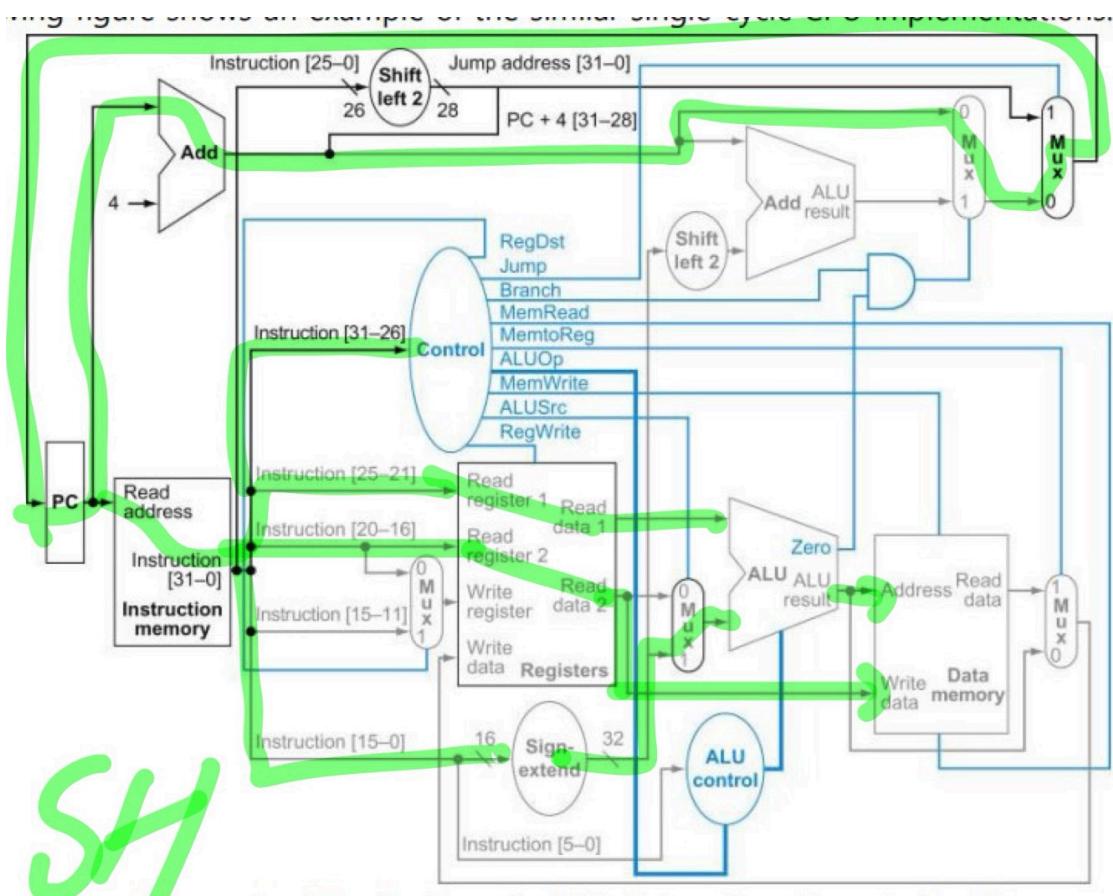


Figure 1 - The single cycle CPU datapath and control path

```

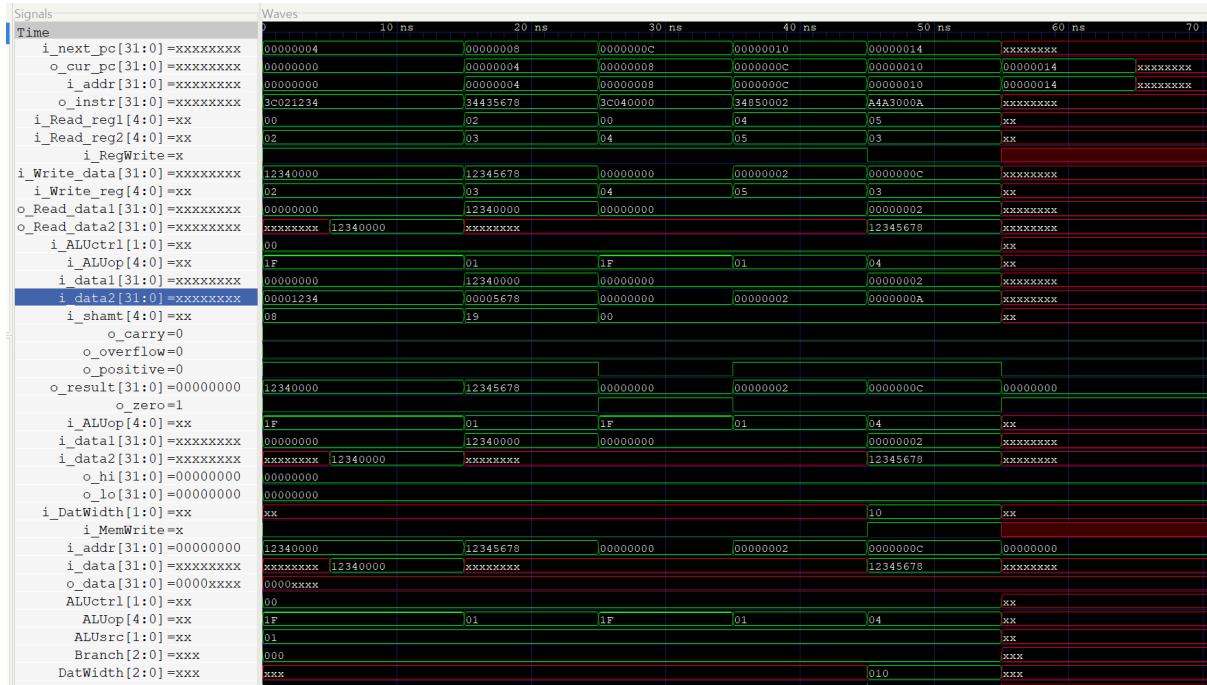
00111100_00000010_00010010_00110100 // lui $2, 0x1234
00110100_01000011_01010110_01111000 // ori $3, $2, 0x5678 ($3 = 0x12345678)
00111100_00000100_00000000_00000000 // lui $4, 0x0000
00110100_10000101_00000000_00000010 // ori $5, $4, 0x0010

10100100_10100011_00000000_00001010 // sh $3, 10($5)

