컴퓨터 구조 프로젝트2 보고서

학번 : 2018202046

이름 : 이준휘

교수님 : 이성원

분반 : 월3, 수4

1. Introduction

해당 과제에서는 Multicycle로 작동하는 processor에서 특정 명령어들을 구현하는 것이다. 명령어의 경우 이전 과제와 동일한 명령어이며 해당 동작 중 IF(Instruction Fetch)와 Decode 단계는 이미 구현되어 있다. ROM\_DISP를 통해 작성된 명령어가 ROM\_MICRO 파일에서 어디에 위치하는지 알 수 있으며 ROM\_MICRO 파일에는 해당 명령에서의 컨트롤을 조작할 수 있다.

1. Result
2. Field 분석

해당 과제에서 주어진 signal을 그룹화하여 아래와 같이 정리하였다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Mem | IR | RF | ALU | PC | uPC |
| lorD, MemRead, MemWrite, Datwidth | IRwrite | RegDst, RegDatSel, RegWrite | EXTmode, ALUsrcA, ALUsrcB, ALUop, ALUctrl | Branch, PCsrc, PCwrite | StateSel |

Mem에서는 Memory에 접근하는 영역에서의 동작에 관할한다. lorD를 통해 Instruction 또는 Data 주소로부터 값을 읽어와 MemRead와 MemWrite를 통해 읽거나 쓸 상태를 정한 뒤, Datwidth를 통해 내보내는 데이터의 크기를 정한다.

IR은 Instruction Register가 값을 언제 쓸지에 대하여 정의한다. 해당 값은 IF 단계에서만 켜지고, 이미 IF 단계는 구현되어 있기 때문에 해당 과제에서 작성한 명령어에서는 0으로 설정하였다. 이로 인해 후에 설명에서는 해당 부분을 제외하고 설명할 예정이다.

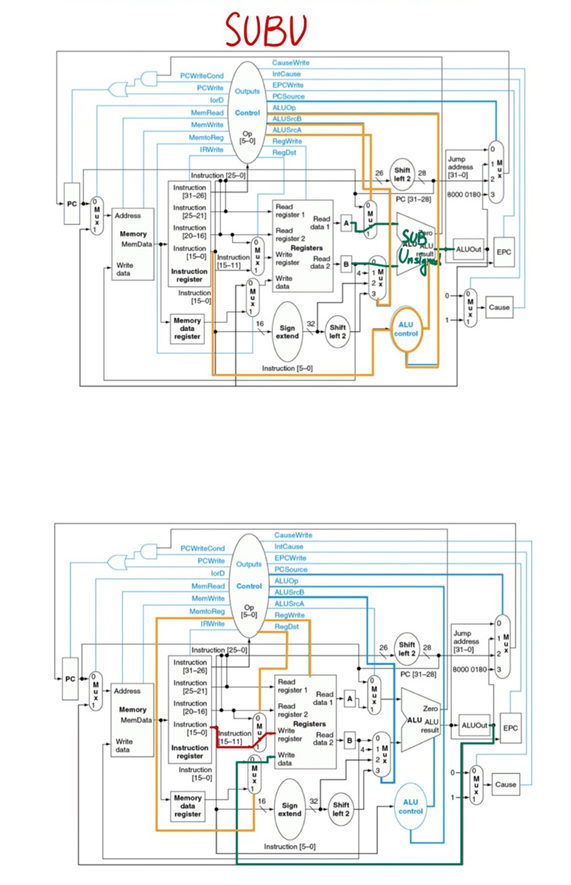
RF는 Register File에 접근하는 명령과 관련된 명령어들의 Field이다. RegDst을 통해 Write을 할 장소를 정의하며 RegDatSel를 통해 Write을 할 Data를 선택한다. RegWrite를 통해 쓸지를 결정하게 된다.

ALU는 연산과 관련된 signal들을 묶어놓았다. EXTmode를 통해 immediate value의 extension을 결정하며 ALUsrcA, ALUsrcB를 통해 연산할 값을 정의한다. ALUop은 어떤 연산을 진행할지를 나타내며, ALUctrl에서는 연산의 순서와 shift를 정의한다.

PC에서는 Program Counter와 관련된 신호들이 있다. Branch를 통해 Branch나 jump 시의 조건을 걸 수 있으며 PCsrc를 통해 변경할 값을 정의할 수 있다. PCwrite 신호는 PC값에 실제적으로 쓸지를 결정한다.

uPC는 micro-program counter와 관련된 신호다. 해당 값을 조절하여 명령어에 대한 State의 수를 조절할 수 있다.

1. SUBU



해당 명령어는 unsigned sub 연산을 진행하는 명령어로, IF와 Decode 이외에 2 cycle이 추가로 동작한다(R-Type). 첫 cycle의 동작에서는 A(rs)와 B(rt)를 연산할 것을 선택하고, 연산의 경우 unsigned SUB을 하도록 한다. 연산한 결과는 다음 cycle에서는 ALUOut에 저장되어 있기 때문에 이를 RF에 쓸 데이터로 설정하고 쓸 위치는 rd로 정한 후 쓰는 동작을 수행한다. 해당 동작을 signal로 정리하면 다음과 같다.

SUBU 3rd cycle

|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | xx | Register File에 쓰이면 안 되기 때문에 RegWrite만 0으로 두고 나머지 신호는 사용하지 않는다. |
| RegDatSel | xxx |
| RegWrite | 0 |
| ALU | EXTmode | x | Immediate value를 사용하지 않는다. |
| ALUsrcA | 000 | 000(rs)를 사용한다. |
| ALUsrcB | 000 | 000(rt)를 사용한다. |
| ALUop | 00111 | unsigned a-b를 사용한다. |
| ALUctrl | 0x | a와 b의 순서를 바꾸지 않고 shift 또한 진행하지 않는다. |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 11 | 다음 cycle이 존재하기에 upc++를 진행한다. |

SUBU 4th cycle

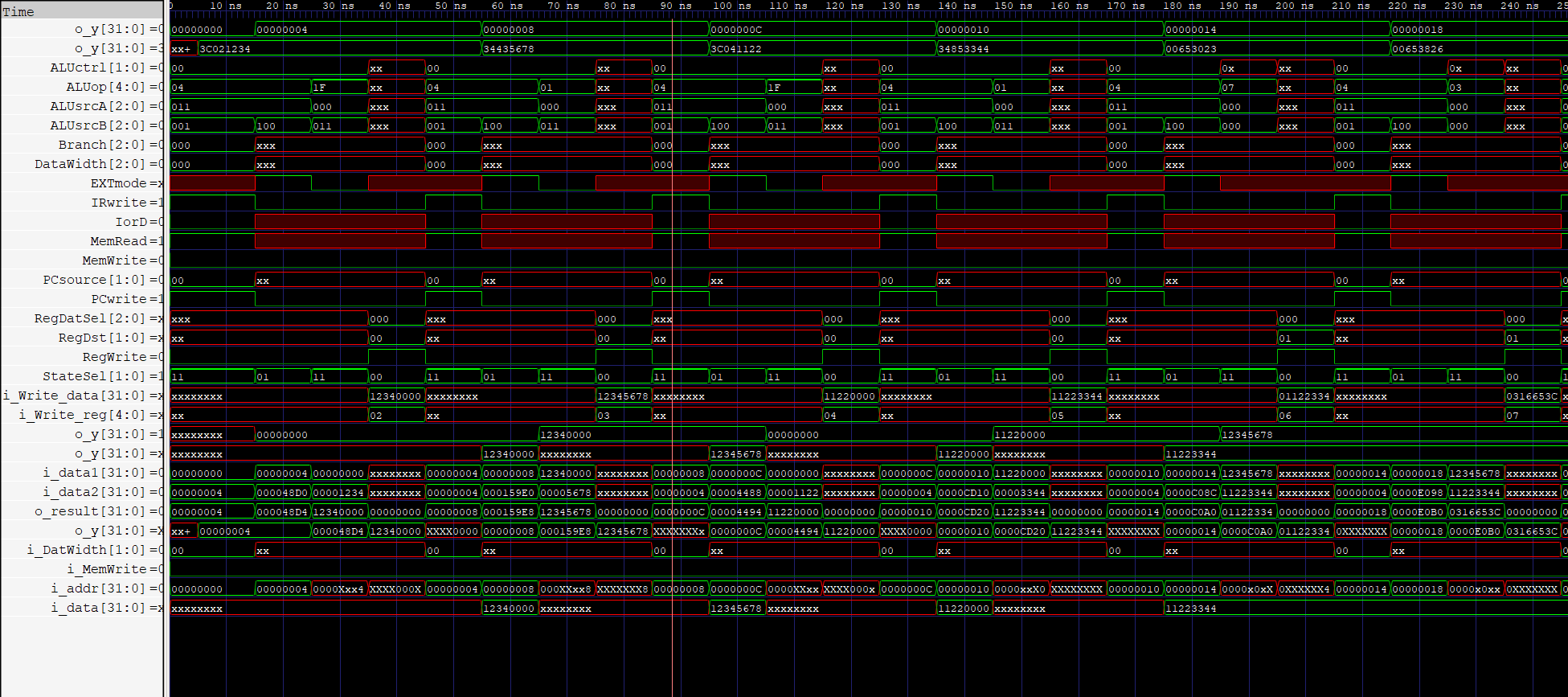
|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | 01 | address는 rd의 주소로 한다. |
| RegDatSel | 000 | 쓸 Data는 ALUOut으로부터 가져온다. |
| RegWrite | 1 | RF에서 쓰기 동작을 수행한다. |
| ALU | EXTmode | x | ALU는 아무 동작을 하지 않는다. |
| ALUsrcA | xxx |
| ALUsrcB | xxx |
| ALUop | xxxxx |
| ALUctrl | xx |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 00 | 다음 cycle이 존재하지 않기에 upc = 0를 진행한다. |

해당 명령이 정상적으로 동작하는지 검증을 하기 위해 다음과 같은 text를 작성하였다.

