컴퓨터구조 1차 프로젝트

MIPS Single Cycle CPU Implementation

컴퓨터정보공학부

2023202083 최우진

수요일 실습 수강

1. Introduction

이번 1차 프로젝트는 MIPS 기반의 싱글 사이클 CPU를 설계하고 구현하는 것을 목표로 한다. 기본적인 명령어 세트에 더하여, 추가 MIPS 명령어들을 직접 설계하고 구현한다. MIPS 아키텍처에서 데이터패스와 제어패스가 어떻게 상호 작용하여 다양한 명령어를 실제 하드웨어 상에서 실행하는지 이해할 수 있다. 프로젝트의 핵심 구성 요소 중 하나는 PLA이다. PLA는 특정 입력 조건을 감지하고, 이에 따라 설계된 출력 신호를 생성하는 데 사용되며, 마이크로프로세서의 명령어 디코더 및 제어 블록에서 중요한 역할을 한다. PLA를 포함한 CPU의 주요 구성 요소들을 직접 설계하고, 시뮬레이션을 통해 동작을 검증하며, 이를 기반으로 성능을 분석한다.

2. Assignment

2.1 AND

AND는 R-type이다. $d = $s & $t

AND는 R-type 명령어로, 두 레지스터의 값을 비트 단위로 AND 연산한 결과를 지정된 레지스터에 저장하는 명령어이다. 형식은 and $rd, $rs, $rt이며, 연산은 $rd ← $rs & $rt 형태로 수행된다.

PLA_AND

000000_100100_xxxxx

PLA_OR

01_00_1_x_00_0x_00000_xxx_0_0_000_00_xxxxx

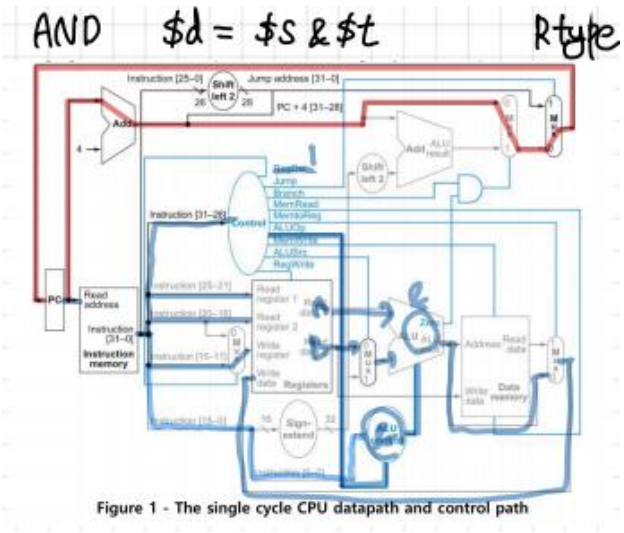


Figure 1 - The single cycle CPU datapath and control path

신호	값	의미
RegDst	01	rd 필드에 저장 (R-type)
RegDatSel	00	ALU 결과 사용
RegWrite	1	레지스터 파일에 기록
EXTMode	x	사용되지 않음
ALUSrcB	00	두 번째 입력은 레지스터
ALUctrl	0x	AND 연산
ALUOp	00000	AND 연산 코드 (R-type용)
ALUSrcA	x	기본값 (레지스터 사용)
MemWrite	0	메모리에 쓰지 않음
MemtoReg	0	메모리에서 읽지 않음
Branch	000	분기 아님
Jump	00	점프 아님
DataWidth	x	사용되지 않음

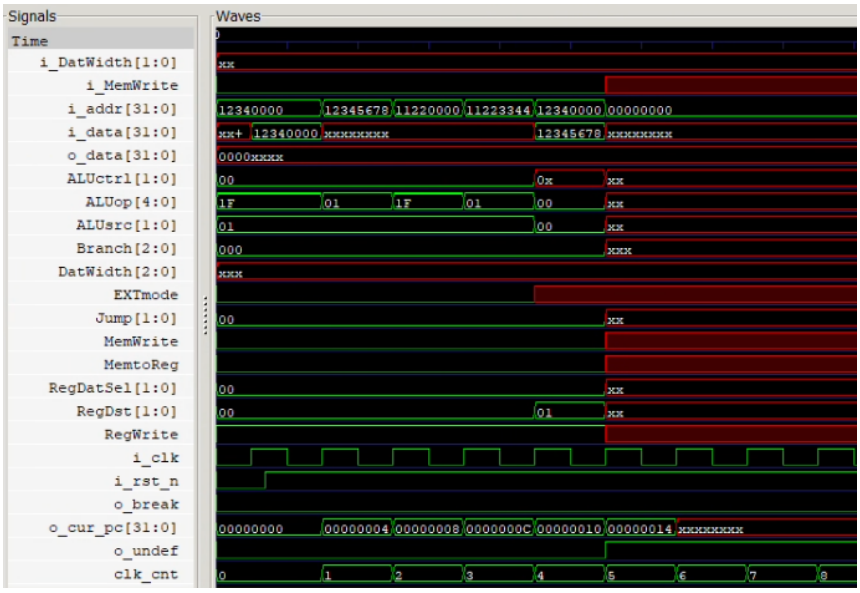
명령어

```
001111_00000_00010_0001_0010_0011_0100    //lui  $2, 0x1234
001101_00010_00011_0101_0110_0111_1000    //ori  $3, $2, 0x5678
001111_00000_00100_0001_0001_0010_0010    //lui  $4, 0x1122
001101_00100_00101_0011_0011_0100_0100    //ori  $5, $4, 0x3344
000000_00010_00011_00100_00000_100100  // and $4, $2, $3
```

$2 = 0x12340000, $3 = 0x12345678으로 설정된 상태에서, and $4, $2, $3 명령어를 실행하였다. and 명령어는 두 레지스터의 값을 비트 단위로 AND 연산하여 결과를 $4에 저장하는 R-type 명령어이다. 명령어의 32비트 이진 표현은 000000_00010_00011_00100_00000_100100으로, 이는 $4 ← $2 & $3 연산을 수행한다. 두 값의 AND 결과는 0x12340000 & 0x12345678 = 0x12340000이 된다.

결과

```
00000002 : 00010010_00110100_00000000_00000000 : 12340000
00000003 : 00010010_00110100_01010110_01111000 : 12345678
00000004 : 00010010_00110100_00000000_00000000 : 12340000
00000005 : 00010001_00100010_00110011_01000100 : 11223344
```



$2 = 0x12340000, $3 = 0x12345678일 때, and $4, $2, $3 명령어는 0x12340000 & 0x12345678 = 0x12340000의 결과를 계산한다. 명령어 실행 후 $4에는 0x12340000이 정확히 저장되었고, ALU의 출력 신호(o_result) 및 레지스터 파일 상태에서도 동일한 결과가 확인되었다.