```

해당 명령어는 \$5에 00000002라는 값으로 초기화시킨 후 이후 해당 값과 + 10을 하여 메모리의 주소에 접근합니다. 이후 \$3에 있는 하위 4개 (5678)을 저장시키려고 합니다. 12(ten)이므로

803번 mem\_dump에 00005678이 저장되기를 예상해봅니다.



먼저 결과 웨이브에서 0002 값과 12345678값을 가져온것을 확인할수 있습니다. 이후 메모리에서 확인하보면 803번 주소에 5678이 저당되어 있음을 확인할 수 있습니다.

```
00000800 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000801 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000802 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000803 : xxxxxxxx_xxxxxxxx_01010110_01111000 : xxxx5678
00000804 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000805 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000806 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000807 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000808 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
```

추가적으로 더하는 값이 훌수일때도 결과가 같은 경우가 발생합니다, 예를들어 합산값이 12가 아닌 13인 경우도 같은 주소에 저당이 되는데요, 이상적인 계산을 위해서 짹수 단위로 확인하였습니다.

이후 추가 결과로 offset이 4씩 가감될때 주소가 바뀌는 것을 확인하였습니다.

## LB

lb (Load Byte) 명령어는 메모리 주소 [ $\$s + offset$ ]의 1바이트를 부호 확장하여 \$t 레지스터에 저장합니다. opcode 를 통해 100000, 이후 i type 및 데이터와 결합하여 100000\_000000\_xxxxx // 0x20 : lb 를 얻어냅니다.

| RegDst | RegDataSel | RegWrite | SEUmode | ALUSrcB | ALUctrl | ALUop | DataWidth | MemWrite | MemtoReg | Branch | Jump |
|--------|------------|----------|---------|---------|---------|-------|-----------|----------|----------|--------|------|
| 00     | 00         | 1        | 1       | 01      | 00      | 00100 | 111       | 0        | 1        | 000    | 00   |