텍스트이(가) 표시된 사진

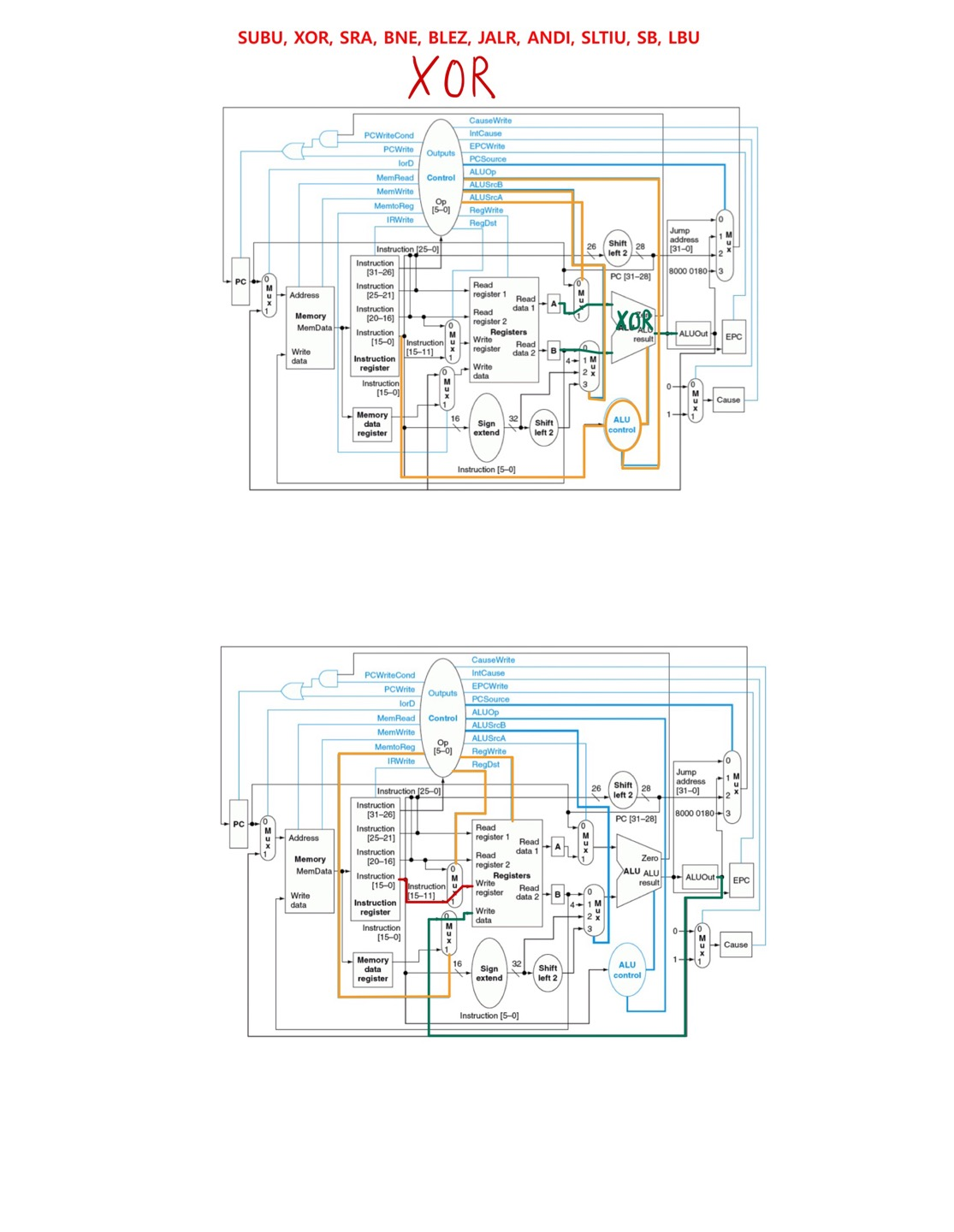
자동 생성된 설명

해당 명령은 이전 과제와 동일한 testbench를 통해 진행되었다. 우선 lui와 ori를 통해 $3과 $5에 각각 0x12345678과 0x11223344를 넣어두고 해당 값을 SUBU 연산하여 결과를 $6에 저장하는 코드다. 이에 대한 결과는 다음과 같았다.



결과 사진에서 주황 박스 영역이 SUBU 연산이다. IF와 Decode된 상태 이후를 확인해보면 3번째 cycle에서는 ALU의 인자로 A와 B가 들어간 것을 확인할 수 있고 연산 결과로 0x12345678 – 0x11223344 = 0x01122334로 제대로 연산이 진행된 것을 확인할 수 있다. 4번째 cycle에서는 ALUOut에 저장되어 있는 연산 결과가 RegWrite 신호에 따라 0x06번지에 저장되는 것을 확인할 수 있다. 이를 통해 해당 명령어가 정상적으로 구현되었음을 확인하였다.

1. XOR



해당 명령어는 XOR 연산을 진행하는 명령어로 IF와 Decode 이외에 2 cycle이 추가로 동작한다(R-Type). 첫 cycle의 동작에서는 A(rs)와 B(rt)를 연산할 것을 선택하고, 연산의 경우 XOR을 하도록 한다. 연산한 결과는 다음 cycle에서는 ALUOut에 저장되어 있기 때문에 이를 RF에 쓸 데이터로 설정하고 쓸 위치는 rd로 정한 후 쓰는 동작을 수행한다. 해당 동작을 signal로 정리하면 다음과 같다.

XOR 3rd cycle

|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | xx | Register File에 쓰이면 안 되기 때문에 RegWrite만 0으로 두고 나머지 신호는 사용하지 않는다. |
| RegDatSel | xxx |
| RegWrite | 0 |
| ALU | EXTmode | x | Immediate value를 사용하지 않는다. |
| ALUsrcA | 000 | 000(rs)를 사용한다. |
| ALUsrcB | 000 | 000(rt)를 사용한다. |
| ALUop | 00011 | Bitwise XOR를 사용한다. |
| ALUctrl | 0x | a와 b의 순서를 바꾸지 않고 shift 또한 진행하지 않는다. |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 11 | 다음 cycle이 존재하기에 upc++를 진행한다. |

XOR 4th cycle

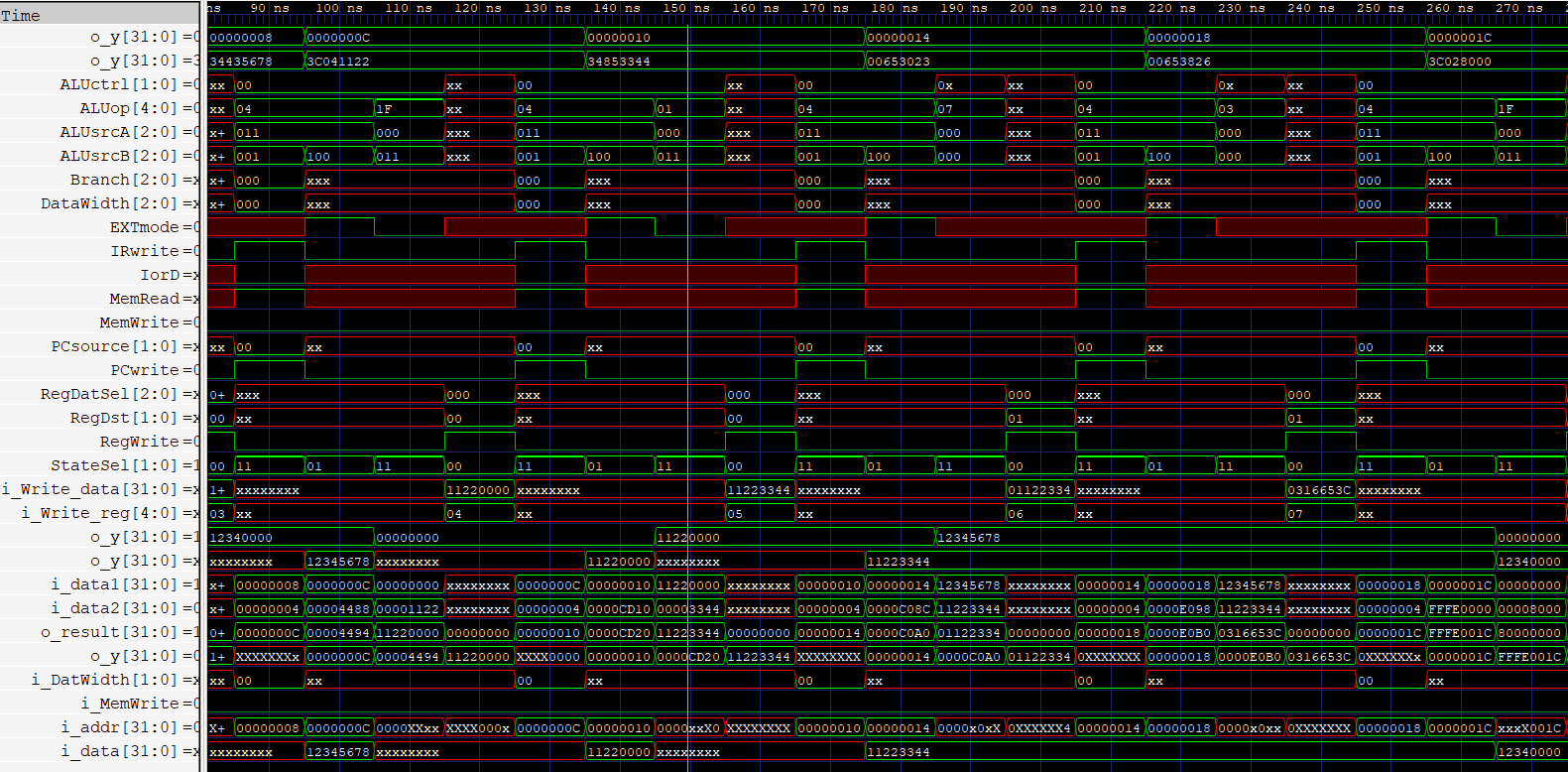
|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | 01 | address는 rd의 주소로 한다. |
| RegDatSel | 000 | 쓸 Data는 ALUOut으로부터 가져온다. |
| RegWrite | 1 | RF에서 쓰기 동작을 수행한다. |
| ALU | EXTmode | x | ALU는 아무 동작을 하지 않는다. |
| ALUsrcA | xxx |
| ALUsrcB | xxx |
| ALUop | xxxxx |
| ALUctrl | xx |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 00 | 다음 cycle이 존재하지 않기에 upc = 0를 진행한다. |

해당 명령어가 정상적으로 동작하는지 확인하기 위해 다음과 같은 TEXT를 작성하였다.