2.2 NOR

NOR는 R-type이다. $d = ~($s | $t)

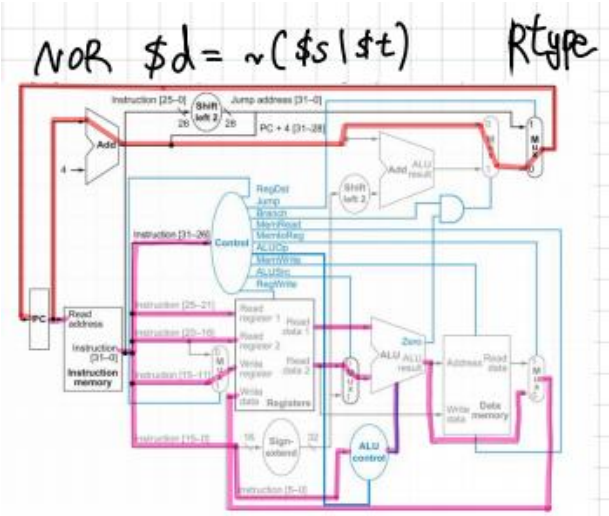
NOR은 R-type 명령어로, 두 레지스터의 값을 비트 OR 연산한 뒤 반전(NOT)하여 결과를 레지스터에 저장하는 연산이다. 명령어 형식은 nor $rd, $rs, $rt이며, 실제 연산은 $rd ← ~($rs | $rt) 형태로 수행된다.

PLA_AND

000000_100111_xxxxx

PLA_OR

01_00_1_x_00_0x_00010_xxx_0_0_000_00_xxxxx



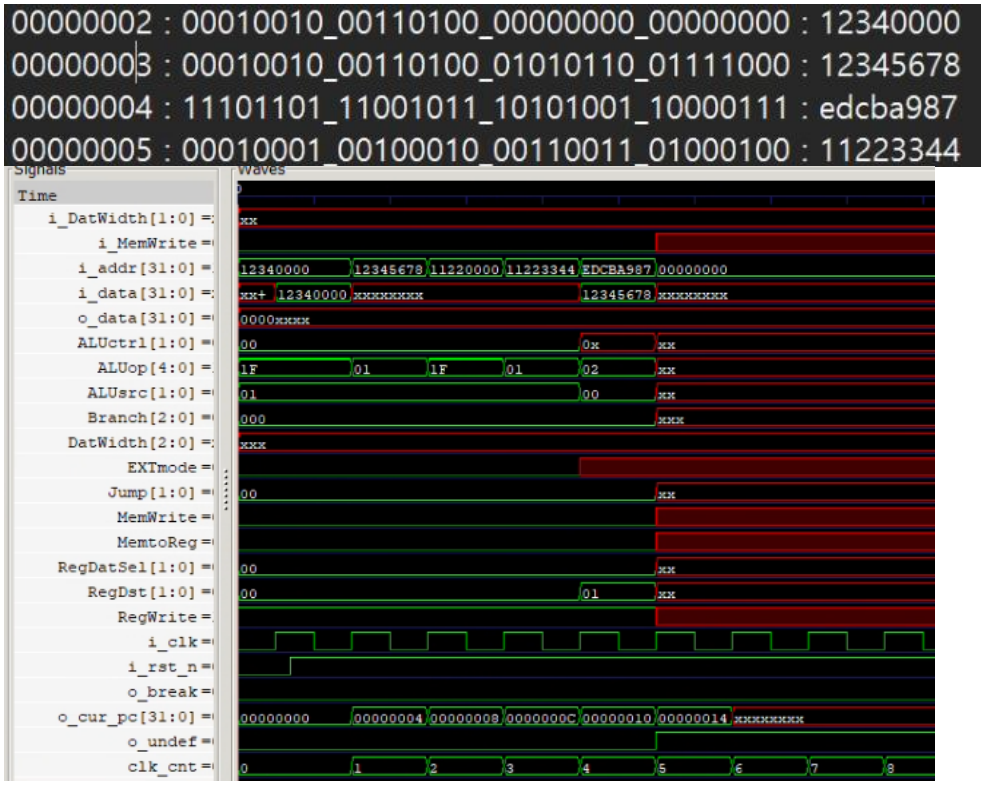
신호	값	의미
RegDst	01	rd 필드에 저장
RegDatSel	00	ALU 결과 사용
RegWrite	1	레지스터 기록
EXTMode	x	사용 안 함
ALUSrcB	00	두 번째 입력은 레지스터
ALUctrl	0x	NOR 연산
ALUOp	00010	NOR 연산 코드
ALUSrcA	x	기본값
MemWrite	0	X
MemtoReg	0	X
Branch	000	X
Jump	00	X
DataWidth	x	무관

명령어

```
001111_00000_00010_0001_0010_0011_0100    //lui  $2, 0x1234
001101_00010_00011_0101_0110_0111_1000    //ori  $3, $2, 0x5678
001111_00000_00100_0001_0001_0010_0010    //lui  $4, 0x1122
001101_00100_00101_0011_0011_0100_0100    //ori  $5, $4, 0x3344
000000_00010_00011_00100_00000_100111  // nor $4, $2, $3
```

NOR 명령어를 테스트하기위해 nor $4, $2, $3 명령어를 실행하였다. $2, $3에 각각 0x12340000, 0x12345678을 저장한 후, nor $4, $2, $3 명령어를 실행하여 두 값을 비트 OR한 결과에 NOT 연산을 수행하였다.

해당 명령어는 32비트 이진 코드로 000000_00010_00011_00100_00000_100111이며, 이는 $2 | $3의 결과를 반전하여 $4에 저장한다.
구체적으로 OR 연산 결과는 0x12345678이고, 이를 NOT 연산하면 0xEDCBA987이 되어야 한다.
결과



nor $4, $2, $3 명령어 실행 후, 레지스터 $4에는 ~(0x12340000 | 0x12345678) = ~0x12345678 = 0xEDCBA987이 정확하게 저장되었다.
시뮬레이션 결과, ALU의 출력값과 레지스터 $4의 값 모두 0xEDCBA987로 나타났으며, 이를 통해 NOR 연산이 비트 단위로 정확히 수행되고 있음을 확인할 수 있었다.

2.3 ADDI

ADDI는 I-type이다. $t = $s + SE(i)

ADDI(Add Immediate)는 I-type 명령어로, 주어진 레지스터 값과 16비트의 immediate을 더한 결과를 목적지 레지스터에 저장한다. 명령어 형식은 addi $rt, $rs, imm이며, 실제 연산은 $rt ← $rs + SignExtend(imm) 형태로 수행된다.