i타입, 일반연산, 값저장, i타입, signExt, and, 자료참고, 작성허용, 사용x, alu결과, 분기x, 점프x

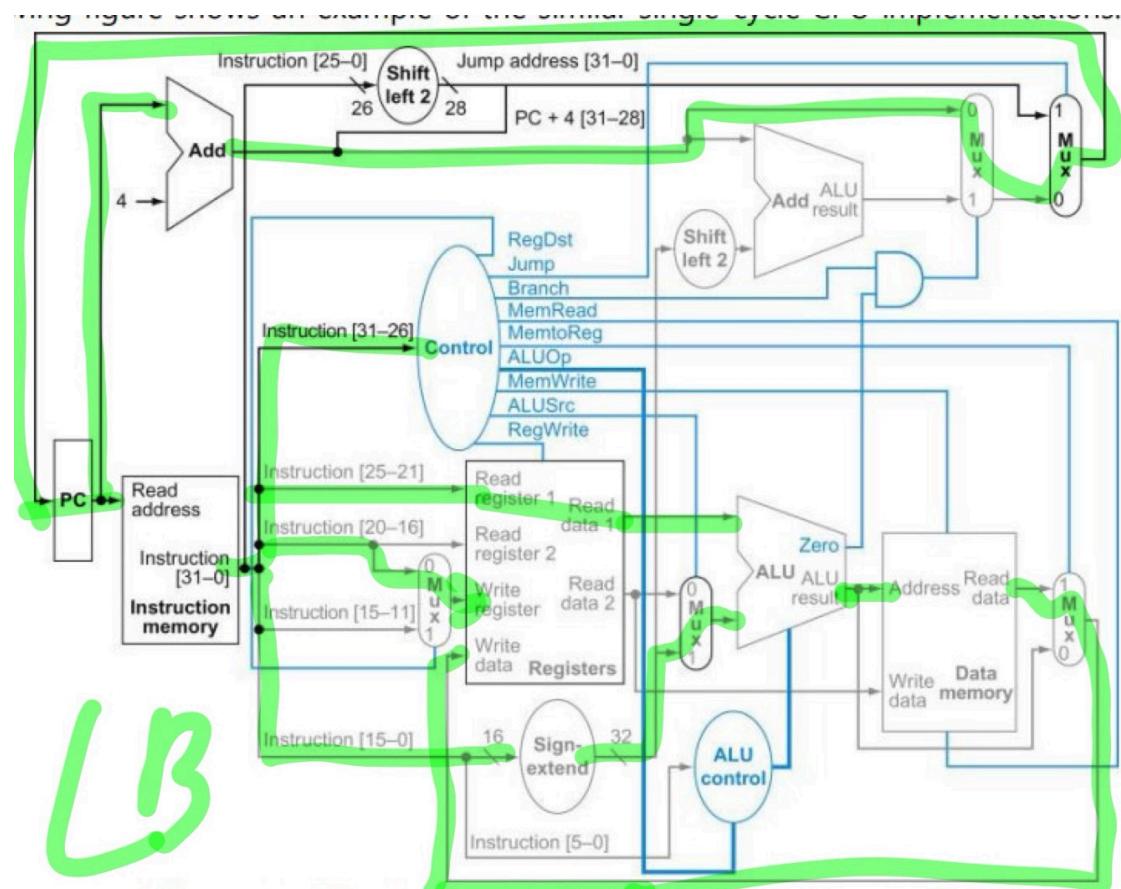


Figure 1 - The single cycle CPU datapath and control path

00111100\_00000010\_00010010\_00110100 // lui \$2, 0x1234

00110100\_01000011\_01010110\_01111000 // ori \$3, \$2, 0x5678 (\$3 = 0x12345678)

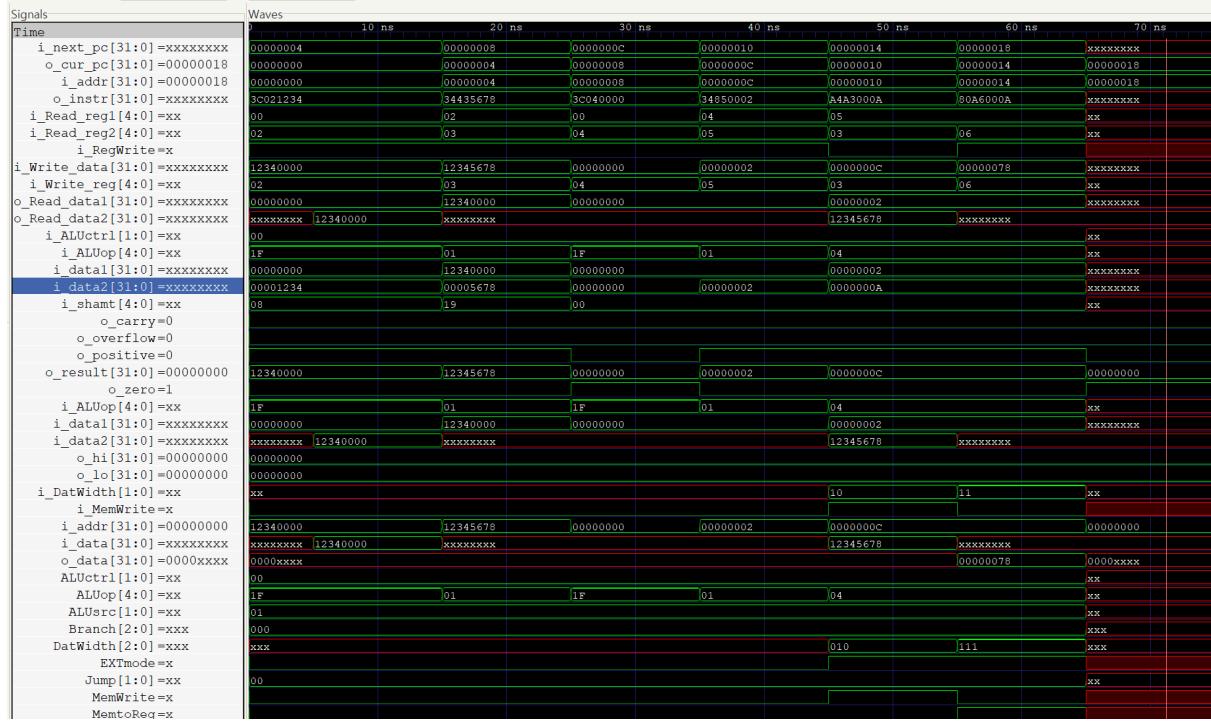
00111100\_00000100\_00000000\_00000000 // lui \$4, 0x0000

00110100\_10000101\_00000000\_00000010 // ori \$5, \$4, 0x0010

10100100\_10100011\_00000000\_00001010 // sh \$3, 10(\$5)

10000000\_10100110\_00000000\_00001010 // lb \$6, 10(\$5)

해당 코드는 앞선 sh 수행 코드에서의 연장선으로  $10+2 = 12$ ,  $12/4 = 3$ 으로 803번 주소에 저장된 5678데이터를 읽지만 1바이트만 읽기때문에 2개인 뒤에 78만 가져와서 \$6에 저장되기를 예상합니다.



write에 78이 있는것을 확인할 수 있습니다.