텍스트이(가) 표시된 사진

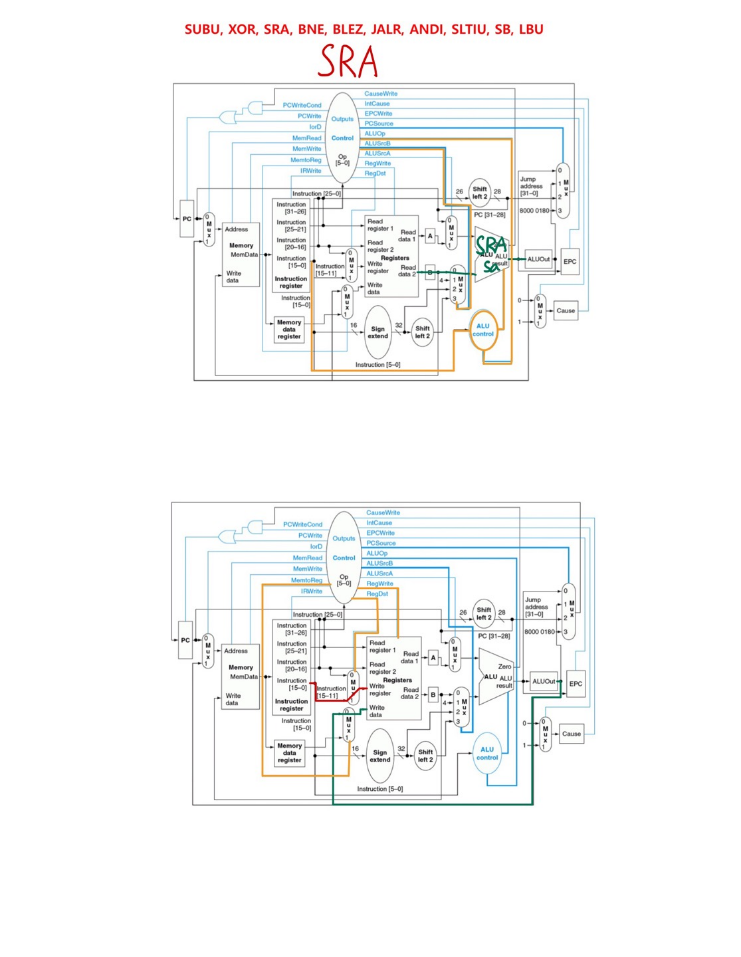
자동 생성된 설명

해당 명령은 이전의 testbench의 이후로 진행된다. 이번에는 0x12345678과 0x11223344와 XOR 연산을 진행하여 해당 결과를 $7에 저장하는 코드다. 해당 결과는 다음과 같이 나왔다.



결과 사진에서 주황 박스 영역이 XOR 연산이다. IF와 Decode된 상태 이후를 확인해보면 3번째 cycle에서는 ALU의 인자로 A와 B가 들어간 것을 확인할 수 있고 연산 결과로 0x12345678 XOR 0x11223344 = 0x0316653C로 계산기를 통해 확인한 값과 일치하는 것을 확인하였다. 4번째 cycle에서는 ALUOut에 저장되어 있는 연산 결과가 RegWrite 신호에 따라 0x07번지에 저장되는 것을 확인할 수 있다. 이를 통해 해당 명령어가 정상적으로 구현되었음을 확인하였다.

1. SRA



해당 명령어는 Shift Right Arithmethic 연산을 진행하는 명령어로 IF와 Decode 이외에 2 cycle이 추가로 동작한다(R-Type). 첫 cycle의 동작에서는 B(rt)만을 연산할 것을 선택하고, 연산의 경우 b >>> shift amount을 하도록 한다. 연산한 결과는 다음 cycle에서는 ALUOut에 저장되어 있기 때문에 이를 RF에 쓸 데이터로 설정하고 쓸 위치는 rd로 정한 후 쓰는 동작을 수행한다. 해당 동작을 signal로 정리하면 다음과 같다.

SRA 3rd cycle

|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | xx | Register File에 쓰이면 안 되기 때문에 RegWrite만 0으로 두고 나머지 신호는 사용하지 않는다. |
| RegDatSel | xxx |
| RegWrite | 0 |
| ALU | EXTmode | x | Immediate value를 사용하지 않는다. |
| ALUsrcA | xxx | 사용하지 않는다. |
| ALUsrcB | 000 | 000(rt)를 사용한다. |
| ALUop | 01111 | 01111(b >>> a)를 사용한다. |
| ALUctrl | 00 | a와 b의 순서를 바꾸지 않고 a를 sa로 대체한다. |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 11 | 다음 cycle이 존재하기에 upc++를 진행한다. |

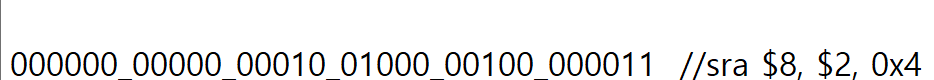
SRA 4th cycle

|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | 01 | address는 rd의 주소로 한다. |
| RegDatSel | 000 | 쓸 Data는 ALUOut으로부터 가져온다. |
| RegWrite | 1 | RF에서 쓰기 동작을 수행한다. |
| ALU | EXTmode | x | ALU는 아무 동작을 하지 않는다. |
| ALUsrcA | xxx |
| ALUsrcB | xxx |
| ALUop | xxxxx |
| ALUctrl | xx |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 00 | 다음 cycle이 존재하지 않기에 upc = 0를 진행한다. |

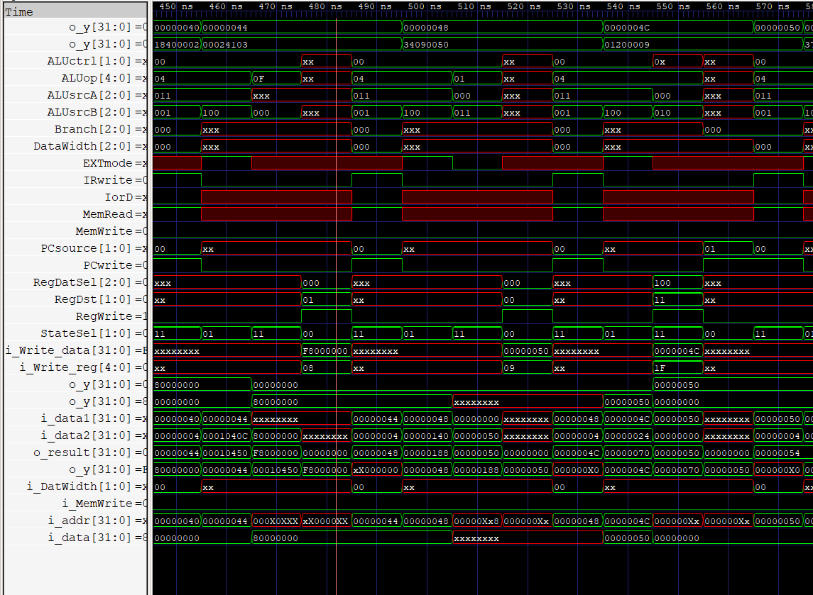
해당 명령어가 정상적으로 동작하는지 확인하기 위해 다음과 같은 TEXT를 작성하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명

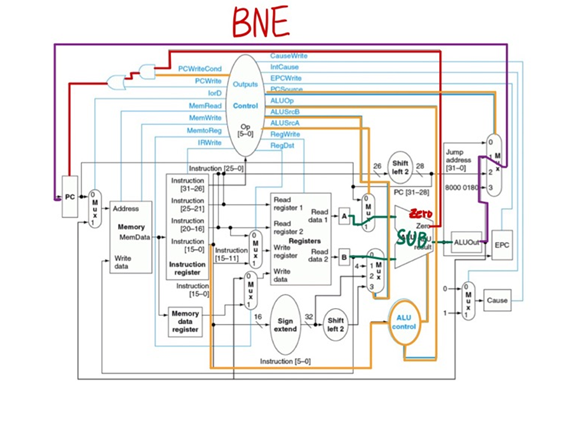


해당 명령은 이전의 testbench의 이후로 진행된다. 이번에는 $2에 0x80000000를 넣고 >>> 4를 진행한 뒤 결과를 $8에 저장하는 코드다. 해당 결과는 다음과 같이 나왔다.



결과 사진에서 주황 박스 영역이 SRA 연산이다. IF와 Decode된 상태 이후를 확인해보면 3번째 cycle에서는 ALU의 인자로 B만 들어간 것을 확인할 수 있고 연산 결과로 0x80000000 >>> 4 = 0xF8000000으로 계산기를 통해 확인한 값과 일치하는 것을 확인하였다. 4번째 cycle에서는 ALUOut에 저장되어 있는 연산 결과가 RegWrite 신호에 따라 0x08번지에 저장되는 것을 확인할 수 있다. 이를 통해 해당 명령어가 정상적으로 구현되었음을 확인하였다.

1. BNE



해당 명령어는 Branch if not equal을 진행하는 명령어로 IF와 Decode 이외에 1 cycle이 추가로 동작한다(I-Type). A(rs)와 B(rt)를 SUB 연산을 진행하여 해당 값의 state를 확인한다. 확인한 state가 0일 경우 기존에 2nd cycle에서 계산한 주소가 저장된 ALUOut으로 Branch를 시도한다. 해당 동작을 signal로 정리하면 다음과 같다.