PLA_AND

001000_xxxxxx_xxxxx

PLA_OR

00_00_0_0_01_0x_00100_xxx_0_0_000_00_xxxxx

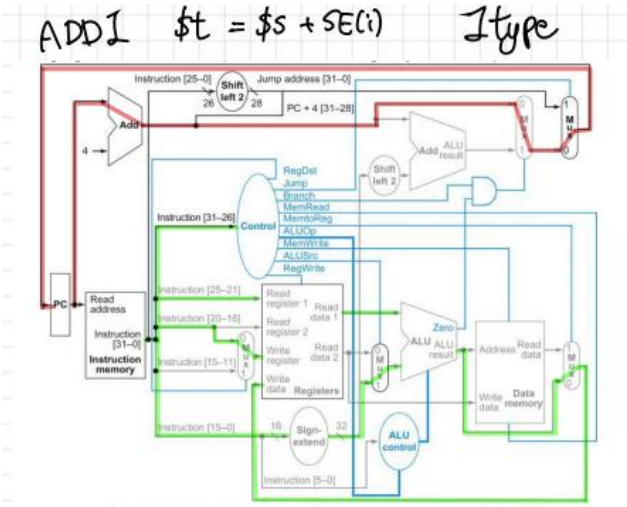


Figure 1 - The single cycle CPU datapath and control path

신호	값	의미
RegDst	00	rt 필드에 저장 (I-type)
RegDatSel	00	ALU 결과
RegWrite	1	레지스터에 쓰기
EXTMode	1	Sign-Extension
ALUSrcB	10	Immediate 값
ALUctrl	0x	ADD 연산
ALUOp	x	직접 쓰지 않음 (opcode로 처리)
ALUSrcA	x	기본값
MemWrite	0	메모리 쓰기 없음
MemtoReg	0	메모리에서 읽지 않음
Branch	000	분기 아님
Jump	00	점프 아님
DataWidth	x	무관

명령어

```
001111_00000_00010_0001_0010_0011_0100    //lui  $2, 0x1234
001101_00010_00011_0101_0110_0111_1000    //ori  $3, $2, 0x5678
001111_00000_00100_0001_0001_0010_0010    //lui  $4, 0x1122
001101_00100_00101_0011_0011_0100_0100    //ori  $5, $4, 0x3344
001000_00000_00011_00000000_00000101  // addi $3, $2, 5
```

ADDI 명령어를 테스트하기 위해 addi $3, $2, 5 명령어를 실행하여 $3 = $2 + 5 결과가 나오는지 확인하였다.

결과

$2 = 0x0000000A인 상태에서 addi $3, $2, 5 명령어를 실행한 결과, $3 = 0x0000000F이 저장되었으며, ALU의 연산 결과 또한 해당 값을 출력하였다. 시뮬레이션 또는 파형 분석을 통해 레지스터 파일의 $3에 정확히 결과값이 저장된 것을 확인하였으며, 이를 통해 ADDI 명령어가 immediate값을 부호 확장하여 덧셈 연산을 정확히 수행함을 확인할 수 있었다.

2.4 SLTU

SLTU는 R-type이다. $d = ($s < $t)

SLTU(Set on Less Than Unsigned)는 R-type 명령어로, 두 레지스터의 값을 부호 없는 정수로 비교하여 $rs < $rt일 경우 결과 레지스터에 1을, 그렇지 않으면 0을 저장하는 명령어이다. 형식은 sltu $rd, $rs, $rt이며, 정수 비교 시 부호를 고려하지 않기 때문에 음수보다는 unsigned 정수 비교가 필요할 때 사용된다.

PLA_AND

000000_101011_xxxxx

PLA_OR

01_00_1_x_00_0x_10000_xxx_0_0_000_00_xxxxx

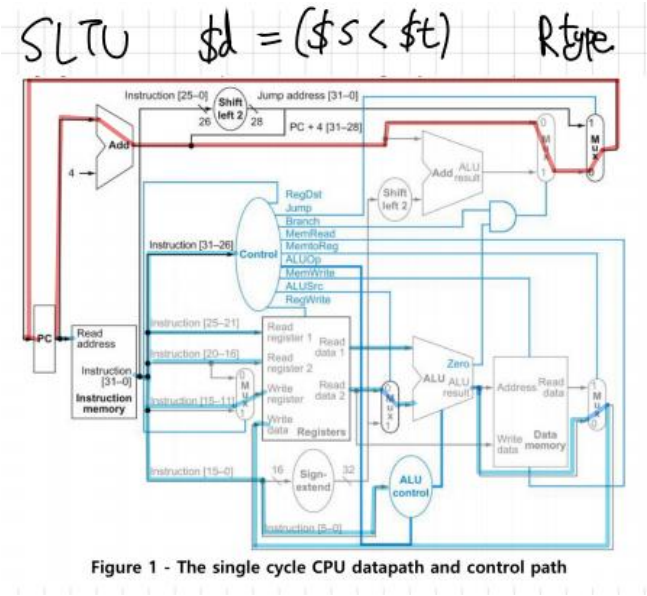


Figure 1 - The single cycle CPU datapath and control path

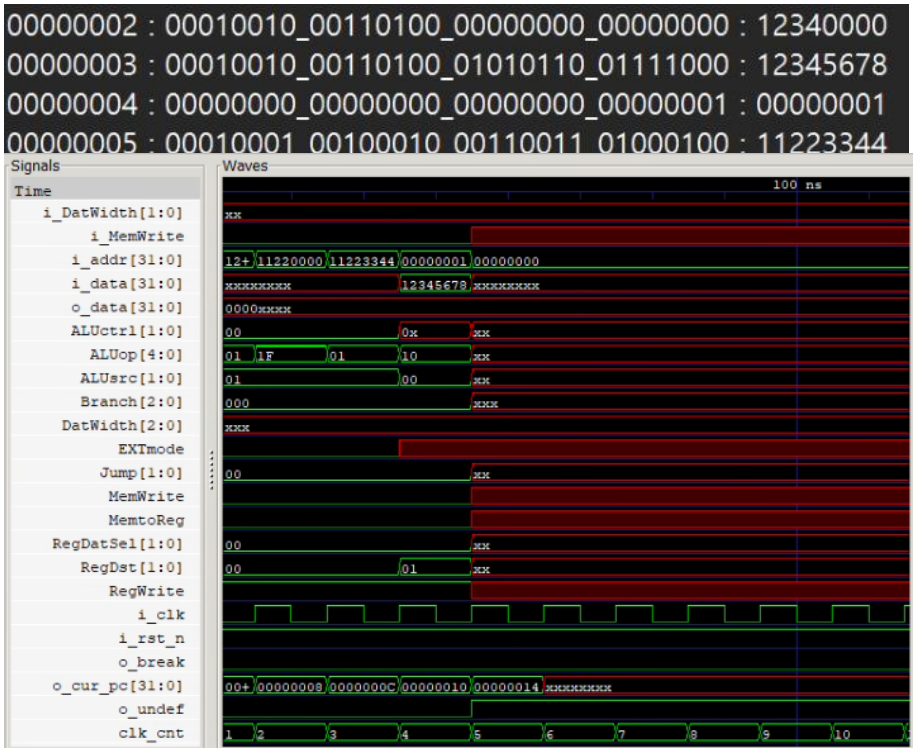
신호	값	의미
RegDst	01	rd 필드에 저장 (R-type)
RegDatSel	00	ALU 결과 사용
RegWrite	1	레지스터에 결과 기록
EXTMode	x	사용 안 함
ALUSrcB	00	두 번째 입력은 레지스터
ALUctrl	0x	SLTU 연산
ALUOp	10000	SLTU에 해당하는 ALU 연산 코드
ALUSrcA	x	기본값
MemWrite	0	메모리에 쓰지 않음
MemtoReg	0	메모리에서 읽지 않음
Branch	000	분기 아님
Jump	00	점프 아님
DataWidth	x	무관

