컴퓨터 구조 프로젝트0 보고서

학번 : 2018202046
이름 : 이준휘
교수님 : 이성원
분반 : 월3, 수4

A. Introduction

해당 과제에서는 기존에 일부의 기능이 구현된 MIPS에서 아직 만들어지지 못한 명령어를 구현하여 완성시키는 것을 목표로 한다. 기존에 구현된 명령어로는 lw, sw, ori, jump, lui, llo, lhi 명령어가 있다. 명령어의 구현은 PLA_AND.txt와 PLA_OR.txt를 완성하는 것으로 구현할 수 있다. PLA_AND에는 Opcode와 Function에 따라 Decoding을 통해 명령어를 구별하는 역할을 수행하며, PLA_OR에서는 각 명령어에 맞는 control를 위한 signal을 조정하는 역할을 수행한다.

B. Result

1. SUBU

SUBU의 동작은 $s와 $t 레지스터로부터 값을 불러와 해당 값을 unsigned sub 연산을 수행한 뒤 이를 $d에 저장한다. 해당 명령은 R-type임으로 Opcode는 000000이며 function은 100011이다.

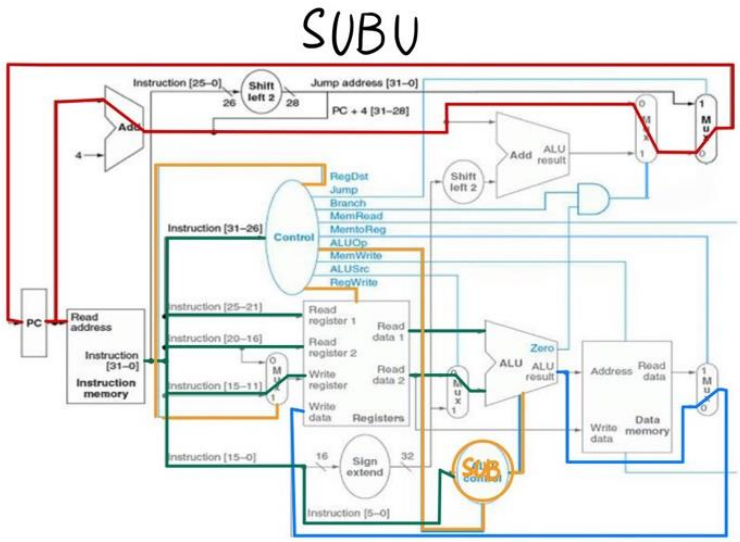
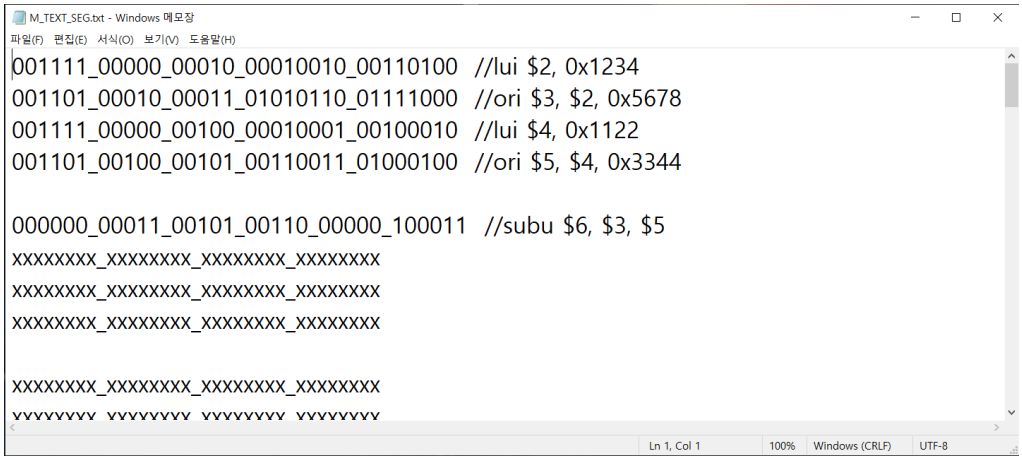
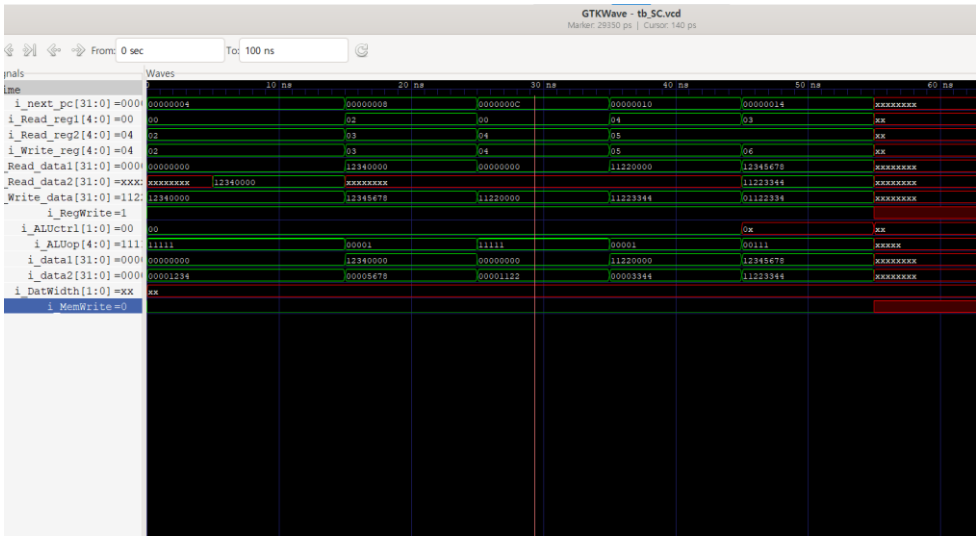


Figure 1 - The single cycle CPU datapath and control path

해당 명령어의 동작을 그림으로 표현하면 위와 같다 우선 IM에서 나온 Instruction은 op, $s, $t, $d, func으로 분리된다. Register File은 2개의 주소를 읽고 이를 쓰는 동작을 수행함으로 RegDst의 동작은 01이 수행된다. 또한 쓰는 동작을 수행함으로 RegWrite 동작 또한 1로 수행된다. RF를 나온 데이터 두 개를 연산하게 된다. 이로 인해 ARUsrcB는 00으로 동작한다. ALUctrl의 경우 해당 연산은 input의 위치를 바꾸지 않고 shift 또한 하지 않음으로 0x로 동작한다. 또한 immediate value를 쓰지 않음으로 EXTmode = x이다. ALUop는 Unsigned a-b 동작인 00111로 설정된다. 메모리는 사용되지 않음으로 MemWrite = 0, DatWidth = xxx로 설정되며, ALU 데이터를 register로 보내기 때문에 MemtoReg = 0이 된다. RegDatSel는 ALU/MEM의 데이터를 보내야 하기 때문에 00으로 설정된다. pc = pc+4로 동작해야 함으로 Branch = 000, jump동작은 하지 않기 때문에 jump = 00으로 동작한다.



동작의 의도는 다음과 같다. lui와 ori를 통해 만들어진 0x12345678과 0x11223344를 subu하는 동작을 수행하여 해당 결과를 관찰한다. 이를 통한 결과는 아래와 같이 나왔다.



해당 결과를 살펴보면 기존에 ori lui 동작은 주어진 명령어이기 때문에 정상적으로 동작한다. 5번째 동작에서 subu $6, $3, $5 동작을 수행하였을 때 2개의 값이 alu에서 subu연산을 하여 0x01122334로 값이 바뀌어 저장되는 것을 볼 수 있다. 이를 통해 해당 명령어가 정상적으로 동작함을 알 수 있다.

2. XOR

XOR의 동작은 $s와 $t 레지스터로부터 값을 불러와 해당 값을 xor 연산을 수행한 뒤 이를 $d에 저장한다. 해당 명령은 R-type임으로 Opcode는 000000이며 function은 100110이다.

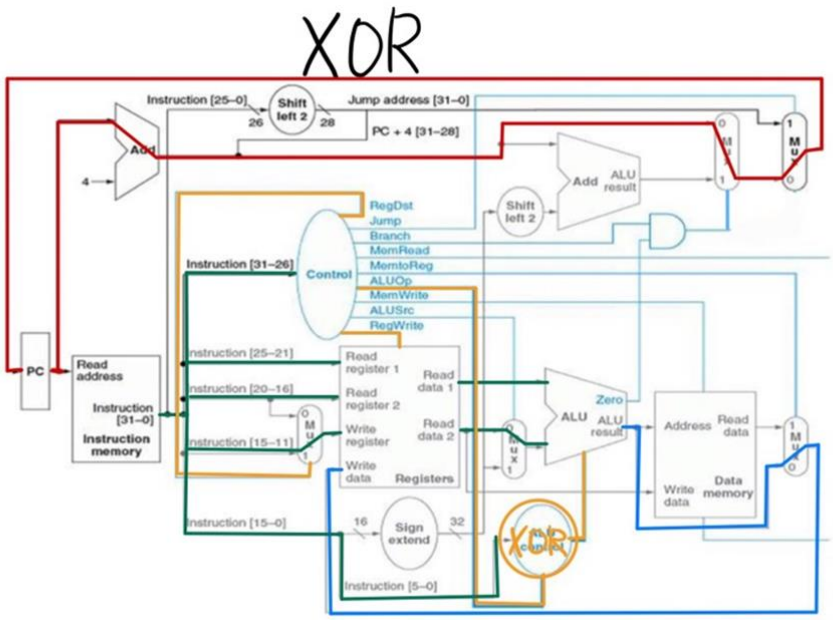
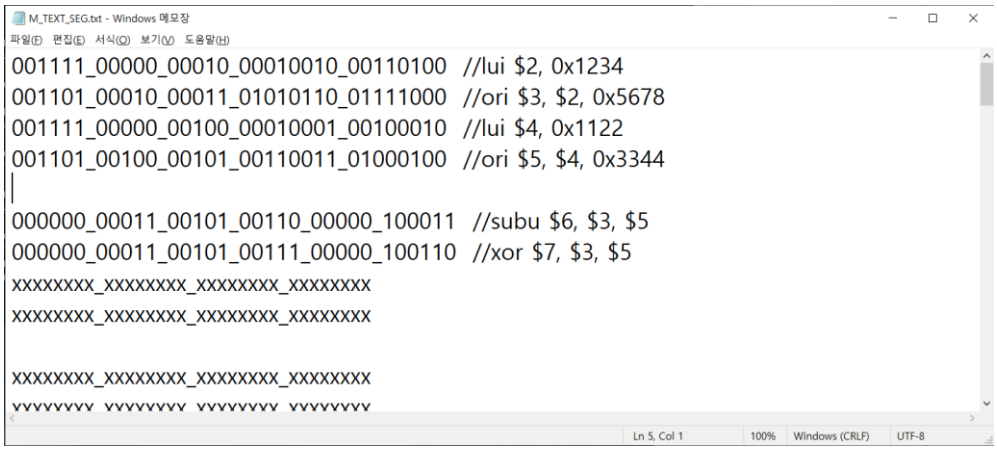