## BNE

bne는 레지스터의 값을 비교하여 같다면 주소이동을 수행하는 명령어입니다.

Opcode 000101과 i type을 참고하여 000101\_xxxxxx\_xxxxx // 0x20 : bne 를 구현할 수 있습니다.

| RegDst | RegDataSel | RegWrite | SEUmode | ALUsrcB | ALUctrl | ALUop | DataWidth | MemWrite | MemtoReg | Branch | Jump |
|--------|------------|----------|---------|---------|---------|-------|-----------|----------|----------|--------|------|
| xx     | 11         | 0        | 1       | 00      | 00      | 00110 | xxx       | 0        | x        | 101    | 00   |

사용x 사용x 레지스터 즉시값 data2 sub 자료참고 사용x 저장x 상관x  
bne사용 점프x

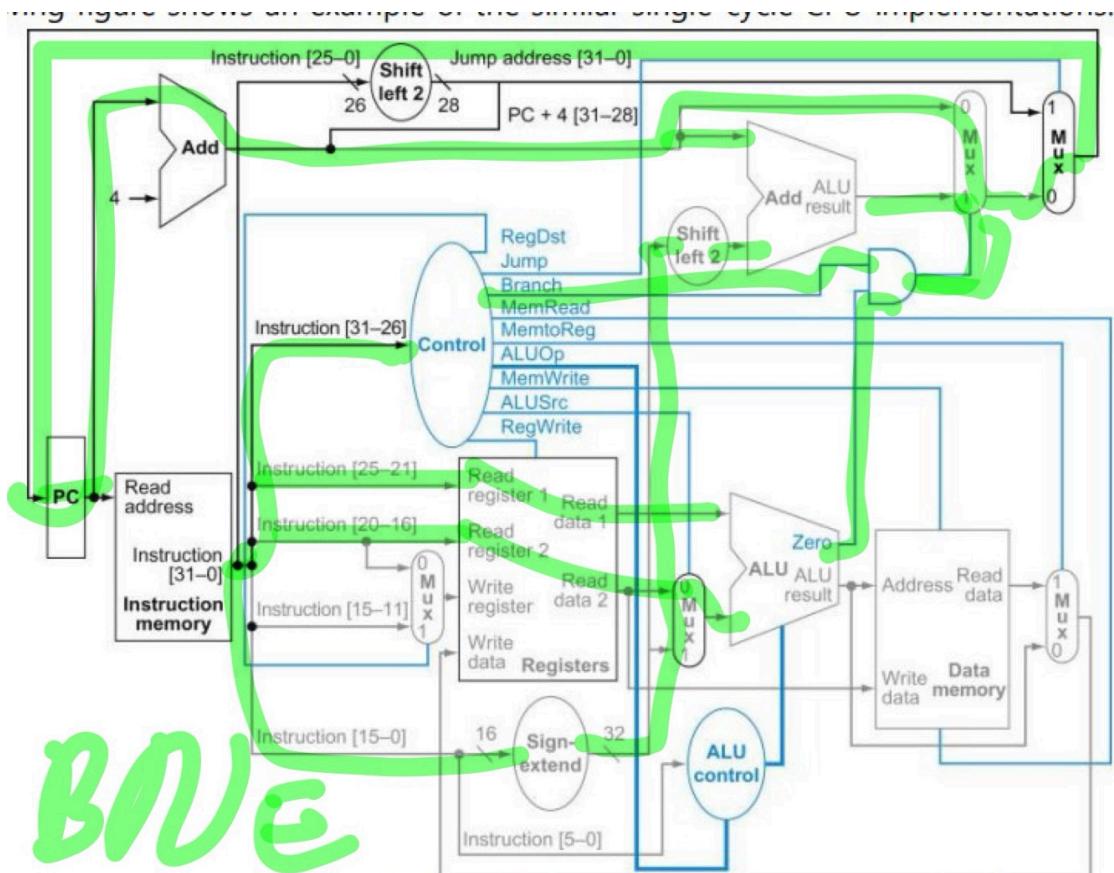


Figure 1 - The single cycle CPU datapath and control path

```

00111100_00000010_00010010_00110100 // lui $2, 0x1234
00110100_01000011_01010110_01111000 // ori $3, $2, 0x5678 ($3 = 0x12345678)
00111100_00000100_00010001_00100010 // lui $4, 0x1122