BNE 3rd cycle

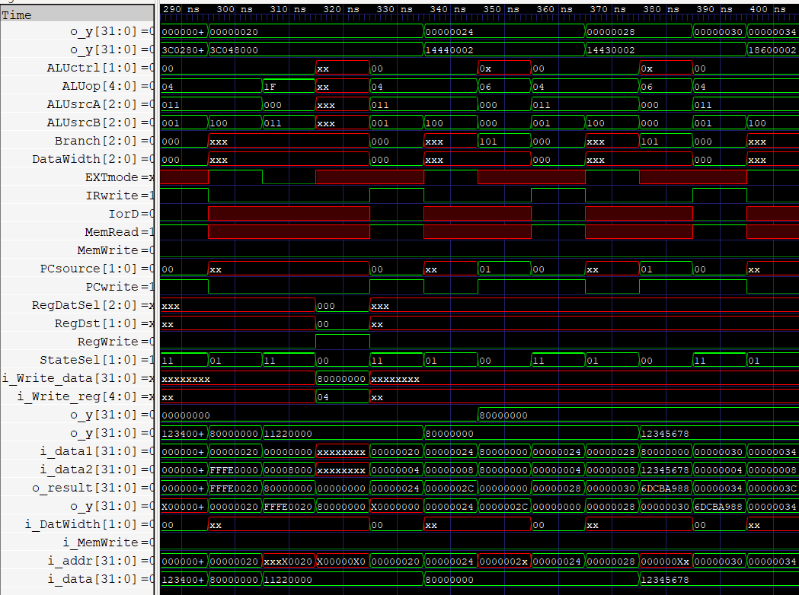
|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | xx | Register File에 쓰이면 안 되기 때문에 RegWrite만 0으로 두고 나머지 신호는 사용하지 않는다. |
| RegDatSel | xxx |
| RegWrite | 0 |
| ALU | EXTmode | x | Immediate value를 사용하지 않는다. |
| ALUsrcA | 000 | 000(rs)를 사용한다. |
| ALUsrcB | 000 | 000(rt)를 사용한다. |
| ALUop | 00110 | 00110(a - b)를 사용한다. |
| ALUctrl | 0x | a와 b의 순서를 바꾸지 않고 shift를 하지 않는다. |
| PC | Branch | 101 | Not equal일 때 Branch를 한다. |
| PCsrc | 01 | ALUOut(2nd cycle에서 계산한 주소)로 변경한다. |
| PCwrite | 1 | Branch일 경우 PC값을 변경한다. |
| uPC | stateSel | 00 | 다음 cycle이 존재하지 않기에 upc = 0를 진행한다. |

해당 명령어가 정상적으로 동작하는지 확인하기 위해 다음과 같은 TEXT를 작성하였다.

텍스트이(가) 표시된 사진

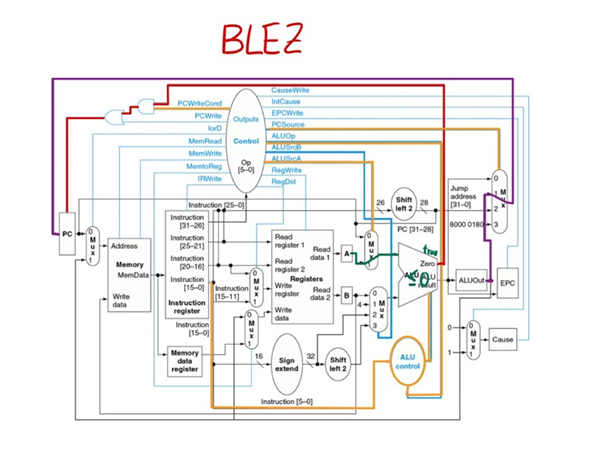
자동 생성된 설명

해당 명령은 이전의 testbench의 중간에 진행된다. 이번에는 $2와 $4에 0x80000000를 넣어둔다. 그 후 $2와 $4를 bne 0x2하였을 때(equal state)의 결과와 $2와 $3를 bne 0x2 하였을 때(not equal state)의 결과를 비교한다. 만약 equal state에서 branch가 된다면 x로 된 명령어줄에 들어가 오류가 발생할 것이고, not equal state에서 정상적으로 branch를 한다면 + 2 << 2 주소인 blez $3, 0x2로 이동할 것이다. 해당 결과는 다음과 같이 나왔다.



결과 사진에서 주황 박스 영역에서 두 영역이 BNE이다. 첫 BNE를 살펴보면 0x80000000(rs)와 0x80000000(rt)를 SUB 연산을 통해 0x0임을 확인한 후 같기 때문에 Branch를 하지 않아 PC값이 PC+4로 이동한 모습을 볼 수 있다. 다음 BNE에서는 0x80000000(rs)와 0x12345678(rt)를 SUB 연산을 통해 같지 않음을 확인한 후 Branch를 통해 PC값이 변경되어 다음 cycle에서 PC의 값이 +( 4 + (0x2 << 1))이 된 것을 확인할 수 있다. 이를 통해 BNE가 정상적으로 구현되었음을 확인하였다.

1. BLEZ



해당 명령어는 Branch if less equal to zero을 진행하는 명령어로 IF와 Decode 이외에 1 cycle이 추가로 동작한다(I-Type). A(rs)와 0을 ADD 연산을 진행하여 해당 값의 state를 확인한다. 확인한 state가 <= 0일 경우 기존에 2nd cycle에서 계산한 주소가 저장된 ALUOut으로 Branch를 시도한다. 해당 동작을 signal로 정리하면 다음과 같다.

BLEZ 3rd cycle

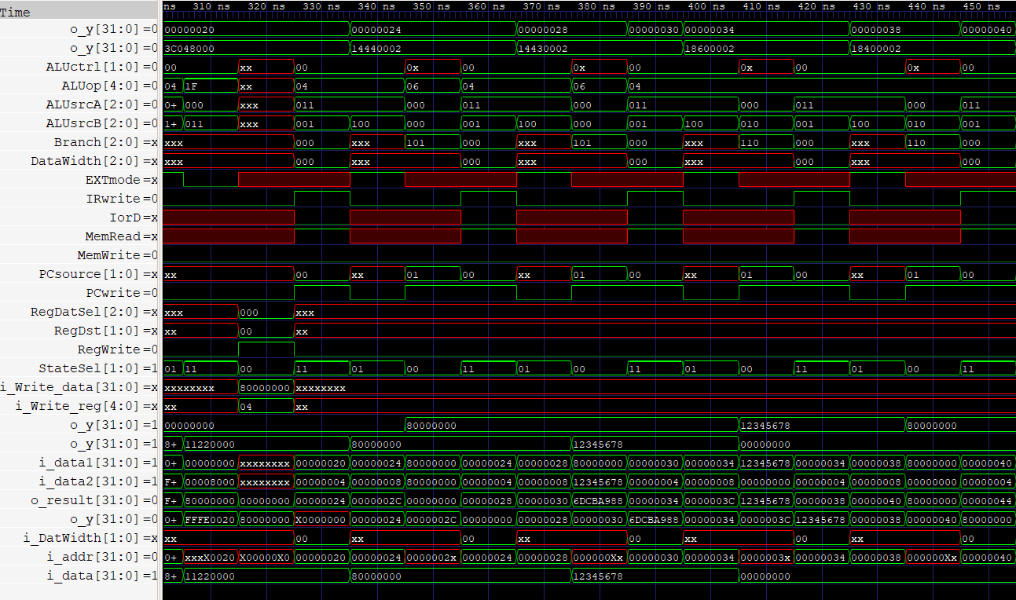
|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | xx | Register File에 쓰이면 안 되기 때문에 RegWrite만 0으로 두고 나머지 신호는 사용하지 않는다. |
| RegDatSel | xxx |
| RegWrite | 0 |
| ALU | EXTmode | x | Immediate value를 사용하지 않는다. |
| ALUsrcA | 000 | 000(rs)를 사용한다. |
| ALUsrcB | 010 | 010(0)를 사용한다. |
| ALUop | 00100 | 00100(a + b = a + 0 = a)를 사용한다. |
| ALUctrl | 0x | a와 b의 순서를 바꾸지 않고 shift를 하지 않는다. |
| PC | Branch | 110 | not positive일 때 Branch를 한다. |
| PCsrc | 01 | ALUOut(2nd cycle에서 계산한 주소)로 변경한다. |
| PCwrite | 1 | Branch일 경우 PC값을 변경한다. |
| uPC | stateSel | 00 | 다음 cycle이 존재하지 않기에 upc = 0를 진행한다. |

해당 명령어가 정상적으로 동작하는지 확인하기 위해 다음과 같은 TEXT를 작성하였다.

텍스트이(가) 표시된 사진

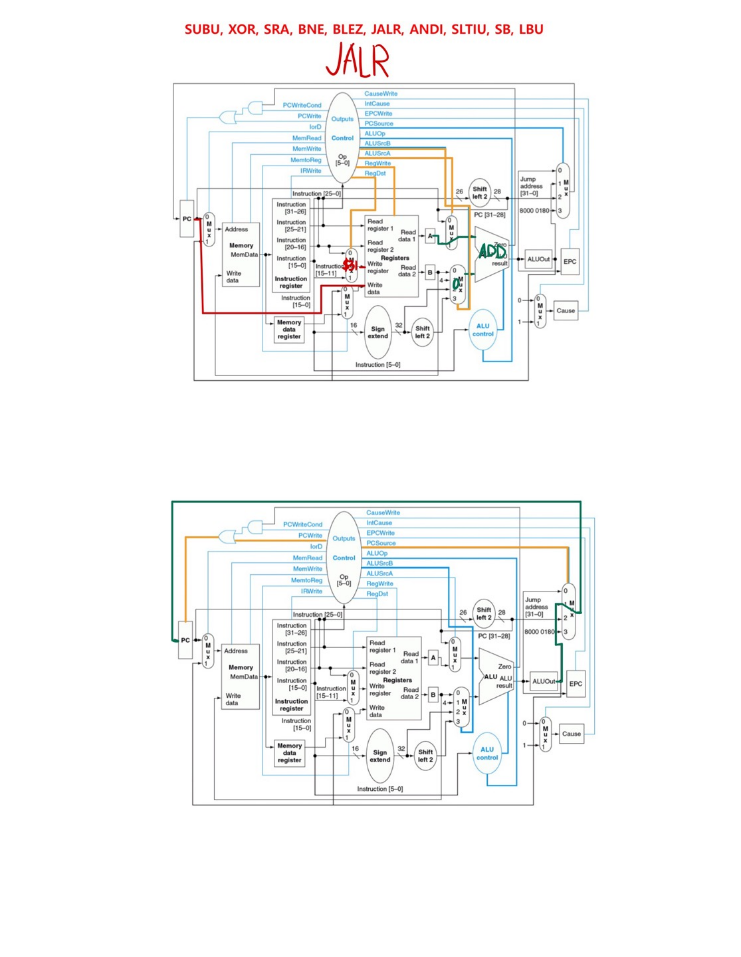
자동 생성된 설명

해당 명령은 이전의 testbench의 이후에 진행된다. 이번에는 $3(0x12345678)를 blez 0x2하였을 때(> 0)의 결과와 $2를 blez 0x2 하였을 때(<=0)의 결과를 비교한다. 만약 >0 state에서 branch가 된다면 x로 된 명령어줄에 들어가 오류가 발생할 것이고, <=0 state에서 정상적으로 branch를 한다면 + 2 << 2 주소인 sra $8, $2, 0x4로 이동할 것이다. 해당 결과는 다음과 같이 나왔다.