명령어

```
001111_00000_00010_0001_0010_0011_0100    //lui  $2, 0x1234
001101_00010_00011_0101_0110_0111_1000    //ori  $3, $2, 0x5678
001111_00000_00100_0001_0001_0010_0010    //lui  $4, 0x1122
001101_00100_00101_0011_0011_0100_0100    //ori  $5, $4, 0x3344
000000_00010_00011_00100_00000_101011  // sltu $4, $2, $3
```

SLTU 명령어를 테스트하기 위해 sltu $4, $2, $3 명령어를 실행하였다. unsign 비교에서 보다 작기 때문에 결과는 1이 되어야 하며, 해당 결과가 $4에 저장된다.

결과



실행 결과, SLTU 명령어를 실행하면 unsign 비교 결과 $2 < $3이므로, $4 = 0x00000001이 저장된다. 시뮬레이션 결과 ALU의 결과 출력이 0x00000001로 나타났으며, 레지스터 파일에서도 $4에 1이 기록된 것을 통해 SLTU 연산이 정확하게 동작함을 확인할 수 있었다.

2.5 SRL

SRL은 R-type이다. $d = $t >> a

SRL(Shift Right Logical)은 R-type 명령어로, 주어진 레지스터의 값을 오른쪽으로 지정된 비트 수만큼 논리적으로 이동시키는 연산을 수행한다. 명령어 형식은 srl $rd, $rt, shamt이며, $rt의 값을 shamt만큼 오른쪽으로 이동시킨 결과가 $rd에 저장된다. 이때 이동된 자리에는 0이 채워지며, 부호 비트는 고려하지 않는다. 따라서 SRL은 unsigned int 처리를 위한 연산으로 사용된다.

PLA_AND

000000_000010_xxxxx

PLA_OR

01_00_1_x_00_00_01110_xxx_0_0_000_00_xxxxx

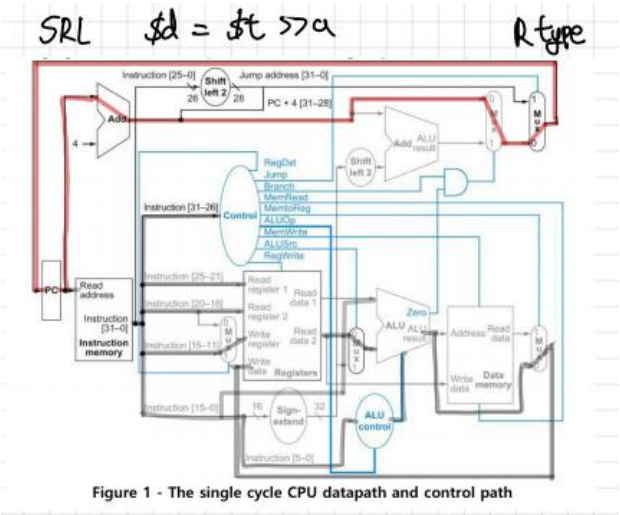


Figure 1 - The single cycle CPU datapath and control path

신호	값	의미
RegDst	01	rd 필드에 저장
RegDatSel	00	ALU 결과
RegWrite	1	결과 저장
EXTMode	x	사용 안 함
ALUSrcB	00	shamt 필드 사용
ALUctrl	00	시프트 연산
ALUOp	01110	SRL 수행 코드
ALUSrcA	x	기본값
MemWrite	0	X
MemtoReg	0	X
Branch	000	X
Jump	00	X
DataWidth	x	무관

명령어

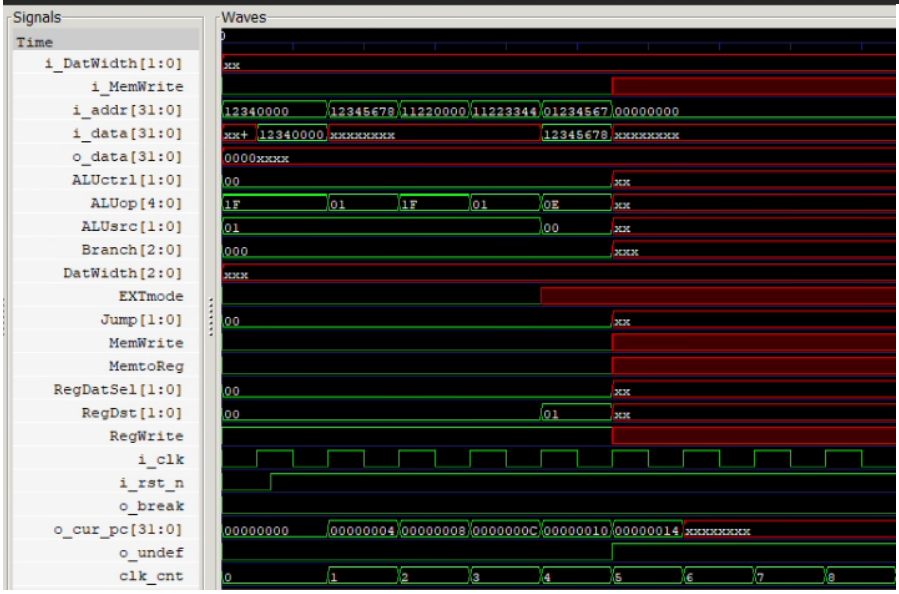
```
001111_00000_00010_0001_0010_0011_0100    //lui  $2, 0x1234
001101_00010_00011_0101_0110_0111_1000    //ori  $3, $2, 0x5678
001111_00000_00100_0001_0001_0010_0010    //lui  $4, 0x1122
001101_00100_00101_0011_0011_0100_0100    //ori  $5, $4, 0x3344
000000_00000_00011_00100_00100_000010  // srl $4, $3, 4
```

SRL 명령어를 테스트하기 위해 명령어를 통해 오른쪽으로 4비트 이동시키는

연산을 수행하였다.