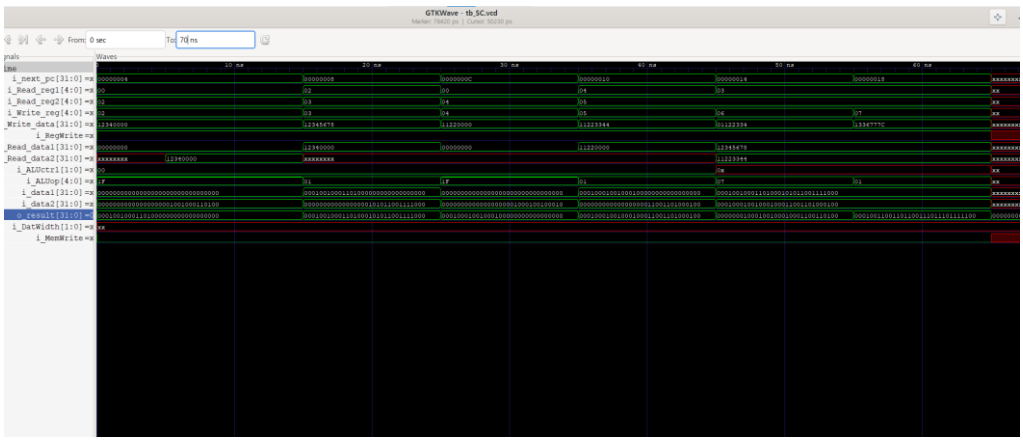
Figure 1 - The single cycle CPU datapath and control path

해당 명령어의 동작을 그림으로 표현하면 위와 같다 우선 IM에서 나온 Instruction은 op, $s, $t, $d, func으로 분리된다. Register File은 2개의 주소를 읽고 이를 쓰는 동작을 수행함으로 RegDst의 동작은 01이 수행된다. 또한 쓰는 동작을 수행함으로 RegWrite 동작 또한 1로 수행된다. RF를 나온 데이터 두 개를 연산하게 된다. 이로 인해 ARUsrcB는 00으로 동작한다. ALUctrl의 경우 해당 연산은 input의 위치를 바꾸지 않고 shift 또한 하지 않음으로 0x로 동작한다. 또한 immediate value를 쓰지 않음으로 EXTmode = x이다. ALUop는 Bitwise XOR 동작인 00011로 설정된다. 메모리는 사용되지 않음으로 MemWrite = 0, DatWidth = xxx로 설정되며, ALU 데이터를 register로 보내기 때문에 MemtoReg = 0이 된다. RegDatSel는 ALU/MEM의 데이터를 보내야 하기 때문에 00으로 설정된다. pc = pc+4로 동작해야 함으로 Branch = 000, jump동작은 하지 않기 때문에 jump = 00으로 동작한다.



해당 연산의 검증은 subu 연산 이후로 실행하여 검증한다. 해당 동작은 $7 register에 $3(0x12345678)과 $5(0x11223344)를 xor 연산을 수행한 값을 저장한다.

해당 결과는 다음과 같다.



해당 결과를 살펴보면 6번째 동작에서는 $3과 $5 레지스터의 값을 읽어와서 xor 연산을 한 값인 o_result가 정상적인 값을 출력하는 모습을 보인다. 그리고 해당 데이터가 RF로 다시 들어가 저장된다. pc의 값은 pc+4로 바뀌는 것 또한 정상적으로 동작하기 때문에 해당 명령어는 정상적으로 동작하는 것을 알 수 있다.

3. SRA

SRA의 동작은 $t register로부터 값을 불러와 해당 값을 shift amount의 크기만큼 ASR 연산을 수행한 뒤 이를 $d에 저장한다. 해당 명령은 R-type임으로 Opcode는 000000이며 function은 000011이다.

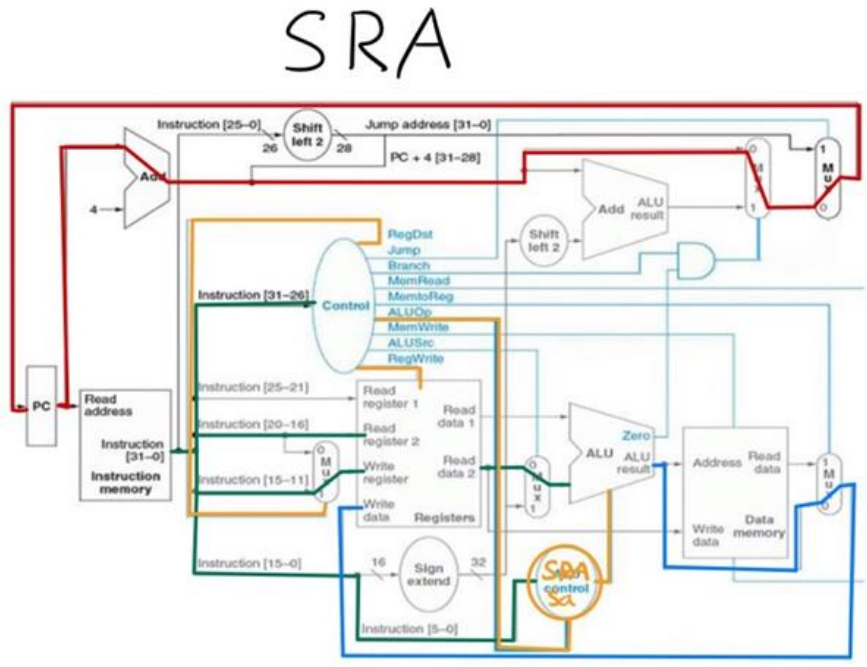
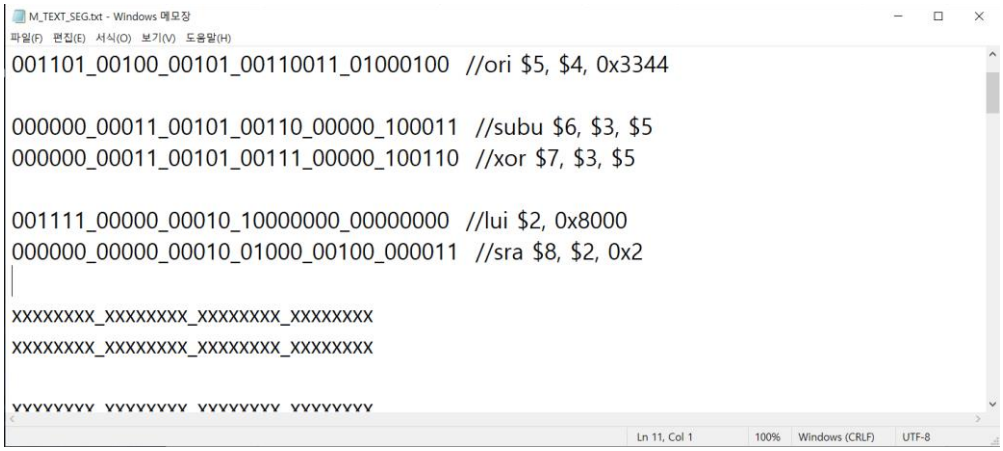


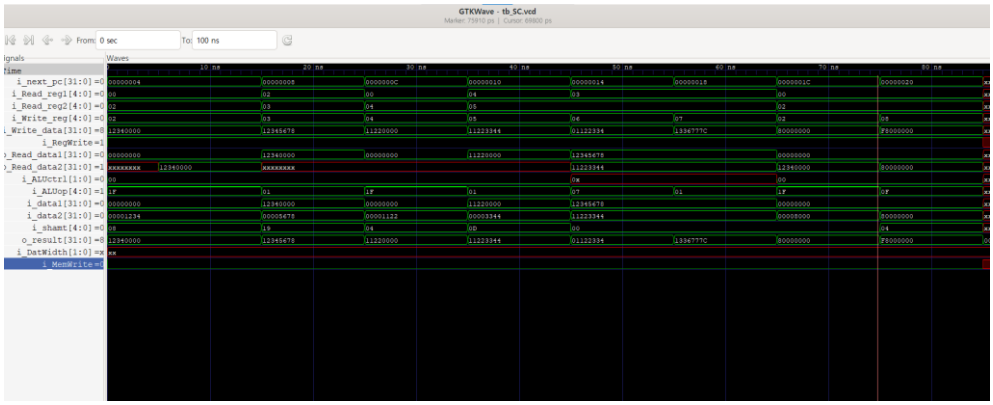
Figure 1 - The single cycle CPU datapath and control path

