Computer Architecture

Assignment #2

담당 교수님 : 이성원 교수님

컴퓨터정보공학부

2018202013

정영민

■ 문제의 해석 및 해결 방향

● 각 명령어의 기능과 동작

- **BGE**

Instruction	Operation
BGE rs, rt, label	If (rs >= rt) PC += immediate << 2

BGE는 i-type으로 입력 받는다. BGE의 opcode는 000110로 지정해 두었다. 입력 형식은 BGE rs, rt, label의 형태이며 rs register의 값이 rt register의 값보다 크거나 같을 시 pc에 immediate값을 shift left 2bit한 값을 더하게 된다. ALUSrcA는 rs를 가리키며 ALUSrcB는 rt를 가리킨다. ALUOp는 SLT를 하며 ALU의 zero flag를 사용하여 branch를 한다.

- **ANDI**

Instruction	Operation
ANDI rs, rt, immediate	Rt = rs & ZeroExtenstion(immediate)

ANDI는 i-type으로 입력 받는다. ANDI의 opcode는 001100이며 입력 형식은 ANDI rs rt immediate로 입력 받는다. Rs register의 값과 zero extension을 한 immediate값을 and연산하여 나온 결과값을 저장하게 된다. ALUSrcA는 rs를 가리키며 ALUSrcB는 extended immediate를 가리킨다. ALUOp는 AND를 하게 된다.

- **LW**

Instruction	Operation
LW base, rt, offset	Rt <- memory[base+offset]

LW의 opcode는 100011이며 입력 형식은 LW base rt offset으로 입력 받는다. Rt register에 memory의 base + offset 주소 값에 있는 값을 저장하게 된다. Mem_Adr state에서 메모리의 주소를 계산한다. 이후 Mem_Read에서 ALU의 결과 값을 이용하여 메모리의 값을 읽어온다. Mem_Writeback에서 메모리의 값을 레지스터에 저장하고 종료하게 된다. RegDst로 rt를 가리키며 RegWrite를 Enable시켜야 하며 MemtoReg를 1로 해야 한다. memRead는 0으로 초기화한다.

- **LWAI**

Instruction	Operation
LWAI rs, rt, rd	Rd <- memory[rs + rt] Rt = rt + 4

LWAI의 opcode는 100111로 지정하였다. 입력 형식은 LWAI rs rt rd로 입력 받는다. Rd register에 memory의 rs+rt 주소 값에 있는 값을 저장하고 이후에 rt register의 값을 4 증가시킨다. 이 명령어는 lw의 state변화와 유사하게 이동하지만 차이점은 주소 값 계산에서 레지스터 두개를 사용하는 것과 rt의 값을 증가시키는 것이다.

- **SLTI**

Instruction	Operation
SLTI rs, rt, immediate	Rt <- (rs < immediate)

SLTI의 opcode는 001010이다. 입력 형식은 SLTI rs, rt, immediate로 입력 받는다. Rs register의 값이 immediate 값보다 작을 시 Rt register에 1을 저장하며 그렇지 않을 경우 0을 저장한다. ALUSrcA에서 rs를 가리키고 ALUSrcB에서 extended immediate값을 가리킨다. ALUOp는 slt를 하게 된다.

- **JAL**

Instruction	Operation
Jal label	$31 = PC+4 PC += immediate << 2

JAL의 opcode는 000011이다. 입력 형식은 JAL Label로 입력 받는다. 31번 register에 pc값 + 4를 한 값을 저장하고, PC값에 immediate값을 Shift left 2 bit한 값을 더한다. JAL state에서 먼저 ALUSrcA를 pc로 지정하고 ALUSrcB를 4로 지정한 후에 ALUOp를 add로 지정한다. 다음 state를 JAL_Writeback으로 하여 해당 state에서 31번 register에 연산 된 ALU 결과를 저장한다. 그리고 ALUSrcB의 값을 바꾼다. 그 후 JAL_Adr state에서 RegWrite를 0으로 바꾸어 이전에 바꾼 ALUSrcB의 값으로 연산 된 값이 31번 register에 쓰이지 않게 하였다. 그 후에 Jump state로 이동하여 연산 된 주소 값으로 Jump하게 된다.

- **DIV4**

Instruction	Operation
DIV4 rs, rt	Rt = rs / 4

DIV4의 opcode는 001110으로 지정하였다. 입력 형식은 DIV4 rs rt로 입력 받는다. Rt register에 rs register 값을 4로 나눈 값을 저장한다. ALUSrc에서 rs와 rt를 가리키고 ALUOp는 Division 혹은 Shift right를 해야한다.