결과

```
00000000 : 00000000_00000000_00000000_00000000 : 00000000
00000001 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000002 : 00010010_00110100_00000000_00000000 : 12340000
00000003 : 00010010_00110100_01010110_01111000 : 12345678
00000004 : 00000001_00100011_01000101_01100111 : 01234567
00000005 : 00010001_00100010_00110011_01000100 : 11223344
00000006 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
```



실행 결과 $3 = 0x12345678의 값을 오른쪽으로 4비트 이동시켜 $4 = 0x01234567이 저장되었으며, 논리적 시프트 연산이 정확하게 수행됨을 확인할 수 있었다. wave 상에서도 ALU 연산 결과 출력 신호(o_result)에 0x01234567이 나타났으며, 이를 통해 SRL 명령어의 기능과 datapath 내 시프트 연산 유닛이 정상적으로 동작함을 확인할 수 있었다.

2.6 SH

SH는 I-type이다. MEM[$s + i]:1 = LH($t)

SH(Store Halfword) 명령어는 I-type 형식으로, 32비트 레지스터의 하위 16비트를 메모리에 저장하는 명령어이다. 명령어 형식은 SH $rt, offset($rs)로, 이는 $rt의 하위 16비트를 $rs + offset 주소의 메모리에 저장한다는 의미이다. 메모리는 바이트 단위로 접근되며, SH 명령어는 해당 주소로부터 2바이트를 차지하게 된다. 테스트벤치에서는 $5 레지스터에 0x11223344가 저장되어 있는 상태에서, SH $5, 9($0) 명령어를 실행하여 9번지 주소에 해당 값의 하위

16비트인 0x3344를 저장하도록 하였다. SH 명령어는 내부적으로 하위 바이트부터 상위 바이트 순으로 메모리에 기록하기 때문에, 결과적으로 메모리 주소 9번지에는 0x44, 10번지에는 0x33이 저장되게 된다.

PLA_AND

101001_xxxxxx_xxxxx

PLA_OR

xx_xx_0_1_01_00_00100_010_1_x_000_00_xxxxx

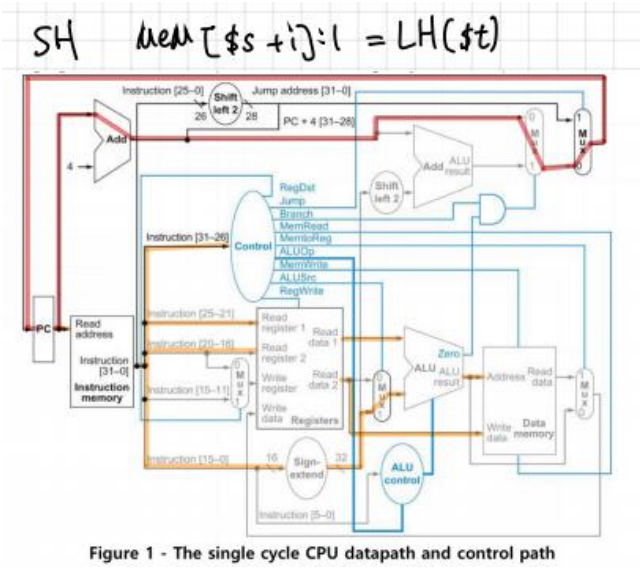


Figure 1 - The single cycle CPU datapath and control path

신호	값	의미
RegDst	00	의미 없음 (레지스터에 저장 안 함)
RegDatSel	xx	사용 안 함
RegWrite	0	레지스터 기록 없음
EXTMode	1	Sign-Extension
ALUSrcB	10	Immediate 사용
ALUctrl	0x	주소 연산용 ADD
ALUOp	x	직접 사용 안 함
ALUSrcA	x	기본값
MemWrite	1	메모리에 기록
MemtoReg	0	사용 안 함
Branch	000	아님
Jump	00	아님

DataWidth 10 Halfword (2 bytes) 저장

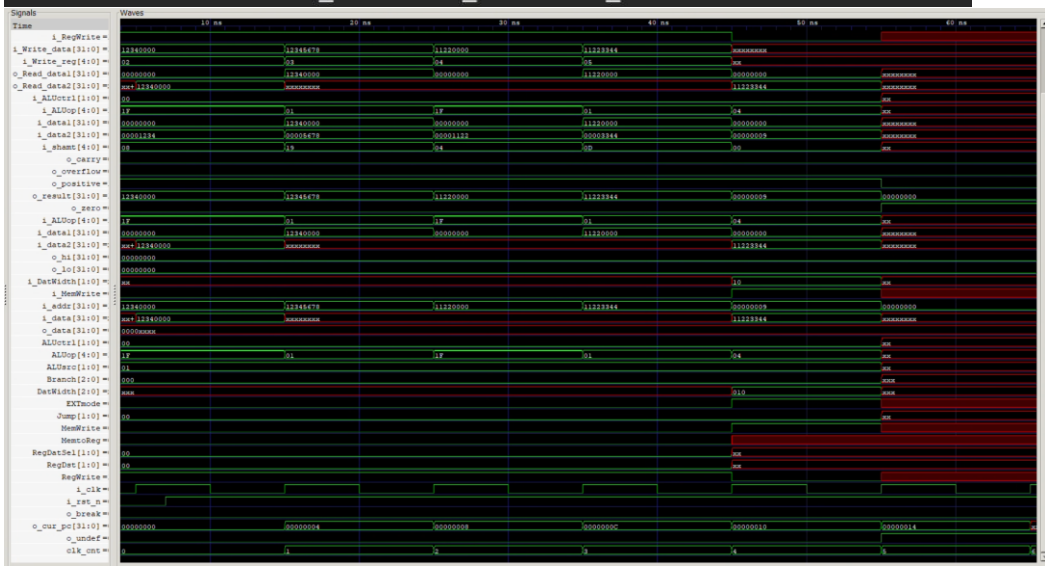
명령어

```
001111_00000_00010_0001_0010_0011_0100
001101_00010_00011_0101_0110_0111_1000
001111_00000_00100_0001_0001_0010_0010
001101_00100_00101_0011_0011_0100_0100
101001_00000_00101_00000000_00001001  // sh
```

$5 레지스터의 하위 16비트 값을 메모리 주소 9($0)에 저장하는 동작을 수행한다. 즉, 레지스터 $5에 저장된 32비트 값 중 하위 16비트(0x3344)만을 메모리의 9번지부터 2바이트에 걸쳐 저장하게 된다.