해당 명령어의 동작을 그림으로 표현하면 위와 같다 우선 IM에서 나온 Instruction

은 op, $t, $d, sa, func으로 분리된다. Register File은 $t 주소를 읽고 이를 $d 주소에 쓰는 동작을 수행함으로 RegDst의 동작은 01, RegWrite 동작은 1로 수행된다. RF를 나온 데이터 b를 연산하게 된다. 이로 인해 ARUsrcB는 00으로 동작한다. ALUctrl의 경우 해당 연산은 input의 위치를 바꾸지 않고 sa만큼 shift를 진행함으로 01로 동작한다. 또한 immediate value를 쓰지 않음으로 EXTmode = x이다. ALUop는 ASR (b>>>sa) 동작인 01111로 설정된다. 메모리는 사용되지 않음으로 MemWrite = 0, DatWidth = xxx로 설정되며, ALU 데이터를 register로 보내기 때문에 MemtoReg = 0이 된다. RegDatSel는 ALU/MEM의 데이터를 보내야 하기 때문에 00으로 설정된다. pc = pc+4로 동작해야 함으로 Branch = 000, jump동작은 하지 않기 때문에 jump = 00으로 동작한다.



해당 동작은 이전 테스트 이후로 덧붙어서 진행되었다. xor 이후에 lui를 통해 $2에 0x80000000을 넣어준다. 그 후 sra를 통해 0x80000000을 ASR 4(사진 상 오타)를 진행한 값을 $8에 넣어주는 것을 관찰한다. 해당 결과는 다음과 같다.



해당 동작을 보았을 때 $2에 0x80000000이 정상적으로 저장되어 있다. 이후에 sra를 진행하였을 때 $2에 있는 값이 $t 위치에 정상적으로 읽혔다. 그리고 해당 값의 MSB는 1이기 때문에 ASR 4를 진행하였을 때 상위 비트에 1이 정상적으로 추가되어 0xF8000000으로 값이 바뀌어 $d에 저장된다는 것을 볼 수 있다. 이를 통해 해당 동작이 정상적으로 이루어짐을 알 수 있다.

4. BNE

BNE의 동작은 $s, $t register로부터 값을 불러와 두 값이 같지 않을 경우 immediate value를 sign extension한 값을 pc값과 더하여 현재 실행하는 pc위치를 이동시킨다. 해당 명령은 i-type임으로 Opcode는 000101이며 function은 없다.

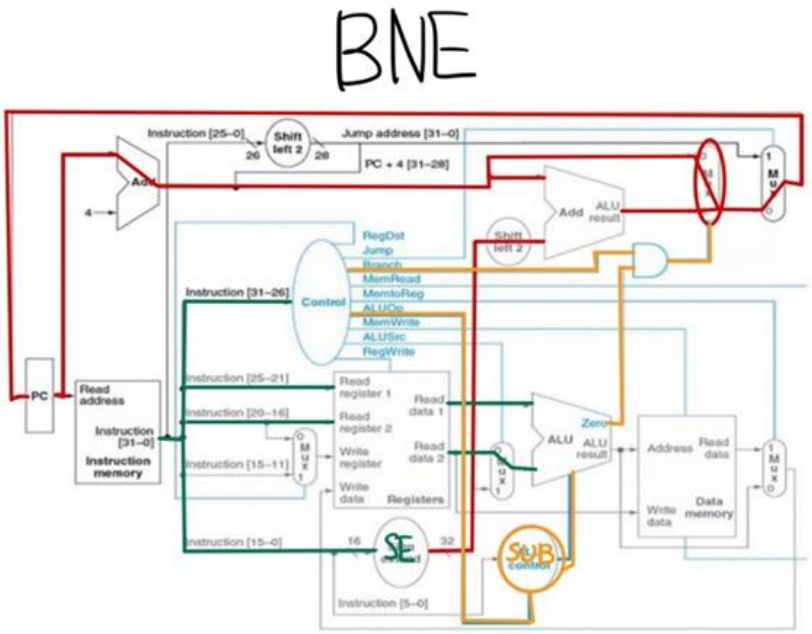
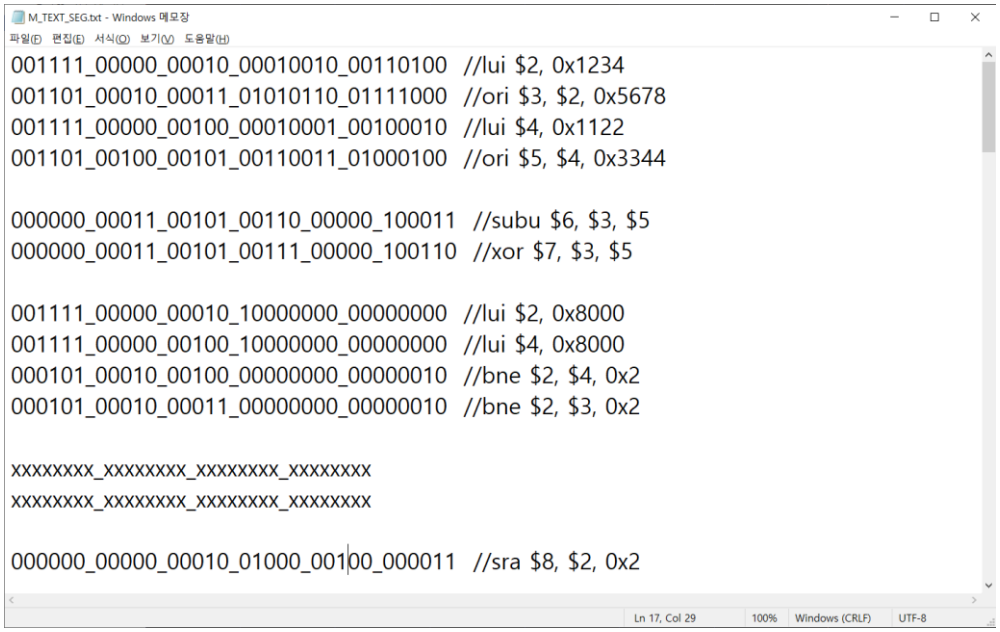
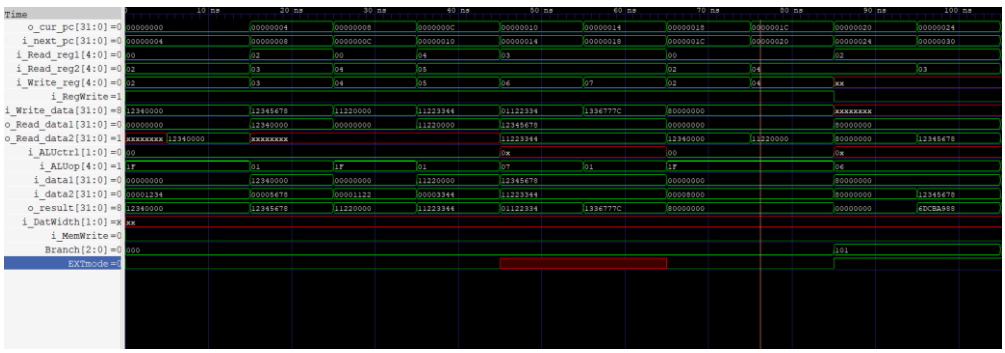


Figure 1 - The single cycle CPU datapath and control path

해당 명령어의 동작을 그림으로 표현하면 위와 같다 우선 IM에서 나온 Instruction은 op, $s, $t, i으로 분리된다. Register File은 $s, $t 주소를 읽는 동작만을 수행함으로 RegDst의 동작은 00, RegWrite 동작은 0로 수행된다. RF를 나온 데이터 a, b를 연산하게 된다. 이로 인해 ARUsrcB는 00으로 동작한다. ALUctrl의 경우 해당 연산은 input의 위치를 바꾸지 않고 shift를 진행하지 않음으로 0x로 동작한다. 또한 immediate value는 sign extension 임으로 EXTmode = 1이다. ALUop는 a-b 동작인 00110로 설정된다. 메모리는 사용되지 않음으로 MemWrite = 0, DatWidth = xxx로 설정되며, ALU 데이터를 register에 사용되지 않기 때문에 MemtoReg = x이 된다. RegDatSel는 사용되지 않기 때문에 xx으로 설정된다. pc는 ALU에서 비교한 값이 같지 않을 경우 branch를 해야 함으로 Branch = 101, jump 동작은 하지 않기 때문에 jump = 00으로 동작한다.



bne의 테스트는 다음과 같이 이루어진다. 이전에 sra 동작 이전에 lui 동작과 2번의 bne 동작, 그리고 x동작 2번을 추가한다. lui를 통해 $2와 $4에는 0x80000000이 저장된 상태며 $3에는 12345678이 저장된 상태다. 첫번째 bne에서는 두 값이 같음으로 branch되지 않아야 하며, 두번째 bne에서는 두 값이 다름으로 0x2만큼 주소를 추가하여 x로 채워진 두 값을 건너 뛴 sra로 이동해야 한다. 실행 결과는 다음과 같다.



bne의 결과를 살펴보았을 때 첫 번째 bne에서는 $2와 $4의 값이 alu를 통해 같음을 확인하여 branch에서 101 신호가 존재하여도 branch가 되지 않는 모습을 보인다. 하지만 두 번째 bne에서는 $2와 $3의 값이 alu를 통해 같지 않은 것이 branch로 전달되어 branch가 진행되어 주소가 4 + 8(2 << 2)만큼 이동하여 sra가 실행되는 것을 볼 수 있다. 이를 통해 해당 명령어가 정상적으로 설계되었음을 알 수 있다.