00110100_10000101_00110011_01000100 // ori $5, $4, 0x3344 ($5 = 0x11223344)

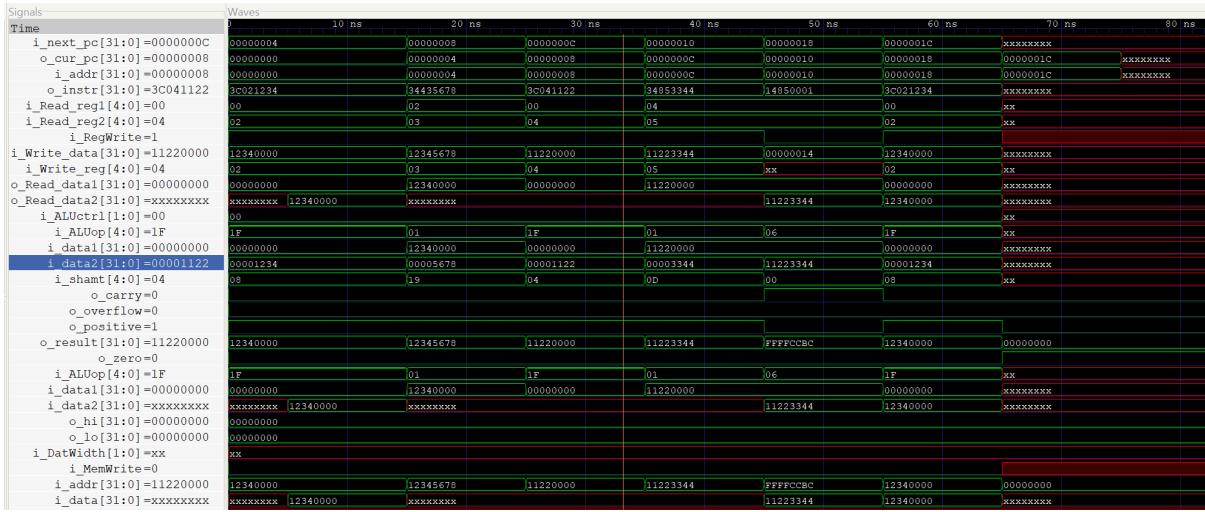
```

```

00010100_10000101_00000000_00000001 // bne $4, $5, 1
xxxxxxxx_xxxxxxxxx_xxxxxxxxx_xxxxxxxxx
00111100_00000010_00010010_00110100

```

해당 코드는 의도적으로 4번과 5번에 다른 값을 넣어두고, 이후에 비교를 통해 주소 이동을 구현해본 코드입니다. 이 코드를 통해 PC는  $>>1 + 4$  한 결과로 이동하도록 예상합니다.



예상대로 11223344와 11220000 값이 달라 14번이 아닌 18번으로 다음 pc주소가 이동한 것을 확인할 수 있었습니다.

반대로

00111100\_00000010\_00010010\_00110100 // lui \$2, 0x1234

00110100\_01000011\_01010110\_01111000 // ori \$3, \$2, 0x5678 (\$3 = 0x12345678)

00111100\_00000100\_00010001\_00100010 // lui \$4, 0x1122

00111100\_00000101\_00010001\_00100010 // lui \$5, 0x1122

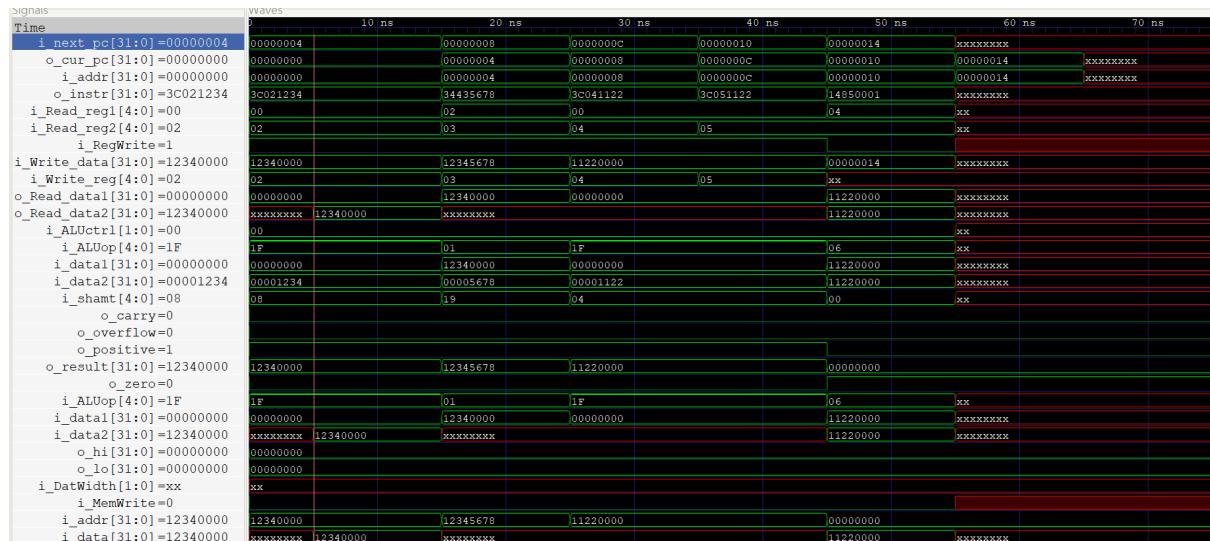
00010100\_10000101\_00000000\_00000001 // bne \$4, \$5, 1