결과

```
00000800 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000801 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000802 : xxxxxxxx_xxxxxxxx_00110011_01000100 : xxxx3344
00000803 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
```



이 값의 하위 16비트는 0x3344이므로, SH 명령어 실행 후 메모리에는 다음과 같은 결과가 나타난다 이후, 동일한 주소에서 LH 명령어를 통해 해당 값을 읽어온 결과, 예상대로 0x3344가 레지스터에 정확히 저장됨을 확인하였다. SH 명령어가 레지스터의 하위 16비트를 메모리에 정상적으로 저장하는 동작을 구현한 datapath 및 control signal이 올바르게 작동함을 확인할 수 있었다.

2.7 LB

LB는 I-type이다. $t = SE(MEM[$s + i]:1)

LB(Load Byte) 명령어는 I-type 형식으로, 메모리에서 1바이트(8비트) 데이터를 읽어온 후 이를 Sign-Extension하여 32비트로 변환하고 레지스터에 저장하는 명령어이다. 명령어 형식은 LB $rt, offset($rs)로, 이는 $rs + offset 주소로부터 1바이트 데이터를 읽어와 32비트로 확장한 뒤 $rt에 저장하는 동작을 수행한다.

PLA_AND

100000_xxxxxx_xxxxx

PLA_OR

00_00_1_1_01_00_00100_010_0_1_000_00_xxxxx

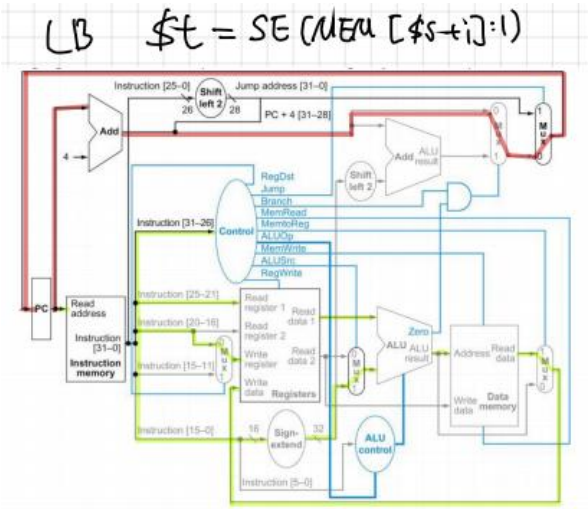


Figure 1 - The single cycle CPU datapath and control path

신호	값	의미
RegDst	00	rt 필드에 저장
RegDatSel	01	Memory에서 읽은 값 사용
RegWrite	1	결과 저장
EXTMode	1	Sign-Extension
ALUSrcB	10	Immediate
ALUctrl	0x	주소 계산용 ADD
ALUOp	x	사용 안 함
ALUSrcA	x	기본값

MemWrite	0	쓰지 않음
MemtoReg	1	메모리 값 사용
Branch	000	아님
Jump	00	아님
DataWidth	00	Byte (1 byte) 읽기

명령어

```
001111_00000_00010_0001_0010_0011_0100    //lui  $2, 0x1234
001101_00010_00011_0101_0110_0111_1000    //ori  $3, $2, 0x5678
001111_00000_00100_0001_0001_0010_0010    //lui  $4, 0x1122
001101_00100_00101_0011_0011_0100_0100    //ori  $5, $4, 0x3344
101001_00000_00101_00000000_00001001  // sh
100000_00000_00110_00000000_00001001  // lb
```

LB 명령어의 동작을 확인하기 위해 먼저 sh $5, 9($0) 명령어를 실행하여 $5의 하위 16비트 값인 0x3344를 메모리 9번지부터 저장했다. 이때 메모리 9번지에는 0x44, 10번지에는 0x33이 저장된다. 이후 lb $6, 9($0) 명령어를 통해 메모리 9번지의 값을 읽고 부호 확장을 수행하여 $6 = 0x00000044를 얻었고, lb $7, 10($0) 명령어로 메모리 10번지의 값을 읽은 결과 $7 = 0x00000033

결과

```
00000000 : 00000000_00000000_00000000_00000000 : 00000000
00000001 : xxxxxxxx_xxxxxxxx_xxxxxxxx_xxxxxxxx : xxxxxxxx
00000002 : 00010010_00110100_00000000_00000000 : 12340000
00000003 : 00010010_00110100_01010110_01111000 : 12345678
00000004 : 00010001_00100010_00000000_00000000 : 11220000
00000005 : 00010001_00100010_00110011_01000100 : 11223344
00000006 : 00000000_00000000_00110011_01000100 : 00003344
```

실행 결과, 레지스터 $6과 $7에는 각각 0x00000044, 0x00000033이 저장되었다. 예상한 바와 동일하게 메모리에서 1바이트를 읽고 이를 정확하게 sign-extend한 값이다. LB 명령어가 메모리로부터 바이트 단위 데이터를 정확히 읽고, sign-extend을 통해 정수로 변환하여 저장하는 과정을 정확히 수행함을 확인할 수 있었다.

2.8 BNE

BNE는 I-type이다. if ($s ≠ $t) pc += SE(i) << 2

BNE(Branch on Not Equal) 명령어는 I-type 명령어로, 두 레지스터의 값이 서로 다를 경우 프로그램 실행 흐름을 지정된 offset만큼 분기시키는 조건 분기 명령어이다. 명령어 형식은 bne $rs, $rt, offset이며, $rs != $rt일 경우 PC ← PC + 4 + (SE(offset) << 2) 방식으로 분기된다. BNE는 비교 연산을 통해 특정 조건에서만 실행 경로를 변경하고자 할 때 활용되며, 조건이 거짓일 경우 다음 명령어로 순차 실행이 이어진다.