결과 사진에서 주황 박스 영역에서 두 영역이 BLEZ이다. 첫 BLEZ를 살펴보면 0x12345678(rs)와 0x0(0)를 ADD 연산을 통해 rs>0임을 확인한 후 Branch를 하지 않아 PC값이 PC+4로 이동한 모습을 볼 수 있다. 다음 BLEZ에서는 0x80000000(rs)와 0x0(0)를 ADD 연산을 통해 rs<=0을 확인한 후 Branch를 통해 PC값이 변경되어 다음 cycle에서 PC의 값이 +( 4 + (0x2 << 1))이 된 것을 확인할 수 있다. 이를 통해 BLEZ가 정상적으로 구현되었음을 확인하였다.

1. JALR



해당 명령어는 Jump address and save Load Register 연산을 진행하는 명령어로 IF와 Decode 이외에 2 cycle이 추가로 동작한다(R-Type). 첫 cycle의 동작에서는 우선 현재 PC의 값을 LR에 저장한다. 그리고 ALU에서는 A(rs)와 0를 연산할 것을 선택하고, 연산의 경우 ADD을 하도록 한다. 연산한 결과는 다음 cycle에서 ALUOut에 저장되어 있기 때문에 jump에 사용할 위치로 설정하고 PC값을 바꾼다.

JALR 3rd cycle

|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | 11 | lr($31)에 저장한다. |
| RegDatSel | 100 | 100(PC)값을 저장한다. |
| RegWrite | 1 | Register File에 쓴다. |
| ALU | EXTmode | x | Immediate value를 사용하지 않는다. |
| ALUsrcA | 000 | 000(rs)를 사용한다. |
| ALUsrcB | 010 | 010(0)를 사용한다. |
| ALUop | 00100 | ADD를 사용한다. |
| ALUctrl | 0x | a와 b의 순서를 바꾸지 않고 shift 또한 진행하지 않는다. |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 11 | 다음 cycle이 존재하기에 upc++를 진행한다. |

JALR 4th cycle

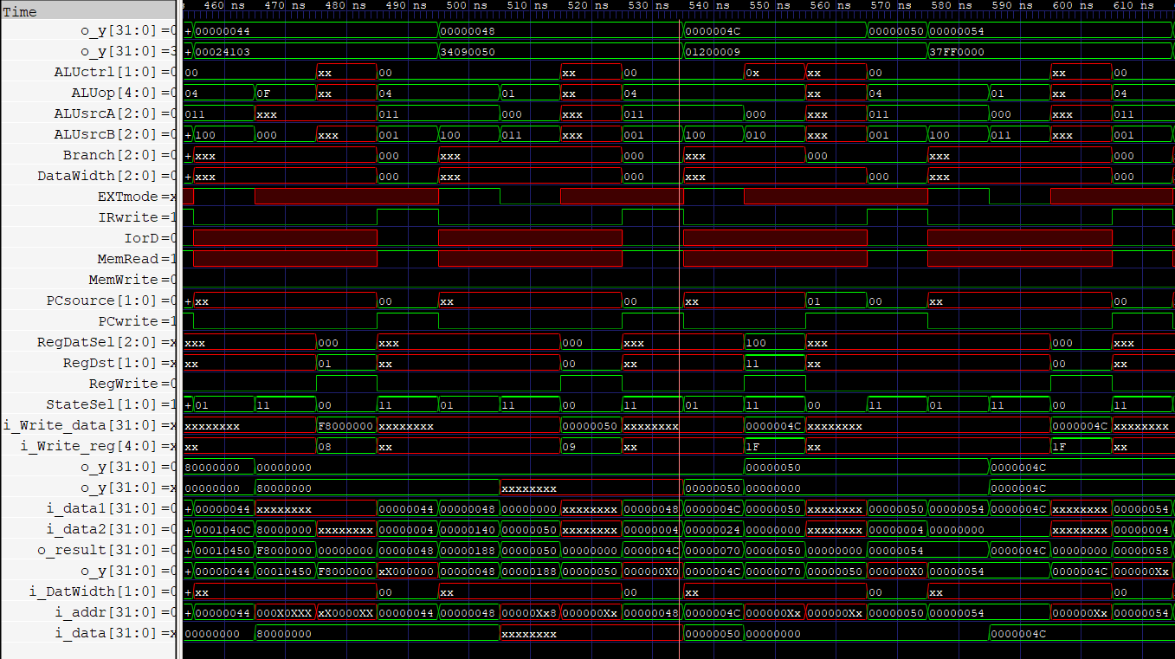
|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | xx | Register File에 쓰이면 안 되기 때문에 RegWrite만 0으로 두고 나머지 신호는 사용하지 않는다. |
| RegDatSel | xxx |
| RegWrite | 0 |
| ALU | EXTmode | x | ALU는 아무 동작을 하지 않는다. |
| ALUsrcA | xxx |
| ALUsrcB | xxx |
| ALUop | xxxxx |
| ALUctrl | xx |
| PC | Branch | 000 | Jump를 한다. |
| PCsrc | 01 | ALUOut의 위치로 Jump한다. |
| PCwrite | 1 | PC에 write를 시도한다. |
| uPC | stateSel | 00 | 다음 cycle이 존재하지 않기에 upc = 0를 진행한다. |

해당 명령어가 정상적으로 동작하는지 확인하기 위해 다음과 같은 TEXT를 작성하였다.

텍스트이(가) 표시된 사진

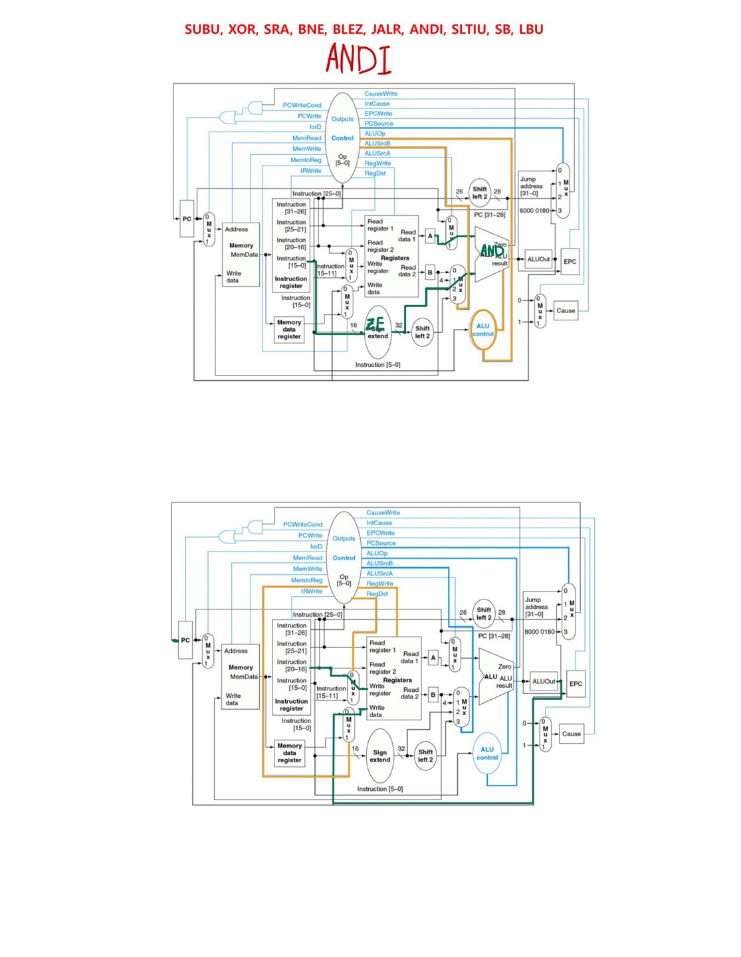
자동 생성된 설명

해당 명령은 sra testbench의 이후로 진행된다. ori를 통해 $9에 0x50의 값을 저장한 후 jalr를 하여 $50주소에 해당하는 ori $31, $41, 0x0으로 이동하는지 확인한다. 해당 결과는 다음과 같이 나왔다.



결과 사진에서 주황 박스 영역이 JALR 연산이다. 해당 결과를 살펴보면 3번째 cycle에서 RF에는 현재의 pc + 4에 해당하는 4C가 $31주소에 저장되는 것을 확인할 수 있다. ALU에서는 0x50(rs)와 0을 더하여 rs값을 그대로 연산결과로 출력하고 있다. 다음 cycle에서 Branch를 통해 ALUOut에 저장된 연산 결과로 jump하는 것을 확인할 수 있다. 이를 통해 해당 명령어가 정상적으로 구현되었음을 확인하였다.

1. ANDI



해당 명령어는 AND with Immediate value 연산을 진행하는 명령어로 IF와 Decode 이외에 2 cycle이 추가로 동작한다(I-Type). 첫 cycle의 동작에서는 A(rs)와 zero Extension을 한 value를 연산할 것을 선택하고, 연산의 경우 AND을 하도록 한다. 연산한 결과는 다음 cycle에서는 ALUOut에 저장되어 있기 때문에 이를 RF에 쓸 데이터로 설정하고 쓸 위치는 rt로 정한 후 쓰는 동작을 수행한다. 해당 동작을 signal로 정리하면 다음과 같다.