xxxxxxxx\_xxxxxxx\_xxxxxxx\_xxxxxxx

00111100\_00000010\_00010010\_00110100

xxxxxxxx\_xxxxxxx\_xxxxxxx\_xxxxxxx

이렇게 4번과 5번 레지스터에 같은 값을 넣어두면 아래의 결과와 같이 기존의 PC+4인 14번 주소를 다음 주소로 가져가는것을 확인할 수 있습니다.



## BGEZ

bgez는 레지스터의 값이 0이상이면 분기하는 명령어입니다.

I타입과, 오피코드 00001을 바탕으로 xxxxxxx\_xxxxxxx\_00001 // 0x1c : bgez 를 구현할 수 있습니다.

| RegDst | RegDataSel | RegWrite | SEUmode | ALUSrcB | ALUctrl | ALUop | DataWidth | MemWrite | MemtoReg | Branch | Jump |
|--------|------------|----------|---------|---------|---------|-------|-----------|----------|----------|--------|------|
| 00     | 00         | 0        | 1       | 10      | 01      | 00100 | 000       | 0        | 0        | 111    | 00   |

i타입 ,일반연산,값저장,i타입,값비교,or,자료참고,작성x,사용x,메모리값,분기o,점프x

Figure 1 shows an example of the single cycle CPU datapath implementation.

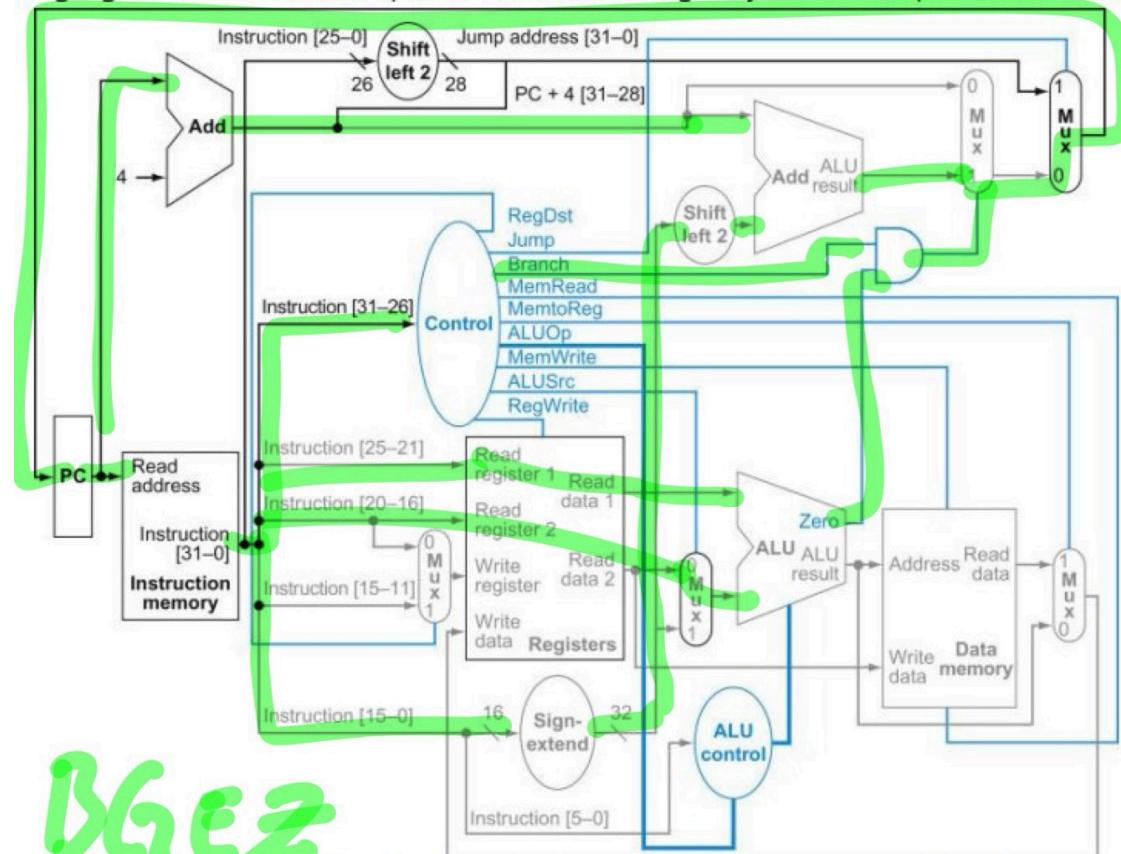


Figure 1 - The single cycle CPU datapath and control path

00111100\_00000010\_00010010\_00110100 // lui \$2, 0x1234

00110100\_01000011\_01010110\_01111000 // ori \$3, \$2, 0x5678 (\$3 = 0x12345678)