5. BLEZ

BLEZ의 동작은 $s, register로부터 값을 불러와 해당 값이 ($s <= 0)를 만족할 경우 immediate value를 sign extension한 값을 pc값과 더하여 현재 실행하는 pc위치를 이동시킨다. 해당 명령은 i-type임으로 Opcode는 000110이며 function은 없다.

BLEZ

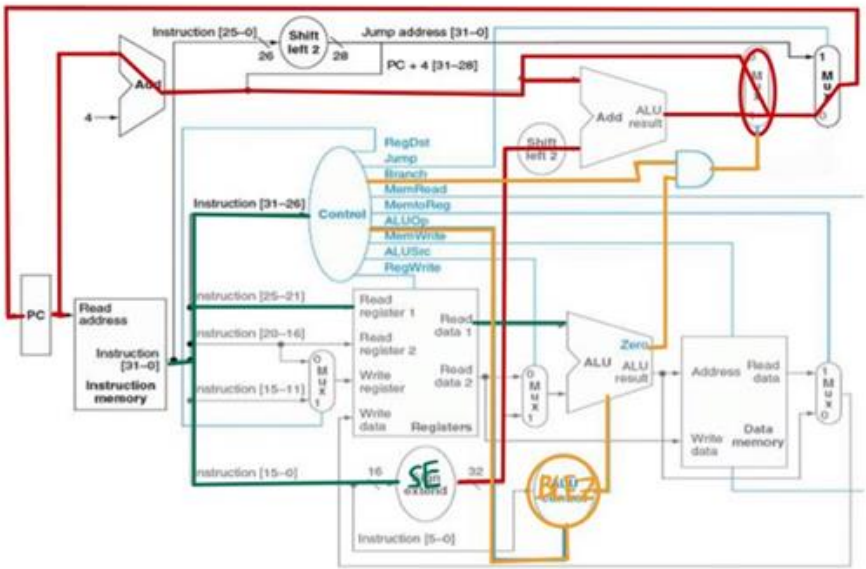
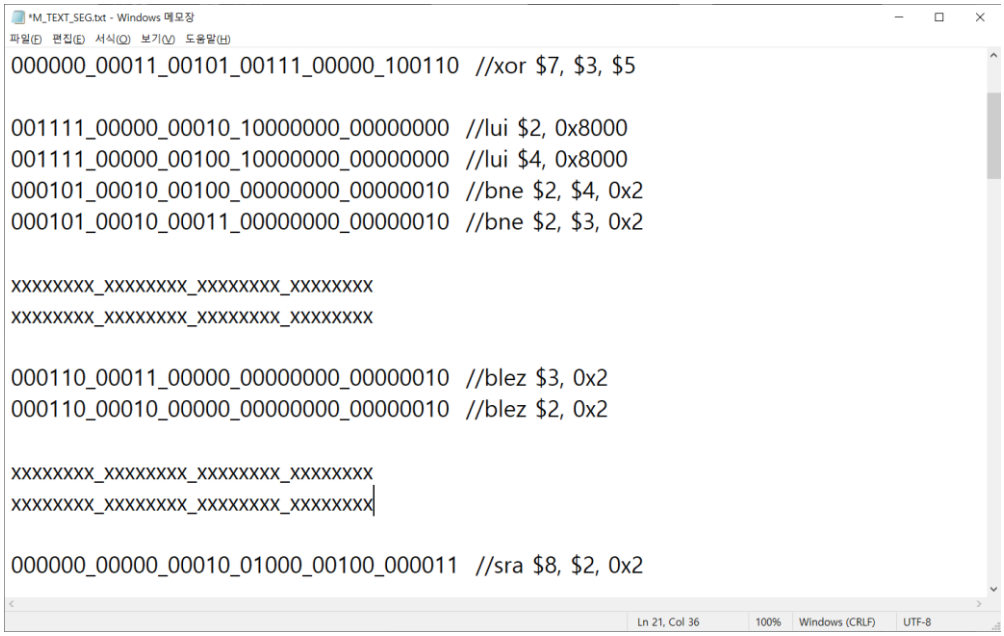
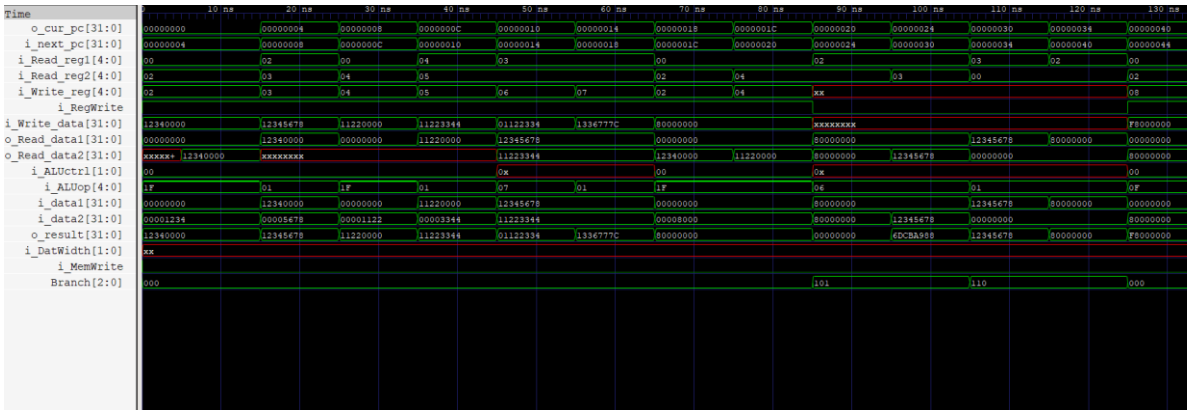


Figure 1 - The single cycle CPU datapath and control path

해당 명령어의 동작을 그림으로 표현하면 위와 같다 우선 IM에서 나온 Instruction 은 op, $s, i으로 분리된다. Register File은 $s 주소를 읽는 동작만을 수행함으로 RegDst의 동작은 00, RegWrite 동작은 0로 수행된다. RF를 나온 데이터 a와 0을 연산하게 된다. 이로 인해 ARUsrcB는 10으로 동작한다. ALUctrl의 경우 해당 연산은 input의 위치를 바꾸지 않고 shift를 진행하지 않음으로 0x로 동작한다. 또한 immediate value는 sign extension 임으로 EXTmode = 1이다. ALUop는 b가 0이기 때문에 빠른 연산을 위해 OR 동작인 00001로 설정된다. 메모리는 사용되지 않음으로 MemWrite = 0, DatWidth = xxx로 설정되며, ALU 데이터를 register에 사용되지 않기 때문에 MemtoReg = x이 된다. RegDatSel는 사용되지 않기 때문에 xx으로 설정된다. pc는 ALU에서 비교한 값이 음수이거나 0보다 작으면 branch를 해야 함으로 Branch = 110, jump 동작은 하지 않기 때문에 jump = 00으로 동작한다.



해당 명령어의 검증은 아까의 bne와 같이 2개의 case를 모두 실행하여 검증한다. 첫 번째 blez는 0보다 큰 값인 0x12345678이기 때문에 branch가 동작하지 않아야 한다. 하지만 두 번째 blez는 0보다 작은 값인 0x80000000이기 때문에 branch가 동작해야 한다. 결과는 다음과 같이 나왔다.



결과를 살펴보면 0x30에서의 실행에서는 0x12345678이 0보다 큼으로 pc값은 pc+4로 정상적으로 이동한 것을 확인할 수 있다. 하지만 0x34에서는 0x80000000은 0보다 작기 때문에 pc의 값이 $4 + 8(2 << 2)$만큼 이동하여 0x40번지가 실행되는 모습을 볼 수 있다. 이를 통해 blez가 정상적으로 구성되었음을 확인할 수 있다.

6. JALR

JALR의 동작은 현재 pc의 값을 $31에 저장하고 현재 pc값을 $s로 바꾸는 동작을 수행한다. 해당 명령은 jump 명령어지만 R-type으로 정의되어있다. 따라서 Opcode는 000000이며 function은 001000이다.