PLA_AND

000101_xxxxxx_xxxxx

PLA_OR

xx_xx_0_1_00_0x_00110_xxx_0_x_101_00_xxxxx

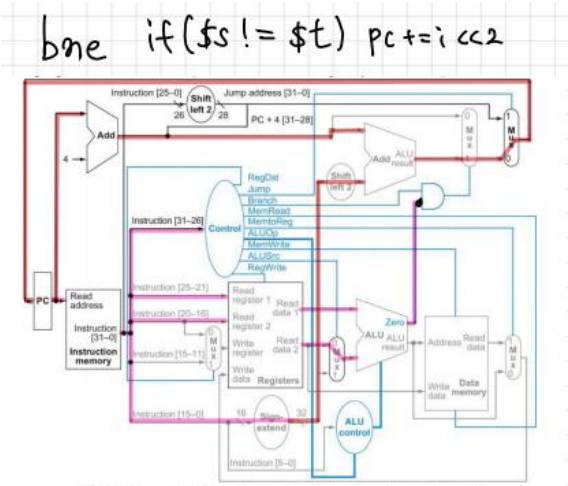


Figure 1 - The single cycle CPU datapath and control path

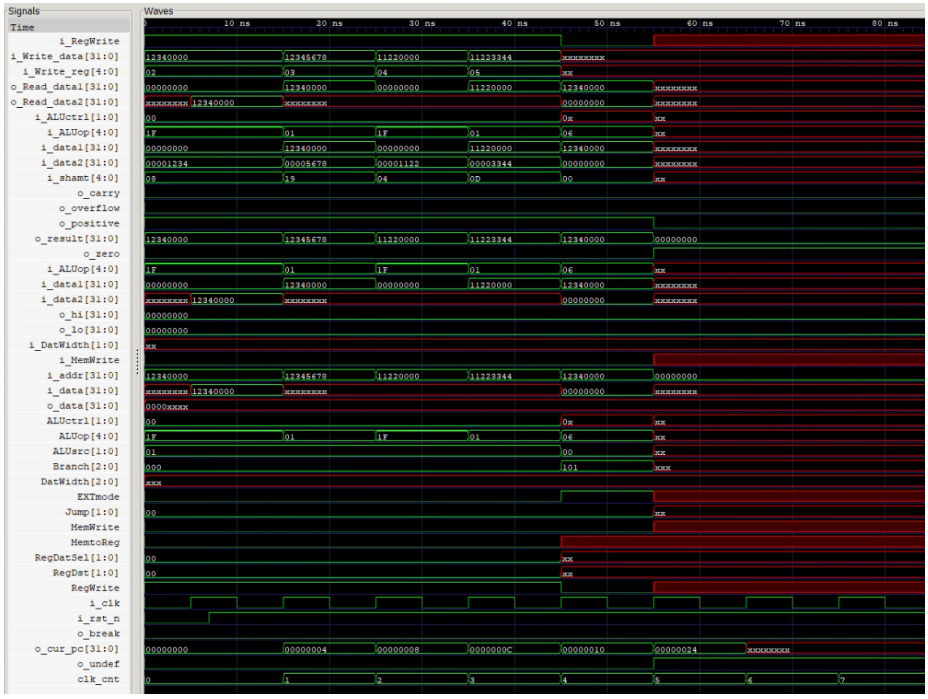
신호	값	의미
RegDst	xx	사용 안 함
RegDatSel	xx	사용 안 함
RegWrite	0	결과 저장 없음
EXTMode	1	Sign-Extension
ALUSrcB	10	Immediate 비교
ALUctrl	0x	비교용 SUB
ALUOp	x	비교 연산 직접 처리
ALUSrcA	x	기본값
MemWrite	0	메모리에 쓰지 않음
MemtoReg	0	X
Branch	010	BNE 조건 분기
Jump	00	아님

명령어

```
001111_00000_00010_0001_0010_0011_0100    //lui  $2, 0x1234
001101_00010_00011_0101_0110_0111_1000    //ori  $3, $2, 0x5678
001111_00000_00100_0001_0001_0010_0010    //lui  $4, 0x1122
001101_00100_00101_0011_0011_0100_0100    //ori  $5, $4, 0x3344
000101_00010_00000_00000000_00000100  // bne $2, $3, offset = 1
```

BNE 명령어 테스트를 위해 $2 = 0x12345678, $3 = 0x9ABCDEF0처럼 서로 다른 값을 저장한 후, bne $2, $3, offset=1 명령어를 실행하였다.
이 명령어는 두 레지스터의 값이 다르기 때문에, PC는 현재 위치에서 4 바이트 다음 명령어를 지나 8바이트 분기된 위치로 점프한다.

결과



실행 결과, $2와 $3 레지스터의 값이 다르기 때문에 조건이 참으로 평가되어, 프로그램 흐름은 정상적으로 8바이트+4 앞으로 분기되었다. 반대로 두 값이 같았다면 분기가 발생하지 않고 다음 명령어로 넘어갔을 것이다.

2.9 BGEZ

BGEZ는 I-type이다. if ($s ≥ 0) pc += SE(i) << 2

BGEZ(Branch on Greater Than or Equal to Zero) 명령어는 I-type 명령어로, 주어진 레지스터 값이 0 이상일 경우 프로그램 실행 흐름을 분기시키는 조건 분기 명령어이다.
형식은 bgez $rs, offset이며, $rs 값이 0 이상이면 PC ← PC + 4 + (SE(offset) << 2)로 분기하고, 그렇지 않으면 다음 명령어로 순차 실행이 계속된다.
이 명령어는 조건 분기를 통해 프로그램 흐름을 제어하는 데 활용되며, 비교 결과에 따라 분기 여부를 결정하는 로직을 datapath에서 수행한다.

PLA_AND

000000_xxxxxx_00001

PLA_OR

xx_xx_0_1_10_1x_10000_xxx_0_0_111_00_xxxxx

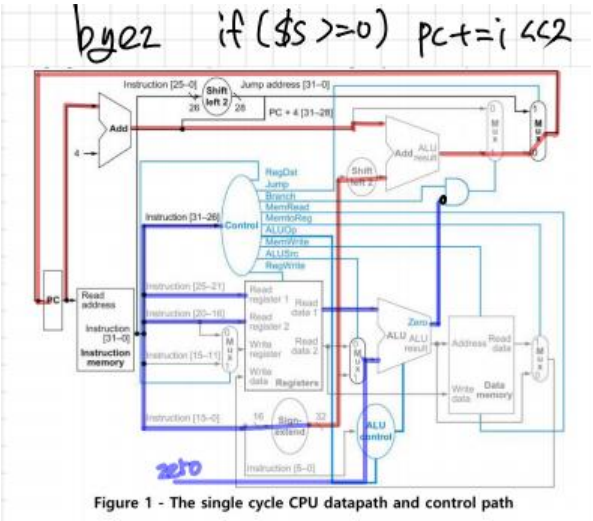


Figure 1 - The single cycle CPU datapath and control path

신호	값	의미
RegDst	xx	사용 안 함
RegDatSel	xx	사용 안 함
RegWrite	0	결과 저장 없음
EXTMode	1	Sign-Extension
ALUSrcB	10	Immediate
ALUctrl	0x	비교용 SUB
ALUOp	x	직접 사용 안 함
ALUSrcA	x	기본값
MemWrite	0	메모리 접근 없음
MemtoReg	0	없음
Branch	011	BGEZ 조건 분기
Jump	00	점프 아님
DataWidth	x	무관

명령어

```
001111_00000_00010_0001_0010_0011_0100    //lui  $2, 0x1234
001101_00010_00011_0101_0110_0111_1000    //ori  $3, $2, 0x5678
001111_00000_00100_0001_0001_0010_0010    //lui  $4, 0x1122
001101_00100_00101_0011_0011_0100_0100    //ori  $5, $4, 0x3344
000001_00011_00001_00000000_00000001  // bgez $3, offset = 1
```

이번 실험에서는 $3 레지스터에 양수 값을 저장한 후, bgez $3, label 명령어를 실행하여 분기 여부를 확인하였다. BGEZ는 특수한 형식으로, opcode는 000001, 그리고 rt 필드가 00001로 고정된다.