00111100\_00000100\_00010001\_00100010 // lui \$4, 0x1122

00110100\_10000101\_00110011\_01000100 // ori \$5, \$4, 0x3344 (\$5 = 0x11223344)

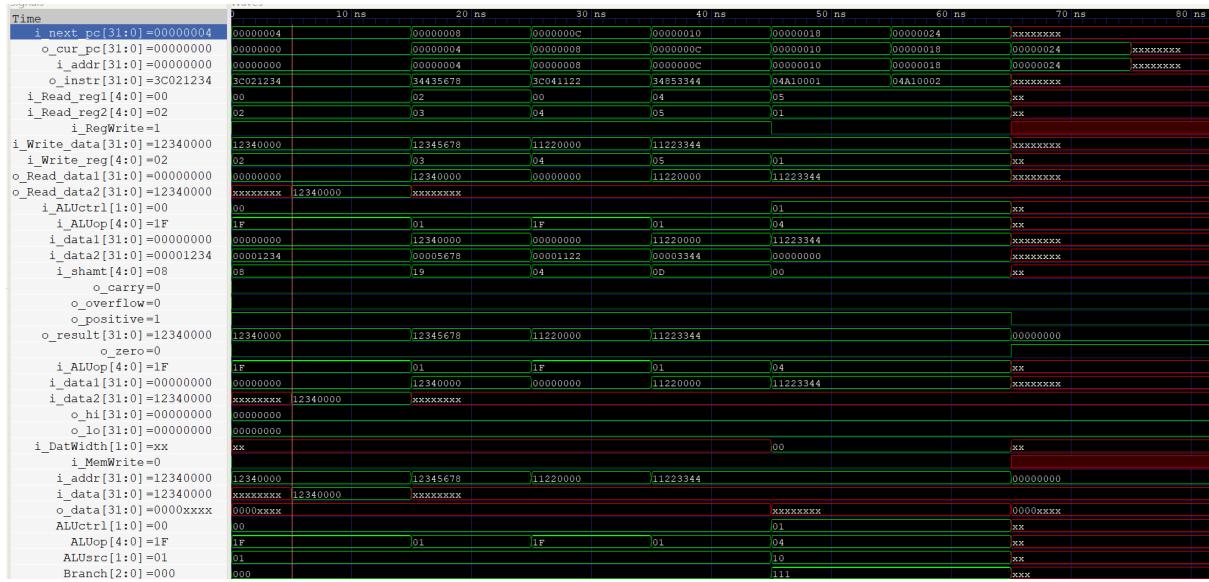
00000100\_10100001\_00000000\_00000001 // bgez \$5, 1

xxxxxxxx\_xxxxxxxxx\_xxxxxxxxx\_xxxxxxxxx

00000100\_10100001\_00000000\_00000010 // bgez \$5, 2

xxxxxxxx\_xxxxxxxxx\_xxxxxxxxx\_xxxxxxxxx

해당 명령어를 통해 \$5에 양수를 넣었으므로 6번으로 점프하는 양상을 기대하고 있습니다.



결과에서 다음 pc주소로 10 > 18로 이동한것을 확인할 수 있으며, 이후 우리가 임의로 넣어둔 명령어를 잘 인식하고 있는 모습을 확인할 수 있습니다.

## JALR

jalr은 입력받은 레지스터로 점프합니다. 이번 과제에서는 \$31에 rs명령어를 저장하는 기능을 구현합니다.

| RegDst | RegDatSel | RegWrite | SEUmod e | ALUsrcB | ALUctrl | ALUop | DataWidt h | MemWrit e | MemtoRe g | Branch | Jump |
|--------|-----------|----------|----------|---------|---------|-------|------------|-----------|-----------|--------|------|
| 10     | 11        | 1        | x        | xx      | xx      | xxxxx | xxx        | 0         | x         | 000    | 10   |