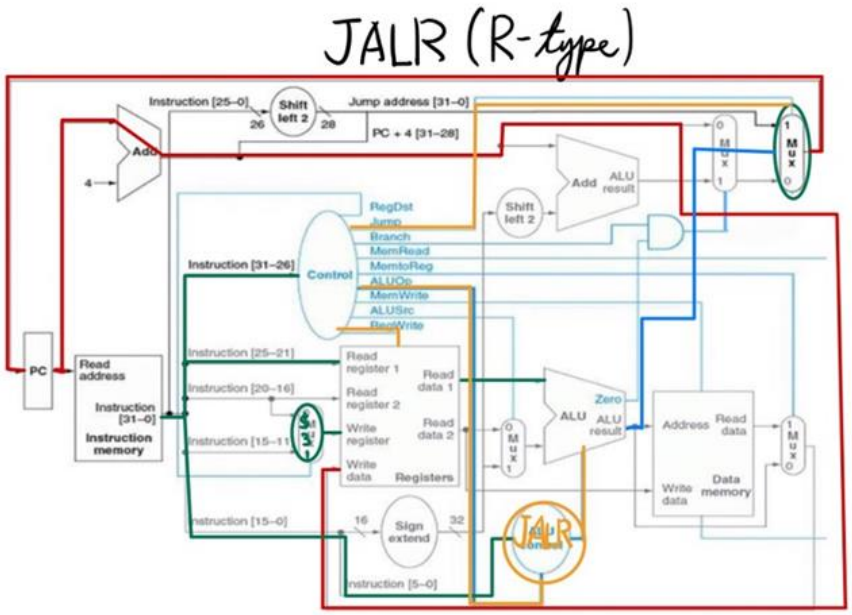
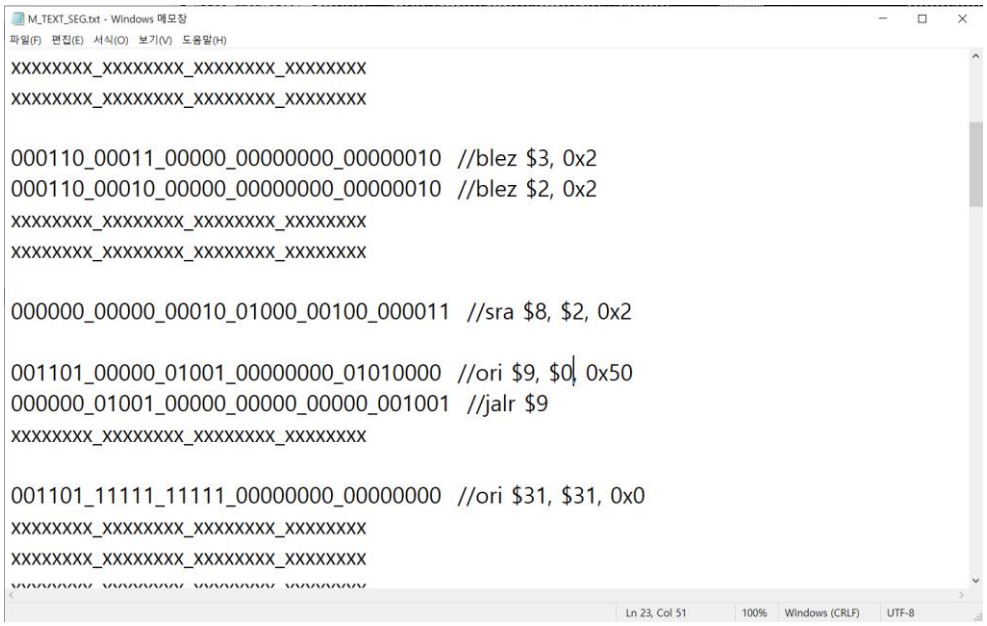
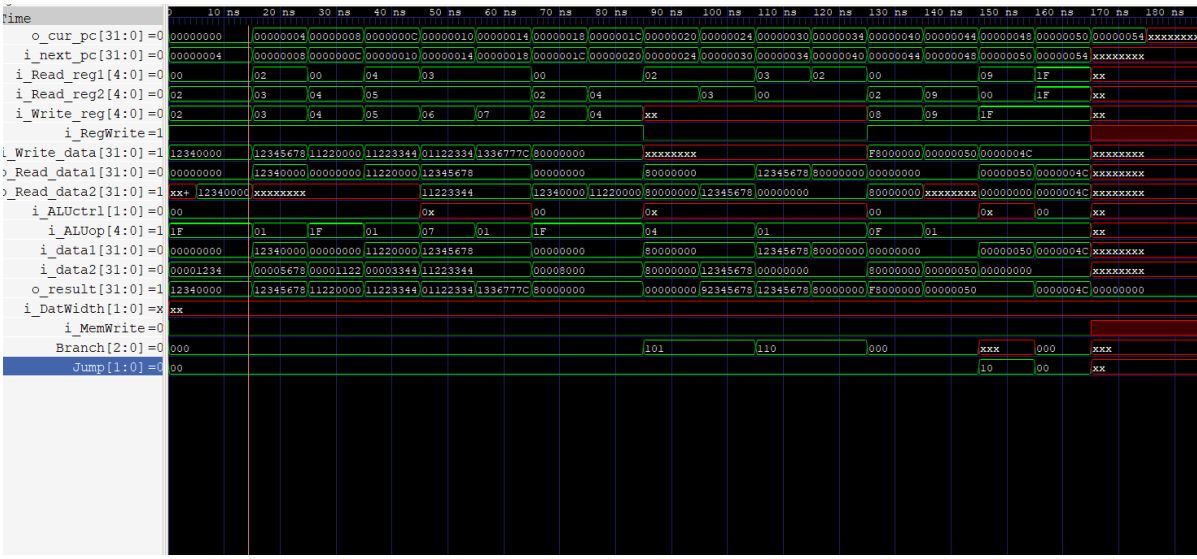


Figure 1 - The single cycle CPU datapath and control path

해당 명령어의 동작을 그림으로 표현하면 위와 같다 우선 IM에서 나온 Instruction은 op, $s, func으로 분리된다. Register File은 $s 주소를 읽고 $31주소를 쓰는 역할을 수행함으로 RegDst의 동작은 10, RegWrite 동작은 1로 수행된다. RF를 나온 데이터 a와 0을 연산하게 된다. 이로 인해 ARUsrcB는 10으로 동작한다. ALUctrl의 경우 해당 연산은 input의 위치를 바꾸지 않고 shift를 진행하지 않음으로 0x로 동작한다. 해당 연산은 immediate value를 사용하지 않기 때문에 EXTmode = x이다. ALUop는 b가 0이기 때문에 빠른 연산을 위해 OR 동작인 00001로 설정된다. 메모리는 사용되지 않음으로 MemWrite = 0, DatWidth = xxx로 설정되며, ALU 데이터를 register에 사용되지 않기에 MemtoReg = x이 된다. RegDatSel는 현재 pc값을 저장하기 때문에 11으로 설정된다. pc는 a값으로 jump하기에 Branch = xxx, jump = 10으로 동작한다.



해당 테스트의 동작은 sra를 마친 상황 이후로 진행된다. ori를 통해 $9에 0x50을 저장한 후 jalr $9를 통해 0x50주소(ori $31, $31, 0x0)위치로 이동한다. 그리고 해당 주소에서는 $31위치에 있는 데이터를 확인하는 과정을 수행한다. 이를 실행하였을 때 다음과 같은 결과를 볼 수 있었다.



해당 결과에서 0x44주소를 보았을 때 0x50의 값이 $9에 정상적으로 저장되어 있다. 이후의 과정에서 0x48을 보았을 때 현재 pc + 4의 값은 i_Write_data에 들어가 $31에 저장되고, 현재 주소는 jump 명령어를 통해 0x50위치로 이동하는 것을 볼 수 있다. 0x50 위치에서는 pc의 값을 출력하였을 때 정상적으로 당시 pc+4가 저장되는 것을 볼 수 있다. 이를 통해 jalr이 정상적으로 구성된 것을 확인할 수 있다.

7. ANDI

ANDI의 동작은 현재 $s register의 값을 불러와 Zero extension을 수행한 immediate value와 AND 연산을 수행한 후 $t에 저장한다. 해당 명령은 i-type으로 정의되어 있다. 따라서 Opcode는 001100이며 function은 없다.