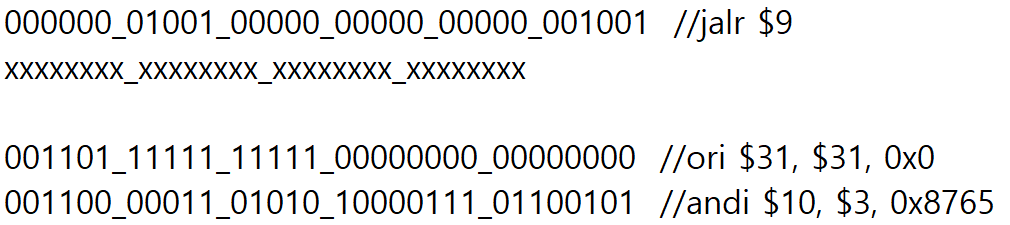
ANDI 3rd cycle

|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | xx | Register File에 쓰이면 안 되기 때문에 RegWrite만 0으로 두고 나머지 신호는 사용하지 않는다. |
| RegDatSel | xxx |
| RegWrite | 0 |
| ALU | EXTmode | 0 | zero extension을 한 value를 사용한다. |
| ALUsrcA | 000 | 000(rs)를 사용한다. |
| ALUsrcB | 011 | 011(zero extension immediate value)를 사용한다. |
| ALUop | 00000 | Bitwise AND를 사용한다. |
| ALUctrl | 0x | a와 b의 순서를 바꾸지 않고 shift 또한 진행하지 않는다. |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 11 | 다음 cycle이 존재하기에 upc++를 진행한다. |

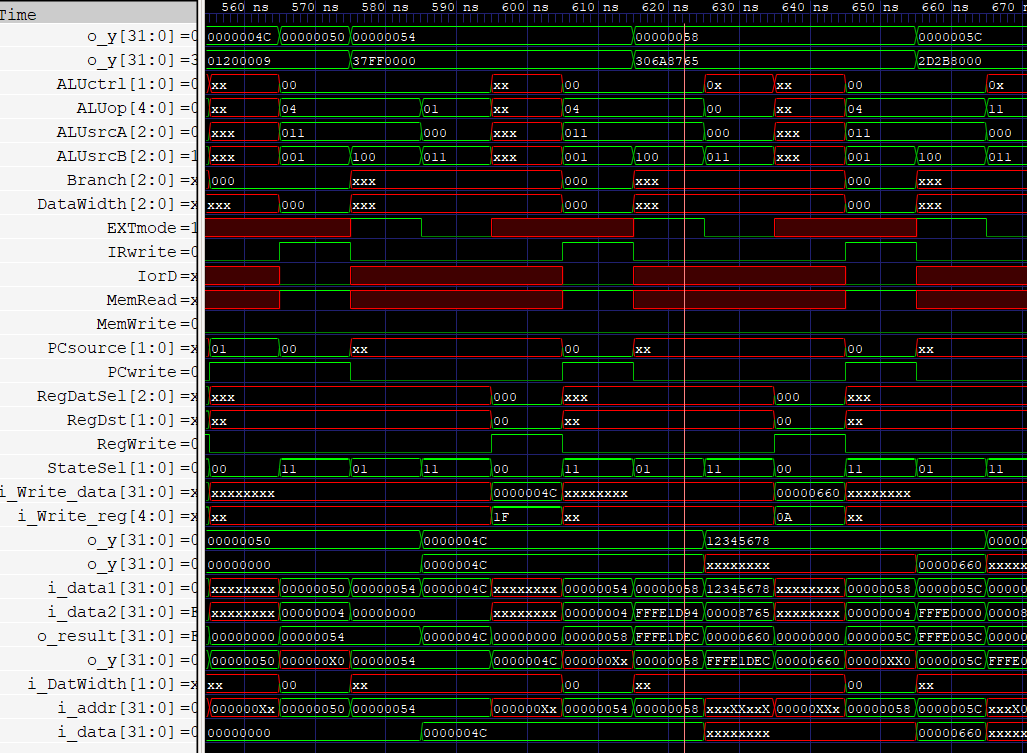
ANDI 4th cycle

|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | 00 | address는 rt의 주소로 한다. |
| RegDatSel | 000 | 쓸 Data는 ALUOut으로부터 가져온다. |
| RegWrite | 1 | RF에서 쓰기 동작을 수행한다. |
| ALU | EXTmode | x | ALU는 아무 동작을 하지 않는다. |
| ALUsrcA | xxx |
| ALUsrcB | xxx |
| ALUop | xxxxx |
| ALUctrl | xx |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 00 | 다음 cycle이 존재하지 않기에 upc = 0를 진행한다. |

해당 명령어가 정상적으로 동작하는지 확인하기 위해 다음과 같은 TEXT를 작성하였다.

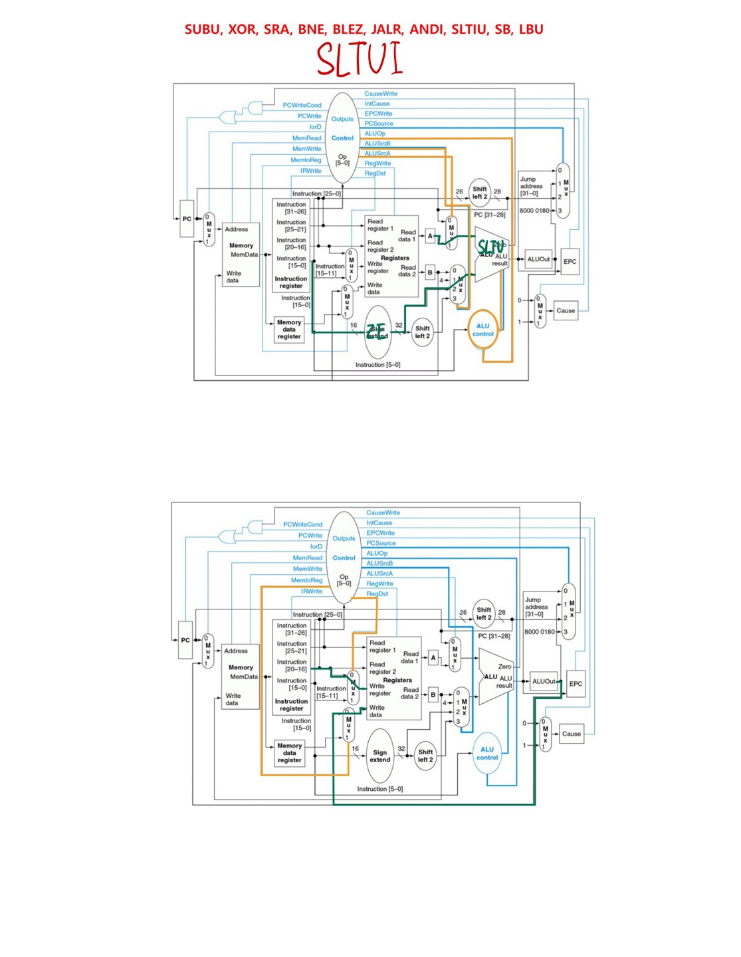


해당 명령은 이전의 testbench의 이후로 진행된다. $3(0x12345678)과 0x8765를 AND연산한 결과를 $10에 저장한다. 해당 결과는 다음과 같이 나왔다.



결과 사진에서 주황 박스 영역이 ANDI 연산이다. IF와 Decode된 상태 이후를 확인해보면 3번째 cycle에서는 ALU의 인자로 A와 zero extention한 immediate value, 0x00008765가 들어간 것을 확인할 수 있고 연산 결과로 0x12345678 XOR 0x00008765 = 0x00000660로 계산기를 통해 확인한 값과 일치하는 것을 확인하였다. 4번째 cycle에서는 ALUOut에 저장되어 있는 연산 결과가 RegWrite 신호에 따라 0xA번지에 저장되는 것을 확인할 수 있다. 이를 통해 해당 명령어가 정상적으로 구현되었음을 확인하였다.

1. SLTIU



해당 명령어는 Set on Less than Immediate unsigned value 연산을 진행하는 명령어로 IF와 Decode 이외에 2 cycle이 추가로 동작한다(I-Type). 첫 cycle의 동작에서는 A(rs)와 zero Extension을 한 value를 연산할 것을 선택하고, 연산의 경우 SLT을 하도록 한다. 연산한 결과는 다음 cycle에서는 ALUOut에 저장되어 있기 때문에 이를 RF에 쓸 데이터로 설정하고 쓸 위치는 rt로 정한 후 쓰는 동작을 수행한다. 해당 동작을 signal로 정리하면 다음과 같다.

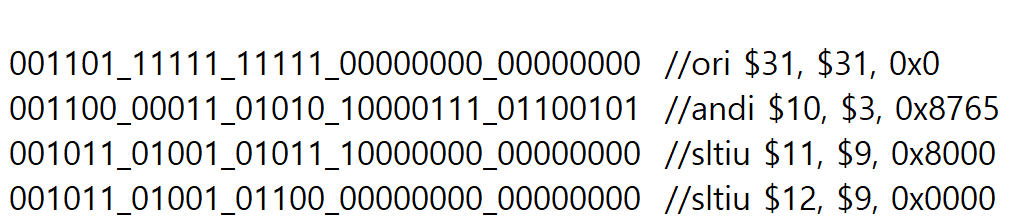
SLTIU 3rd cycle

|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | xx | Register File에 쓰이면 안 되기 때문에 RegWrite만 0으로 두고 나머지 신호는 사용하지 않는다. |
| RegDatSel | xxx |
| RegWrite | 0 |
| ALU | EXTmode | 0 | zero extension을 한 value를 사용한다. |
| ALUsrcA | 000 | 000(rs)를 사용한다. |
| ALUsrcB | 011 | 011(zero extension immediate value)를 사용한다. |
| ALUop | 10001 | Unsigned SLT를 사용한다. |
| ALUctrl | 0x | a와 b의 순서를 바꾸지 않고 shift 또한 진행하지 않는다. |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 11 | 다음 cycle이 존재하기에 upc++를 진행한다. |