결과

$3 레지스터에 양수 값인 0x00000001을 저장한 후 bgez $3, offset = 1 명령어를 실행한 결과, 조건이 참이므로 프로그램은 현재 PC 기준으로 8바이트 떨어진 명령어 위치로 정확히 분기하였다.

분기 결과는 시뮬레이터 또는 Verilog 파형을 통해 확인하였으며, $3 값이 0 이상일 경우에만 분기하는 조건 분기 로직이 datapath 상에서 정확히 구현되었음을 검증할 수 있었다.

이를 통해 BGEZ 명령어의 비교 및 분기 동작이 정확하게 수행되는 것을 확인하였다.

2.10 JALR

JALR는 R-type이다. $31 = PC; PC = $s

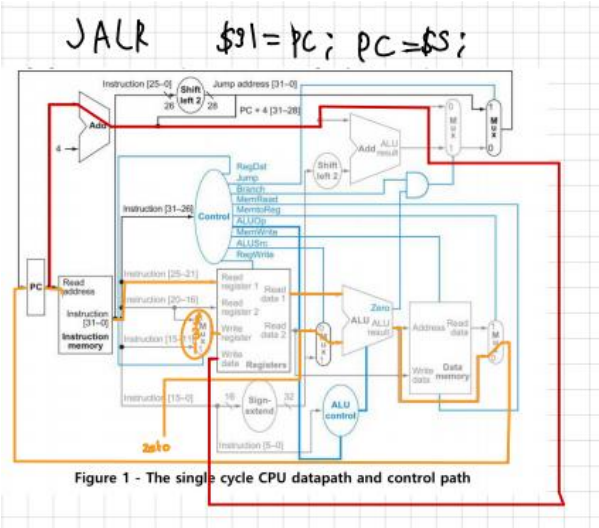
JALR(Jump And Link Register) 명령어는 R-type 명령어로, 레지스터에 저장된 주소로 점프하고 동시에 현재 PC 값을 다른 레지스터에 저장하는 기능을 수행한다. 형식은 jalr $rd, $rs이며, 명령어가 실행되면 PC는 $rs의 값으로 변경되고, 이전 PC 값은 $rd에 저장된다. 이 명령어는 함수 호출 및 복귀 구현에 주로 사용되며, 하드웨어적으로는 점프 주소를 ALU를 통해 계산하고, PC와 $rd의 기록이 동시에 수행되는 복합 동작이다.

PLA_AND

000000_001001_xxxxx

PLA_OR

10_11_1_x_10_0x_00001_xxx_0_x_xxx_10_xxxxx



신호	값	의미
RegDst	10	$31에 저장
RegDatSel	11	현재 PC 값
RegWrite	1	레지스터 저장
EXTMode	x	사용 안 함
ALUSrcB	10	무관 (주소 X)
ALUctrl	0x	무관
ALUOp	00001	Jump에 해당
ALUSrcA	10	레지스터($s) 사용
MemWrite	x	사용 안 함
MemtoReg	x	사용 안 함
Branch	xxx	없음
Jump	10	Register Jump (JALR)
DataWidth	x	무관

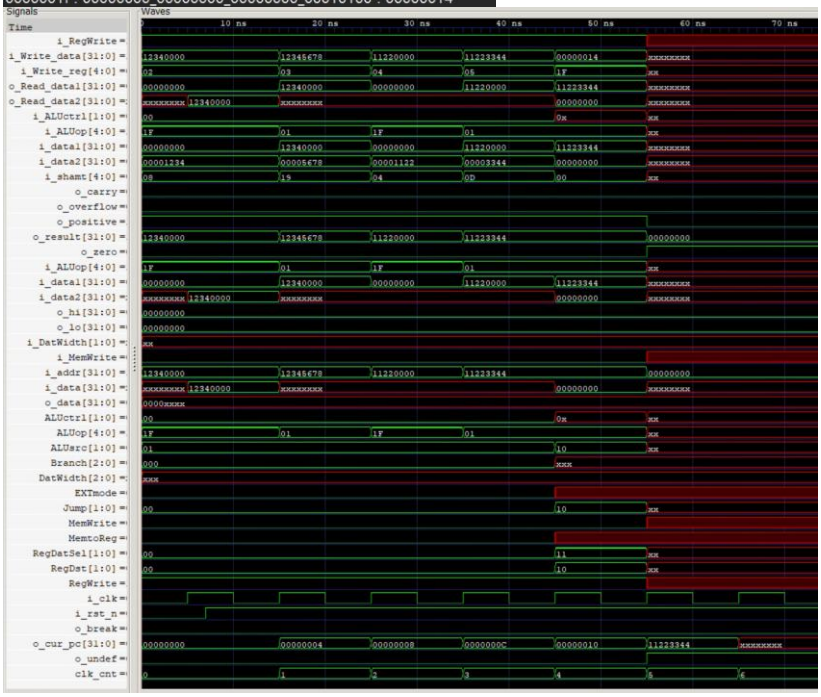
명령어

```
001111_00000_00010_0001_0010_0011_0100    //lui  $2, 0x1234
001101_00010_00011_0101_0110_0111_1000    //ori  $3, $2, 0x5678
001111_00000_00100_0001_0001_0010_0010    //lui  $4, 0x1122
001101_00100_00101_0011_0011_0100_0100    //ori  $5, $4, 0x3344
000000_00101_00000_11111_00000_001001     // jalr $31, $5
```

$5에 점프할 주소를 저장한 후, jalr $31, $5 명령어를 실행하였다. 해당 명령어는 점프 목적지인 $5의 값을 PC로 설정하고, 점프 직전의 PC 값을 $31에 저장한다.

결과

```
00000003 : 00010010_00110100_01010110_01111000 : 12345678
00000004 : 00010001_00100010_00000000_00000000 : 11220000
00000005 : 00010001_00100010_00110011_01000100 : 11223344
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
```



$5에 미리 특정 주소 값을 저장한 뒤 jalr $31, $5 명령어를 실행한 결과, PC는 $5의 값으로 정확히 이동하였고, 이전 PC 값은 $31에 정상적으로 저장되었다. 실행 흐름은 $5가 가리키는 위치로 전환되었으며, $31을 통해 원래 위치로 되돌아올 수 있는 기반도 함께 마련되었음을 확인하였다. 이를 통해 JALR 명령어가 점프와 링크 기능을 정확하게 수행함을 검증할 수 있었다.