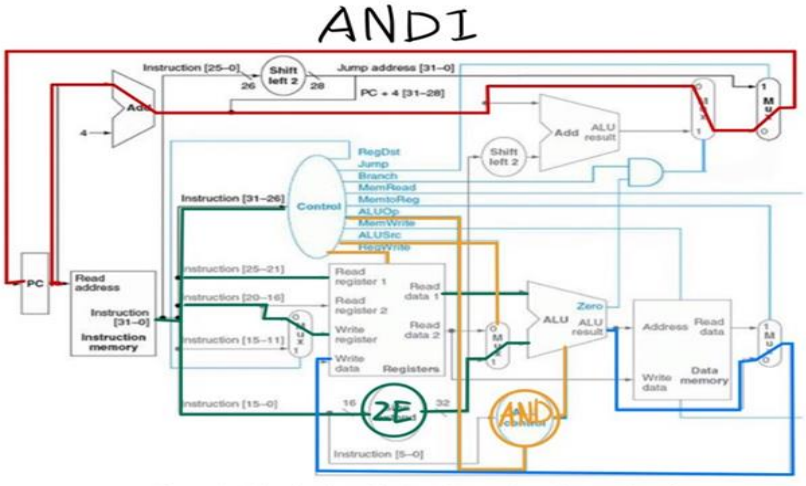
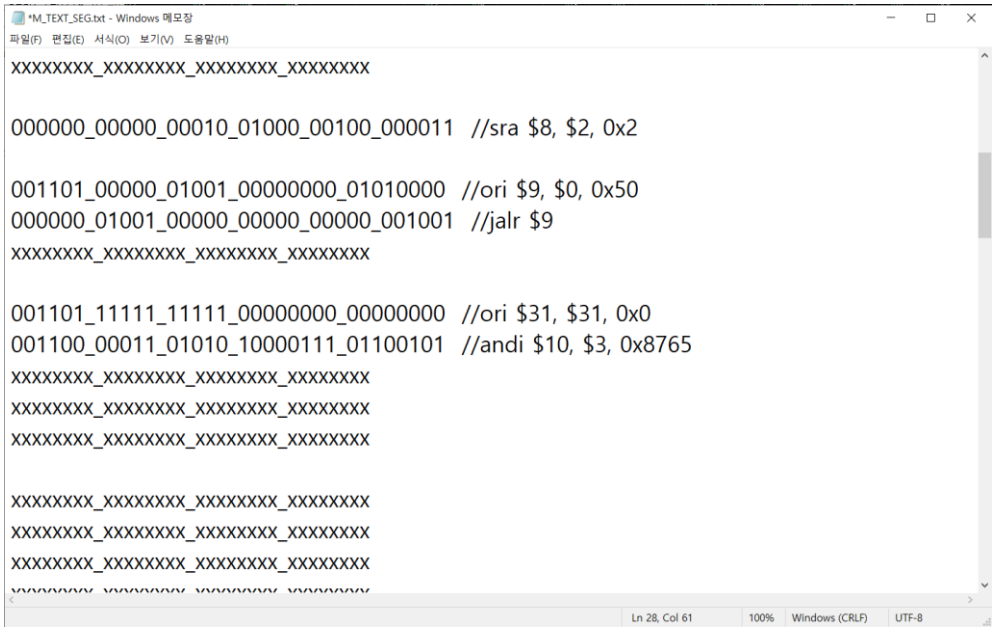
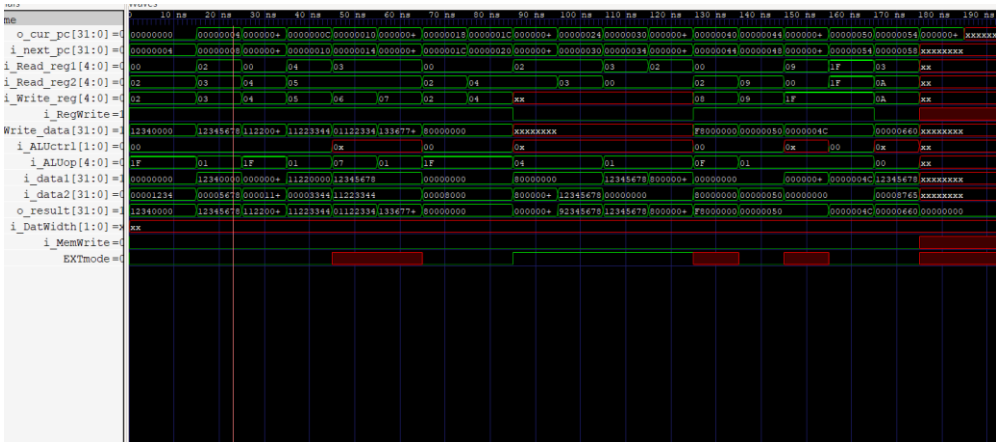


Figure 1 - The single cycle CPU datapath and control path

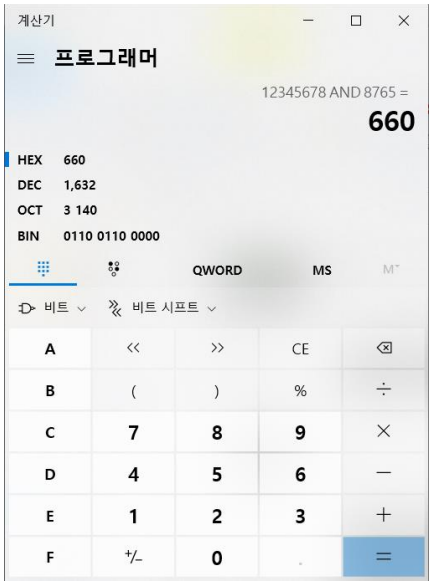
해당 명령어의 동작을 그림으로 표현하면 위와 같다 우선 IM에서 나온 Instruction은 op, $s,$t, i으로 분리된다. Register File은 $s 주소를 읽고 $t주소를 쓰는 역할을 수행함으로 RegDst의 동작은 00, RegWrite 동작은 1로 수행된다. RF를 나온 데이터 a는 immediate value와 연산하게 된다. 이로 인해 ARUsrcB는 01으로 동작한다. ALUctrl의 경우 해당 연산은 input의 위치를 바꾸지 않고 shift를 진행하지 않음으로 0x로 동작한다. 해당 연산에서 immediate value는 Zero extension을 사용하기 때문에 EXTmode = 0이다. ALUop는 Bitwise AND 연산인 00000로 설정된다. 메모리는 사용되지 않음으로 MemWrite = 0, DatWidth = xxx로 설정되며, ALU 데이터를 register에 저장하기 때문에 MemtoReg = 0이 된다. RegDatSel는 ALU/MEM 값을 RF에 저장하기 때문에 00으로 설정된다. pc값은 pc+4로 저장되기 때문에 Branch = 000, jump = 00으로 동작한다.



해당 명령어의 검증은 이전 검증 이후에서 이루어진다. andi $10, $3, 0x8765를 통해 0x12345678과 0x8765를 and 연산한 결과를 $10에 저장하는 것을 확인한다. 결과는 다음과 같이 나타났다.



해당 결과를 확인하였을 때 $3의 값 0x12345678은 정상적으로 읽혔고 immediate value 0x8765 또한 zero extension을 통해 정상적으로 읽어왔다. 두 값의 AND 연산은 아래 그림을 통해 0x660으로 정상적으로 들어온 것을 확인할 수 있다.



이후 해당 값은 $10번지에 들어가서 쓰이는 것 또한 확인할 수 있어 해당 명령어가 정상적으로 작동하는 것을 검증하였다.

8. SLTIU

SLTIU의 동작은 현재 $s register의 값을 불러와 Zero extension을 수행한 immediate value보다 작은 지 확인한 후 해당 상태를 $t에 저장한다. 해당 명령은 i-type으로 정의되어 있다. 따라서 Opcode는 001011이며 function은 없다.

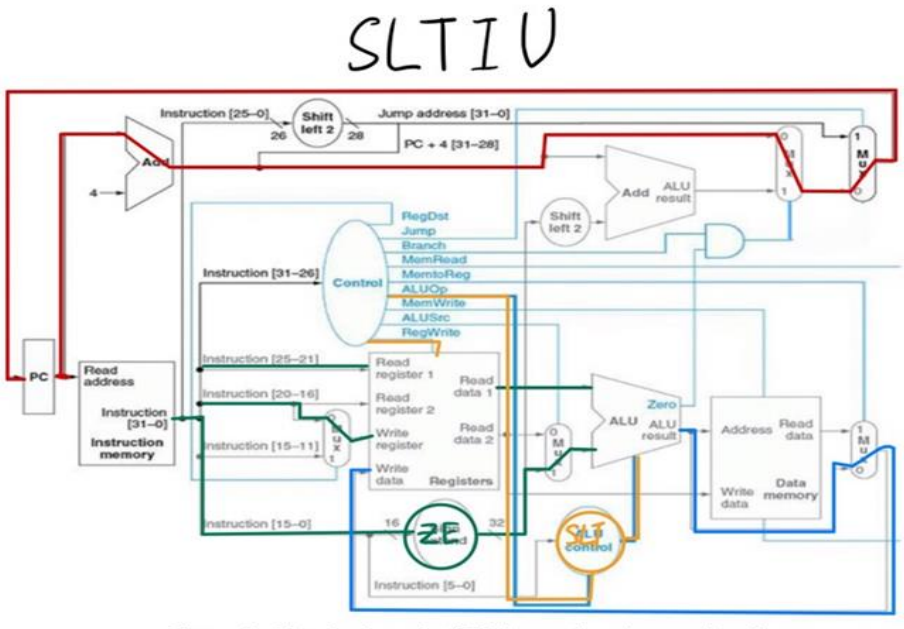
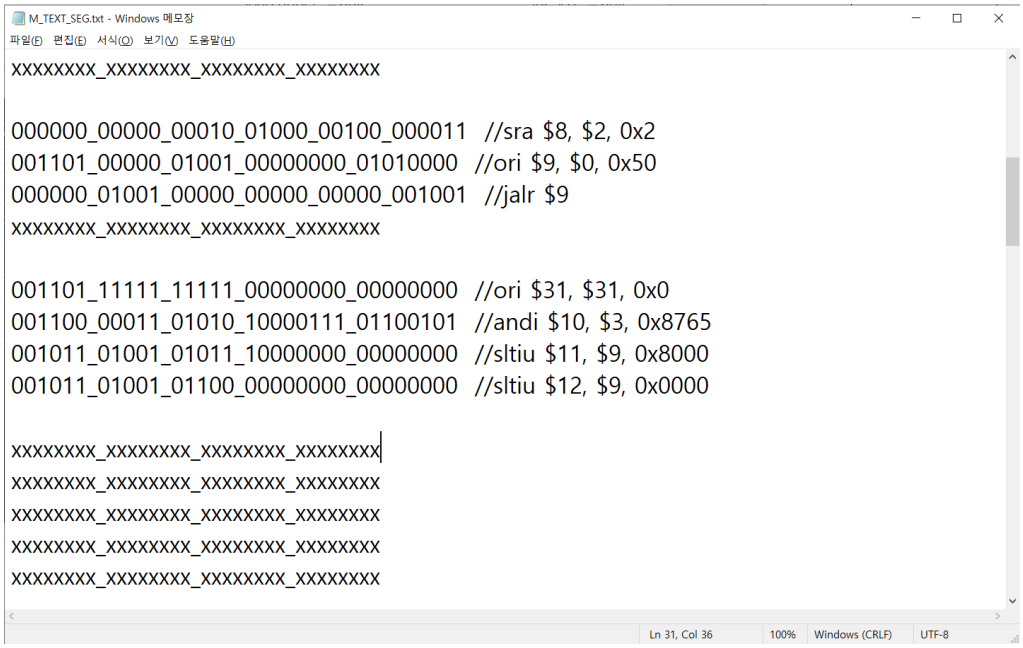
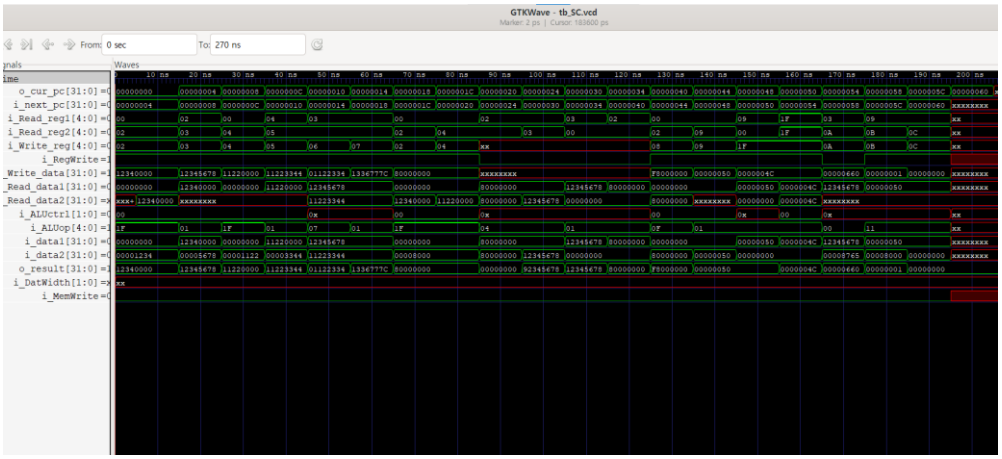