값저장용, 저장11, 저장1, I타입x

,alu사용x,alu사용x,alu사용x,메모리x,메모리x,메모리x브랜치x점프rs사용

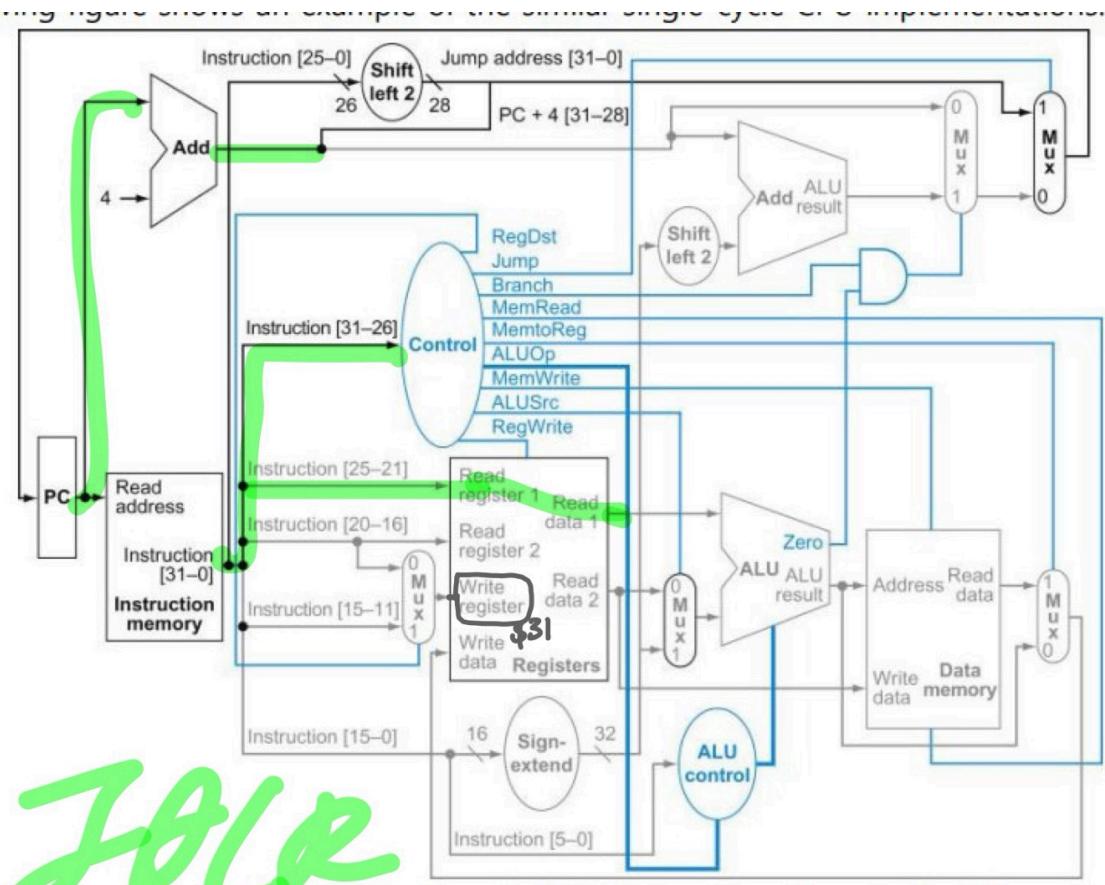


Figure 1 - The single cycle CPU datapath and control path

```

00111100_00000010_00010010_00110100 // lui $2, 0x1234
00110100_01000011_01010110_01111000 // ori $3, $2, 0x5678 ($3 = 0x12345678)
00111100_00000100_00000000_00000000 // lui $4, 0x1122
00110100_10000101_00000000_00001000 // ori $5, $4, 0x3344 ($5 = 0x00000008)

```

00000000\_10100000\_00000000\_00001001 // jalr \$5

해당 자료를 기반으로 5번 레지스터를 8로 맞추고 jalr명령어를 통해서 5번레지스터의 8번 pc값을 로드합니다.



jalr 다음 8번 pc값으로 들어가 반복하는 모습을 확인할 수 있습니다.

```
b0000000 : 00000000_00000000_00000000_00000000 : 00000000
00000001 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000002 : 0010010_00110100_00000000_00000000 : 12340000
00000003 : 0010010_00110100_01010110_01111000 : 12345678
00000004 : 00000000_00000000_00000000_00000000 : 00000000
00000005 : 00000000_00000000_00000000_00001000 : 00000008
00000006 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000007 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000008 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000009 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
0000000a : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
0000000b : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
0000000c : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
0000000d : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
0000000e : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
0000000f : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000010 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000011 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000012 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000013 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000014 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000015 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000016 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000017 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000018 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000019 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
0000001a : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
0000001b : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
0000001c : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
0000001d : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
0000001e : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
0000001f : 00000000_00000000_00000000_00010100 : 00000014
```

레지스터도 1f , 10진수로 31번째 값에 14주소 값이 저장되어 있는 모습을 확인할 수 있습니다.

## 고찰

이번 프로젝트에서는 이미 구현되어있는 싱글 사이클 형태를 기반으로 PRA and 와 or을 각각 구현해보았습니다. 해당 파일을 구현하면서 파일을 참고했을 때, 메뉴얼을 참고하면서 구현하지만 또한 추가적으로 해당 명령어의 개념을 알고 구현해야하기 때문에 추가적인 판단을 해야했습니다.

하지만 PLAor에서 사용한 시그널 중 몇몇은 비트수가 더 많이 설명되어 있어 내부적으로 실험을 통해서 맞춰가는 부분이 있었습니다. 컨트롤 시그널에서는 2비트로 00 01 11 10으로 4개의 값이 나올 수 있지만 구현된 회로에서는 mux는 8개로 맞지 않는 상황이 발생합니다. 이런 상황은 막스의 변수 중 하나를 제한하는 방식으로 가정하고 구현했습니다.

## Reference

2025-1\_SPLab\_proxy\_Assginment1-3