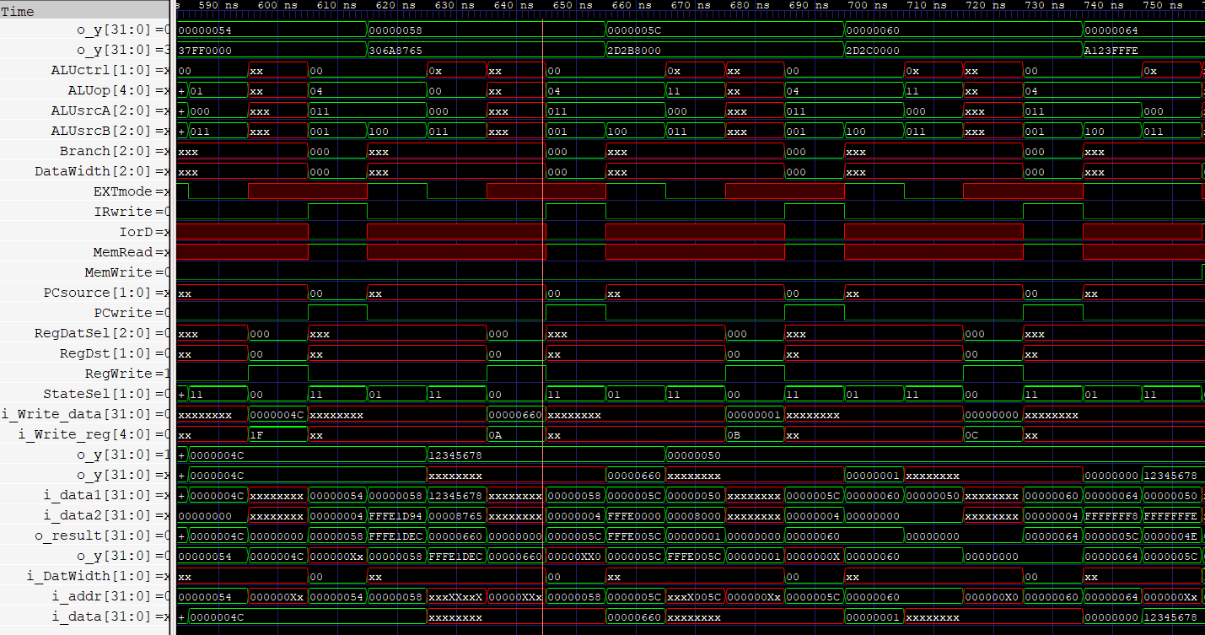
SLTIU 4th cycle

|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | 00 | address는 rt의 주소로 한다. |
| RegDatSel | 000 | 쓸 Data는 ALUOut으로부터 가져온다. |
| RegWrite | 1 | RF에서 쓰기 동작을 수행한다. |
| ALU | EXTmode | x | ALU는 아무 동작을 하지 않는다. |
| ALUsrcA | xxx |
| ALUsrcB | xxx |
| ALUop | xxxxx |
| ALUctrl | xx |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 00 | 다음 cycle이 존재하지 않기에 upc = 0를 진행한다. |

해당 명령어가 정상적으로 동작하는지 확인하기 위해 다음과 같은 TEXT를 작성하였다.

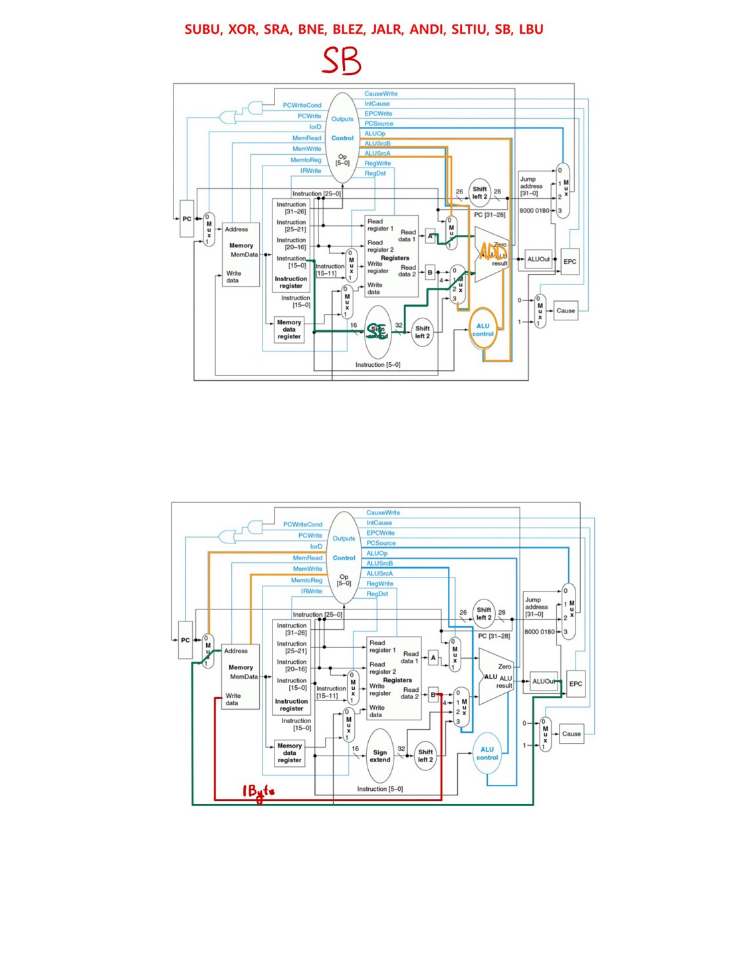


해당 명령은 이전의 testbench의 이후로 진행된다. $9(0x50)과 immediate value 0x8000, 0x0을 비교하여 0x50 < 0x8000인 케이스와 0x50 > 0x0인 케이스의 결과를 확인한다. 결과는 다음과 같다.



결과 사진에서 주황 박스 영역이 SLTIU 연산이다. IF와 Decode된 상태 이후를 확인해보면 첫 결과에서는 0x50(rs) < 0x8000(zero extension)이기 때문에 SLT로 나온 상태는 1이다. 두 번째 결과에서는 0x50(rs) >= 0x0(zero extension)이기 때문에 SLT로 나온 상태는 0이다. 해당 결과들은 4번째 cycle에서 각각 $11, $12에 정상적으로 저장되는 것을 확인할 수 있다. 이를 통해 해당 명령어가 정상적으로 구현되었음을 확인하였다.

1. SB



해당 명령어는 Store Byte 연산을 진행하는 명령어로 IF와 Decode 이외에 2 cycle이 추가로 동작한다(I-Type). 첫 cycle의 동작에서는 A(rs)와 sign Extension을 한 value를 연산할 것을 선택하고, 연산의 경우 ADD을 하도록 한다. 연산한 결과(저장 위치)는 다음 cycle에서는 ALUOut에 저장되어 있다. 해당 값을 Memory에 저장할 위치로 설정하고 저장할 값을 B(rt)로 설정하여 1byte 저장을 진행한다. 해당 동작을 signal로 정리하면 다음과 같다.

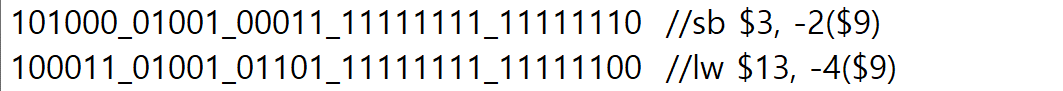
SB 3rd cycle

|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | xx | Register File에 쓰이면 안 되기 때문에 RegWrite만 0으로 두고 나머지 신호는 사용하지 않는다. |
| RegDatSel | xxx |
| RegWrite | 0 |
| ALU | EXTmode | 1 | sign extension을 한 value를 사용한다. |
| ALUsrcA | 000 | 000(rs)를 사용한다. |
| ALUsrcB | 011 | 011(sign extension immediate value)를 사용한다. |
| ALUop | 00100 | ADD를 사용한다. |
| ALUctrl | 0x | a와 b의 순서를 바꾸지 않고 shift 또한 진행하지 않는다. |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 11 | 다음 cycle이 존재하기에 upc++를 진행한다. |

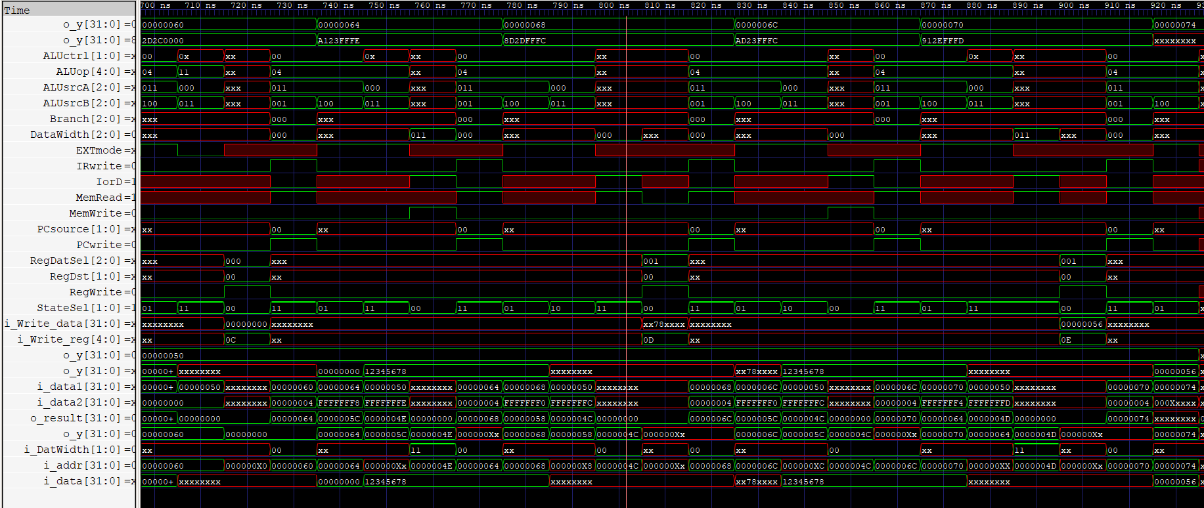
SB 4th cycle

|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | 1 | ALUOut의 결과를 주소로 한다. |
| MemRead | x | Memory를 읽지 않는다. |
| MemWrite | 1 | Memory Write 동작을 수행한다. |
| DatWidth | 011 | 8 byte를 저장한다. |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | xx | Register File에 쓰이면 안 되기 때문에 RegWrite만 0으로 두고 나머지 신호는 사용하지 않는다. |
| RegDatSel | xxx |
| RegWrite | 0 |
| ALU | EXTmode | x | ALU는 아무 동작을 하지 않는다. |
| ALUsrcA | xxx |
| ALUsrcB | xxx |
| ALUop | xxxxx |
| ALUctrl | xx |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 00 | 다음 cycle이 존재하지 않기에 upc = 0를 진행한다. |

해당 명령어가 정상적으로 동작하는지 확인하기 위해 다음과 같은 TEXT를 작성하였다.