Figure 1 - The single cycle CPU datapath and control path

해당 명령어의 동작을 그림으로 표현하면 위와 같다 우선 IM에서 나온 Instruction은 op, $s,$t, i으로 분리된다. Register File은 $s 주소를 읽고 $t주소를 쓰는 역할을 수행함으로 RegDst의 동작은 00, RegWrite 동작은 1로 수행된다. RF를 나온 데이터 a는 immediate value와 연산하게 된다. 이로 인해 ARUsrcB는 01으로 동작한다. ALUctrl의 경우 해당 연산은 input의 위치를 바꾸지 않고 shift를 진행하지 않음으로 0x로 동작한다. 해당 연산에서 immediate value는 Zero extension을 사용하기 때문에 EXTmode = 0이다. ALUop는 Unsigned SLT 연산인 10001로 설정된다. 메모리는 사용되지 않음으로 MemWrite = 0, DatWidth = xxx로 설정되며, ALU 데이터를 register에 저장하기 때문에 MemtoReg = 0이 된다. RegDatSel는 ALU/MEM 값을 RF에 저장하기 때문에 00으로 설정된다. pc값은 pc+4로 저장되기 때문에 Branch = 000, jump = 00으로 동작한다.



해당 명령어의 검증은 이전 테스트에 이어 진행된다. 두 번의 sltiu를 통해 $9(0x50)를 0x8000, 0x0000과 비교하여 해당 상태를 $11, $12에 저장한다. $11은 (unsigned)(0x50 < 0x8000) == true이기 때문에 0x1이 저장되고, $12는 (unsigned)(0x50 < 0x0000) == false이기 때문에 0x0이 저장되어야 한다. 해당 결과는 다음과 같이 동작하였다.



해당 결과를 살펴보면 첫 번째 비교의 결과는 0x1로 정상적으로 출력되어 $11에 저장되었고, $12의 비교 결과는 0x0으로 예측한 결과와 동일하게 출력되어 $12에 저장된 것을 볼 수 있다. 이를 통해 해당 명령어가 정상적으로 만들어진 것을 확인할 수 있다.

9. SB

SB의 동작은 현재 $s, $t register의 값을 불러온다. sign extension을 수행한 immediate value는 $s와 덧셈 연산을 수행하여 해당 메모리 위치에 $t의 값을 저장한다. 해당 명령은 i-type으로 정의되어 있다. 따라서 Opcode는 101000이며 function은 없다.

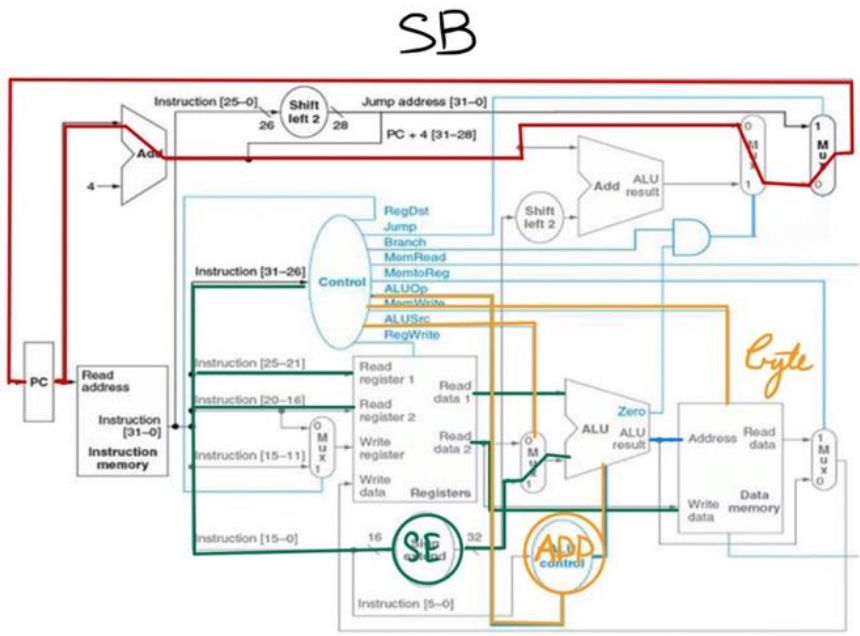
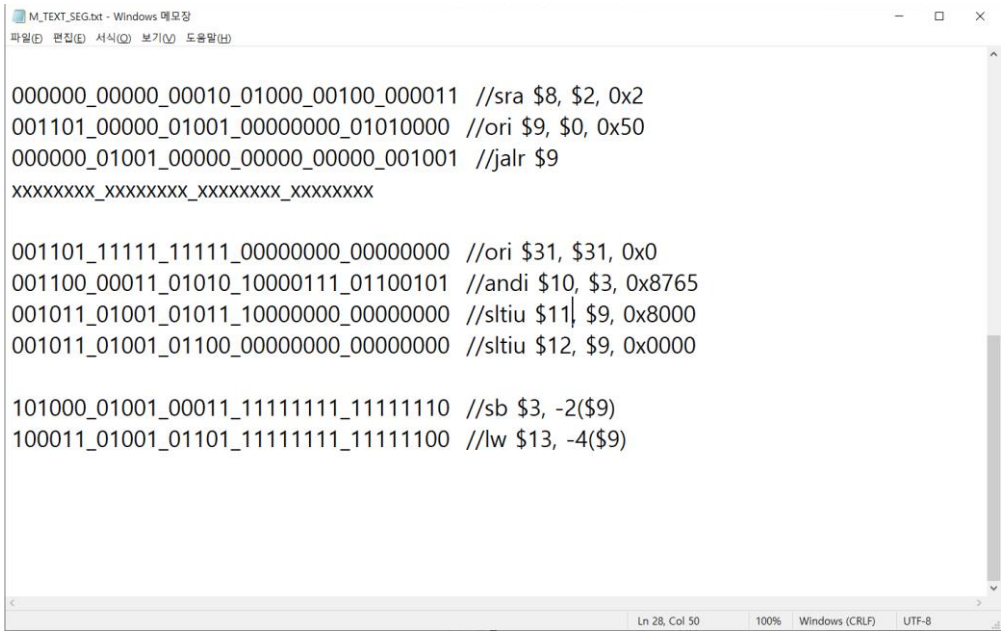


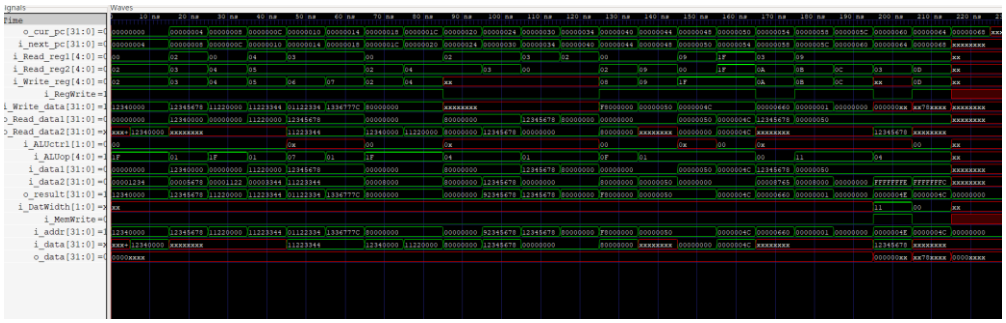
Figure 1 - The single cycle CPU datapath and control path

해당 명령어의 동작을 그림으로 표현하면 위와 같다 우선 IM에서 나온 Instruction은 op, $s,$t, i으로 분리된다. Register File은 $s, $t주소를 읽는 역할만 수행함으로

RegDst의 동작은 00, RegWrite 동작은 0로 수행된다. RF를 나온 데이터 a는 immediate value와 연산하게 된다. 이로 인해 ARUsrcB는 01으로 동작한다. ALUctrl의 경우 해당 연산은 input의 위치를 바꾸지 않고 shift를 진행하지 않음으로 0x로 동작한다. 해당 연산에서 immediate value는 sign extension을 사용하기 때문에 EXTmode = 1이다. ALUop는 a+b 연산인 00100로 설정된다. 메모리에 byte 단위로 저장되기 때문에 MemWrite = 1, DatWidth = 011로 설정되며, ALU 데이터를 register에 저장할 필요가 없기 때문에 MemtoReg = x이 된다. RegDatSel는 RF에서 쓰는 동작이 없기 때문에 xx으로 설정된다. pc값은 pc+4로 저장되기 때문에 Branch = 000, jump = 00으로 동작한다.



해당 명령어를 검증은 이전 검증에 이어서 진행된다. sb $3, -2($9)를 통해 $3(0x12345678)의 데이터 1byte(0x78)를 $9(0x50) – 2 위치에 저장한다. 그리고 lw를 통해 저장된 0x4c위치에서 word로 데이터를 읽어와 해당 데이터가 정상적으로 저장되었는지 $13에서 확인한다. 만약 정상적으로 저장된 경우 xx78xxxx가 $13에 표시되어야 한다. 결과는 다음과 같이 출력되었다.



해당 결과를 살펴보면 0x60 위치에서 $3의 0x12345678의 값이 읽혔고 쓰는 주소로는 0x50 – 2인 0x4E로 정상적으로 설정되었다. i_datWidth는 11로 쓰여져있는 것 또한 확인할 수 있다. 다음 pc에서 이를 확인하였을 때 0x4C에서 0x4F까지 읽은

데이터가 xx78xxxx로 0x12345678의 LSB 1byte만 저장되고 저장된 위치도 0x4E로 정상적인 것을 확인할 수 있다. 이를 통해 해당 명령어가 정상적으로 구동하는 것을 알 수 있다.

10.LBU

LBU의 동작은 현재 $s register의 값을 불러온다. sign extension을 수행한 immediate value는 $s와 덧셈 연산을 수행하여 해당 메모리 위치에 있는 값을 읽어와 $t에 저장한다. 해당 명령은 i-type으로 정의되어 있다. 따라서 Opcode는 100100이며 function은 없다.

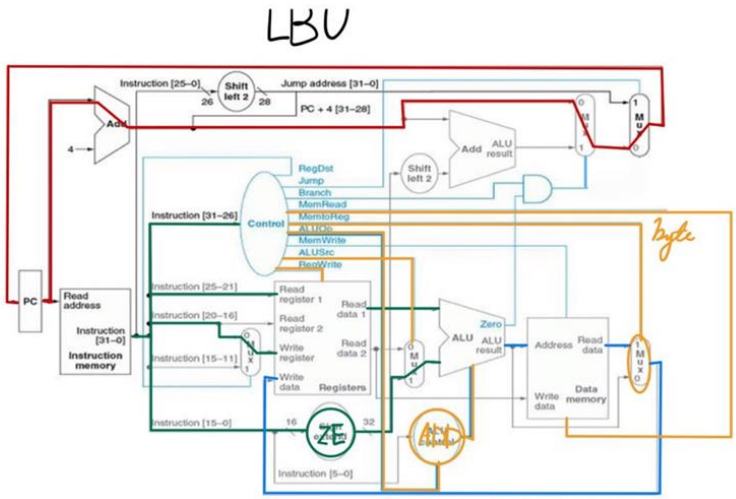
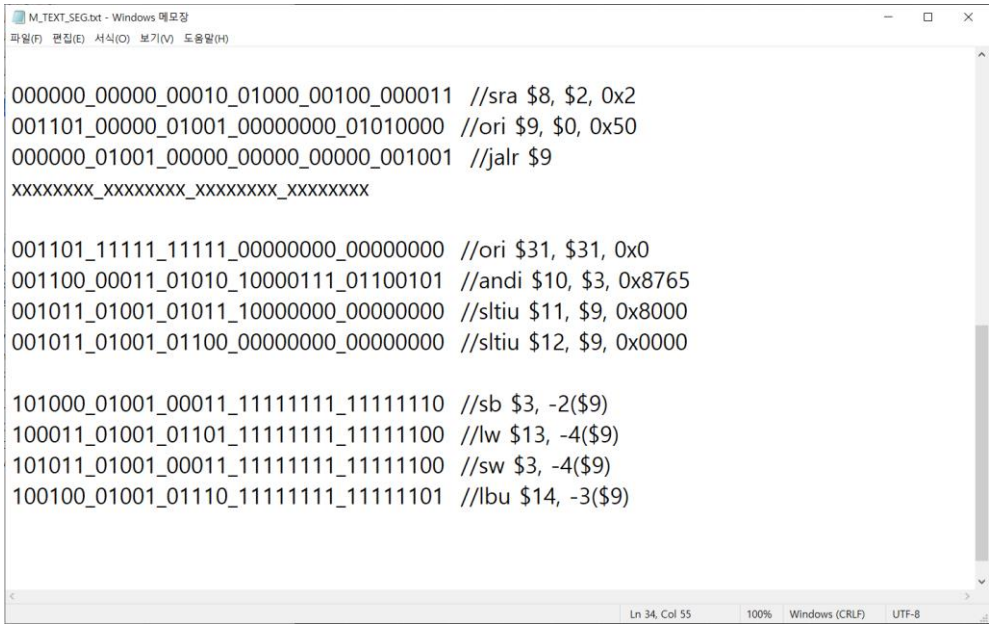
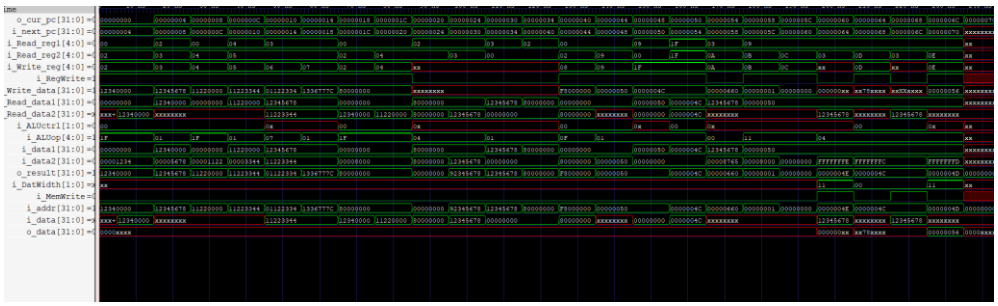


Figure 1 - The single cycle CPU datapath and control path

해당 명령어의 동작을 그림으로 표현하면 위와 같다 우선 IM에서 나온 Instruction은 op, $s,$t, i으로 분리된다. Register File은 $s, 주소를 읽고 $t 주소를 쓰는 역할을 수행함으로 RegDst의 동작은 00, RegWrite 동작은 1로 수행된다. RF를 나온 데이터 a는 immediate value와 연산하게 된다. 이로 인해 ARUsrcB는 01으로 동작한다. ALUctrl의 경우 해당 연산은 input의 위치를 바꾸지 않고 shift를 진행하지 않음으로 0x로 동작한다. 해당 연산에서 immediate value는 sign extension을 사용하기 때문에 EXTmode = 1이다. ALUop는 a+b 연산인 00100로 설정된다. 메모리에 byte 단위로 읽기 때문에 MemWrite = 0, DatWidth = 011로 설정되며, Memory 데이터를 register에 저장하기 때문에 MemtoReg = 1이 된다. RegDatSel는 ALU/MEM 데이터가 RF에 쓰이기 때문에 00으로 설정된다. pc값은 pc+4로 저장되기 때문에 Branch = 000, jump = 00으로 동작한다.



해당 명령어의 검증은 이전 테스트 이후로 진행된다. sw $3, -4($9)를 통해 $9(0x50) – 4 주소에 $3(0x12345678)을 저장한다. 그 후 lbu $14, -3($9)를 사용하여 $14에 $9(0x50) – 3 메모리 주소에서 값 0x56을 가져와서 저장한다. 이를 통해 해당 명령어를 검증할 수 있다. 검증한 결과는 다음과 같다.



해당 결과를 살펴보면 pc 0x68에서 12345678이 0x4C(0x50 - 4)위치에 정상적으로 저장되어 있다. 그리고 lbu를 통해 데이터를 불러왔을 때 0x4E에 해당하는 값 0x56 1byte가 Zero extension을 통해 불러와 $13에 저장된 것을 확인할 수 있다. 이를 통해 해당 명령어가 정상적으로 작동하는 것을 알 수 있다.

C. Consideration

1. Consideration

- 해당 과제를 통해 기본적으로 R-Type, I-Type, J-Type 명령어들의 동작의 차이에 대해 구분할 수 있는 과제였다. 또한 각 명령어가 들어왔을 때 이를 어떻게 구분하고 어떠한 module을 동작 시킬 것인지 고민하는 과정을 통해 MIPS Architecture에 대한 이해를 높일 수 있었다. 또한 이를 MIPS Assembly Language로 변환하여 코드를 작성하면서 MIPS 코드를 작성하고 검증할 수 있는 기회였다.

2. Problem & Solution

- 해당 프로젝트에서는 lw와 sw가 동작하였을 때 Shift Left 2를 진행하지 않고 해당 주소에서 그대로 데이터를 가져오는 것을 확인하였다. 기본적으로 load word의 경우 4byte로 데이터를 읽어 오기 때문에 해당 프로젝트에서는 address의 LSB가 00으로 고정되어 있다. 만약 해당 값이 들어오기 전에 Shift Left 2를 진행한다면 기존보다 4배 많은 메모리에 접근할 수 있을 것이다(기존 메모리 주소 : 2^32 = 4G, Shift 진행 메모리 주소 : 2^34 = 16G).

D. Reference

- 강의자료만을 참고