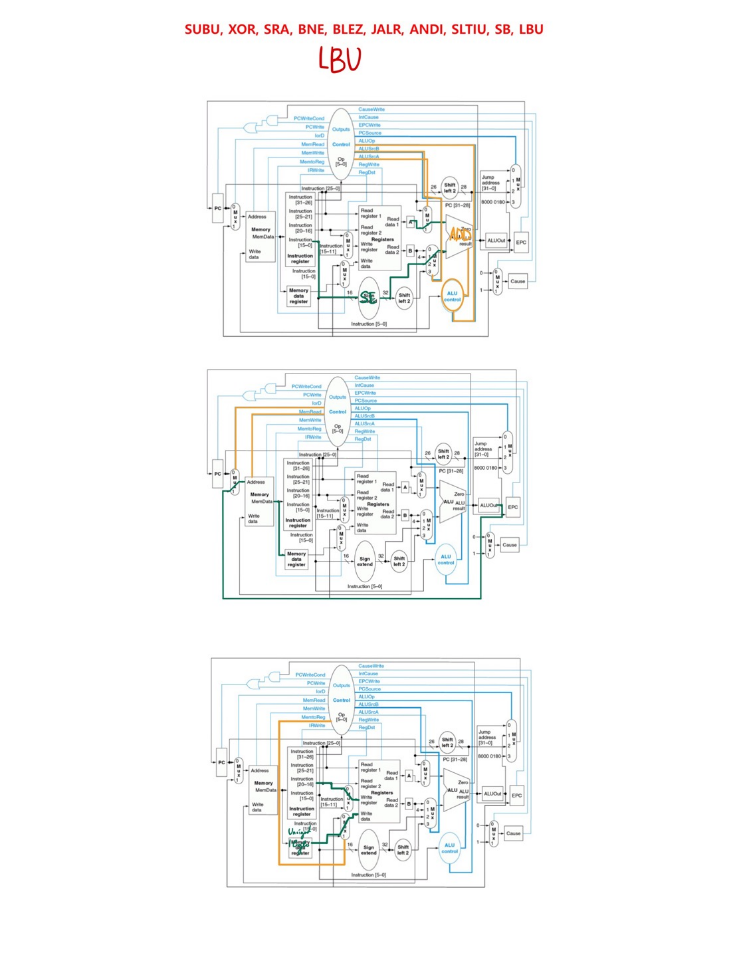


해당 명령은 이전의 testbench의 이후로 진행된다. $9(0x50)에서 -2 offset 위치에 $3(0x12345678)중 1byte를 저장한다. 해당 Processor는 little endian이기 때문에 LSB에 해당하는 0x78이 저장될 것이다. 해당 결과를 보면 다음과 같다.



결과 사진에서 주황 박스 영역이 SB이다. IF와 Decode된 상태 이후를 확인해보면 첫 cycle에서는 0x50(rs)에서 0xFFFE(sign extension)을 더한 위치인 0x4E가 ALU에서 출력된다. 이후 Memory는 해당 값에 해당하는 주소에 $rt가 가진 값 0x12345678에서 1byte인 0x78을 저장한다. 저장한 값은 다음 LW를 통해 확인할 수 있다. 저장한 값을 불러올 때 결과를 보면 0x50 - 0x2 = 0x4E 위치에 해당하는 곳에 데이터가 존재하는 것을 확인할 수 있다. 이를 통해 해당 명령어가 정상적으로 구현되었음을 확인하였다.

1. LBU



해당 명령어는 Load unsigned Byte 연산을 진행하는 명령어로 IF와 Decode 이외에 3 cycle이 추가로 동작한다(I-Type, LOAD). 첫 cycle의 동작에서는 A(rs)와 sign Extension을 한 value를 연산할 것을 선택하고, 연산의 경우 ADD을 하도록 한다. 연산한 결과(저장 위치)는 다음 cycle에서는 ALUOut에 저장되어 있다. 해당 값을 Memory에 접근할 위치로 설정하고 1byte 읽기를 진행한다. 마지막 cycle에서 RF는 읽어들인 값이 저장된 Memory Data Register의 데이터를 $rt위치에 저장한다. 해당 동작을 signal로 정리하면 다음과 같다.

LBU 3rd cycle

|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | xx | Register File에 쓰이면 안 되기 때문에 RegWrite만 0으로 두고 나머지 신호는 사용하지 않는다. |
| RegDatSel | xxx |
| RegWrite | 0 |
| ALU | EXTmode | 1 | sign extension을 한 value를 사용한다. |
| ALUsrcA | 000 | 000(rs)를 사용한다. |
| ALUsrcB | 011 | 011(sign extension immediate value)를 사용한다. |
| ALUop | 00100 | ADD를 사용한다. |
| ALUctrl | 0x | a와 b의 순서를 바꾸지 않고 shift 또한 진행하지 않는다. |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 11 | 다음 cycle이 존재하기에 upc++를 진행한다. |

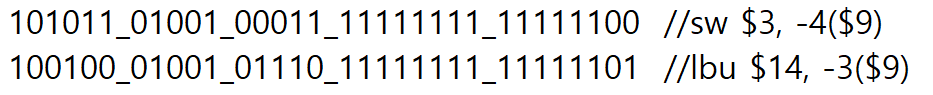
LBU 4th cycle

|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | 1 | ALUOut의 결과를 주소로 한다. |
| MemRead | 1 | Memory를 읽지 않는다. |
| MemWrite | 0 | Memory Write 동작을 수행한다. |
| DatWidth | 011 | 8 byte를 읽는다. |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | xx | Register File에 쓰이면 안 되기 때문에 RegWrite만 0으로 두고 나머지 신호는 사용하지 않는다. |
| RegDatSel | xxx |
| RegWrite | 0 |
| ALU | EXTmode | x | ALU는 아무 동작을 하지 않는다. |
| ALUsrcA | xxx |
| ALUsrcB | xxx |
| ALUop | xxxxx |
| ALUctrl | xx |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 11 | 다음 cycle이 존재하기에 upc++를 진행한다. |

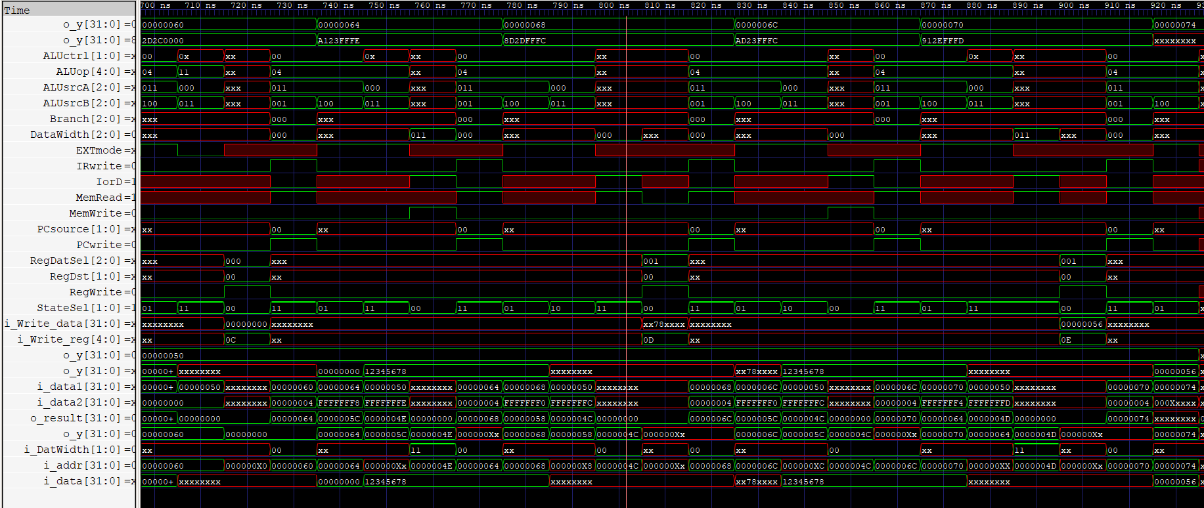
LBU 5th cycle

|  |  |  |  |
| --- | --- | --- | --- |
| Field | signal | state | Cause |
| Mem | lorD | x | 메모리와 관련된 사용을 하지 않기 때문에 MemWrite를 0으로 두고 나머지 신호는 사용하지 않는다. |
| MemRead | x |
| MemWrite | 0 |
| DatWidth | xxx |
| IR | IRwrite | 0 | Instruction이 바뀌면 안 된다. |
| RF | RegDst | 00 | rt에 쓰기 동작을 한다. |
| RegDatSel | 001 | Memory Data Register의 값을 저장한다. |
| RegWrite | 1 | Write 동작을 수행한다. |
| ALU | EXTmode | x | ALU는 아무 동작을 하지 않는다 |
| ALUsrcA | xxx |
| ALUsrcB | xxx |
| ALUop | xxxxx |
| ALUctrl | xx |
| PC | Branch | xxx | PC값을 변경하지 않기 때문에 PCwrite 신호만 0으로 두고 나머지 신호는 사용하지 않는다. |
| PCsrc | xx |
| PCwrite | 0 |
| uPC | stateSel | 00 | 다음 cycle이 존재하지 않기에 upc = 0를 진행한다. |

해당 명령어가 정상적으로 동작하는지 확인하기 위해 다음과 같은 TEXT를 작성하였다.



해당 명령은 이전의 testbench의 이후로 진행된다. $9(0x50)에서 -4 offset 위치에 $3(0x12345678)를 저장한다. 이후에 LBU를 통해 0x50위치에서 -3 offset에 해당하는 위치의 값을 읽어온다. 해당 Processor는 little endian이기 때문에 읽어올 때 0x56이 읽혀야 한다. 해당 결과를 보면 다음과 같다.



결과 사진에서 주황 박스 영역이 LBU이다. IF와 Decode된 상태 이후를 확인해보면 첫 cycle에서는 0x50(rs)에서 0xFFFD(sign extension)을 더한 위치인 0x4D가 ALU에서 출력된다. 이후 Memory에서 해당 값의 주소에서 1byte를 읽어온다. 다음 cycle에서는 읽어온 데이터가 $14에 저장되는 것을 확인할 수 있다. 읽어온 데이터는 0x56으로 위에서 예측한 것과 동일하게 출력되었다. 이를 통해 해당 명령어가 정상적으로 구현되었음을 확인하였다.

1. Consideration

* 해당 과제를 통해 이전과 같은 명령어가 multicycle processor에서는 어떻게 동작하는지에 대하여 더욱 명확하게 개념을 정리할 수 있었다. 특히 LOAD와 STORE의 과정에서 어떠한 방식으로 메모리에 접근하여 읽고 쓰는지에 대하여 메커니즘을 이해할 수 있었다. 또한 uPC를 활용하여 ROM을 작성하는 방법을 해당 과제를 통해 공부할 수 있었다. 해당 방식과 같은 Multicycle processor가 이러한 방식 때문에 명령어의 추가가 쉬워 CISC가 발전하였다는 수업시간의 설명이 이해가 되었다.

1. Reference

* 강의자료만을 참고