- **SLLV**

Instruction	Operation
SLLV rs, rt, rd	Rd <- rt << rs

SLLV는 R-type이기 때문에 opcode는 0이며 function bit가 000100이다. 입력 형식은 SLLV rs rt rd이며 rd register에 rt register의 값을 rs register의 값 만큼 shift left한 값을 저장한다. ALUSrcA에서 rs를 가리키고 ALUSrcB에서 rt 값을 가리킨다. ALUOp는 function bit에 따라 다른 동작을 하게 된다.

- **BLTZ**

Instruction	Operation
BTLZ rs offset	If (rs<0) then branch

BLTZ의 opcode는 000001이다. 입력 형식은 BLTZ rs offset으로 받으며 rs register의 값이 0보다 작을 때 PC값에 offset값을 더한다. ALUSrcA에서 rs를 가리키고 ALUSrcB에서 rt값을 가리킨다. ALUOp는 sub를 하게 되고 해당 연산의 flag에 따라 Branch 동작을 하게 된다.

- **XOR**

Instruction	Operation
XOR rs rt rd	Rd <- rs ^ rt

XOR의 opcode는 000000이며 function bit가 100110이다. 입력 형식은 XOR rs rt rd로 받으며 rd register에 rs register과 rt register을 XOR연산한 값을 저장한다. ALUSrcA에서 rs를 가리키고 ALUSrcB에서 rt값을 가리킨다. ALUOp는 function bit에 따라 다른 동작을 하게 된다.

- **문제점 해결 방향**

  - **BGE**

    BGE 명령어는 기존 BEQ 명령어를 활용하였다. Opcode의 경우 000110을 사용하였다. 이 프로젝트는 zero flag가 ALU의 연산 결과가 0이거나 음수일 때 들어오게 된다. 따라서 BGE를 구현하기 위해서 Rs와 rt register을 slt연산을 하였다. 만약 rs가 rt보다 작다면 1이 되고 rs가 rt보다 크거나 같다면 0이 되기 때문에 BGE 조건과 zero flag의 조건에 맞게 된다. 따라서 해당 ALU의 zero flag를 통하여 Branch동작을 하도록 하였다.

  - **ANDI**

    ANDI는 기존 ADDI 명령어에서 Op동작만 바꾼 것이다. Rs와 Extended Immediate 값을 지정하여 and연산한 결과 값을 ADDI_Writeback state에서 rt에 저장하도록 하였다.

  - **LW & LWAI**

    LW의 경우 기존 구현되어 있던 그대로를 사용하였다. LWAI의 경우 구현되어 있던 LW를 활용하였는데, LW와 다른 점은 immediate값을 사용하지 않고 register 값을 사용했다는 점과 rt register의 값을 4만큼 증가시킨다는 것이다. Rt register의 값을 증가시키는 것을 제외한 다른 것은 LW의 방법에서 Destination만 바꾸어 구현하였다. LWAI의 경우 opcode로 100111을 사용하였다.

  - **STLI**

    SLTI는 마찬가지로 기존 ADDI 명령어에서 Op동작만 바꾸어 구현하였다.

  - **JAL**

    JAL 명령어의 경우 현재 PC값에 4를 더한 값을 31번 레지스터에 저장하고, 현재 PC값에 target의 값을 shift left 2bit한 값을 더하여 다시 PC에 넣는 동작을 하게 된다. 우선 먼저 현재 PC값에 4를 더한 값을 31번 레지스터에 저

장하였고, 다음 target과 PC값의 연산을 하도록 하였다. JAL을 통하여 PC+4를 계산하였고 JAL_Writeback을 통하여 31번 레지스터에 값을 저장하고 target과 PC값의 연산을 하였다. 이후 JAL_Adr에서 RegWrite를 0으로 설정하여 31번 레지스터의 값이 바뀌는 것을 막았고, 이후 Jump state를 통하여 연산 된 주소 값으로 이동하도록 하였다.

- **DIV4**

  DIV4 명령어는 opcode를 001110로 두어 구현하였다. DIV의 경우 ALUSrcA에서 rs를 선택하고 ALUSrcB에서 4를 선택한 다음 ALUOp로 Div이나 SR을 사용하면 되지만 이번 과제는 해당 동작이 구현되어 있지 않다. 따라서 임의로 다른 Op값을 입력하였으며 이 명령어가 제대로 동작할 수 있도록 하기 위해선 ALU control에서 Division의 기능을 추가하면 된다.

- **SLLV**

  SLLV 명령어는 opcode가 0일 때 function bit를 통하여 구현하였다. R-type이기 때문에 타겟을 rs, rt register로 지정하였고, ALUOp는 R-Type으로 지정하였다. 이후 ALU_Writeback을 통하여 rd register에 값을 저장하였다.

- **XOR**

  XOR 명령어는 opcode가 0일 때 function bit를 통하여 구현하였다. 마찬가지로 R-type이기 때문에 타겟을 rs, rt register로 지정하였고, ALUOp는 R-Type으로 지정하였다. 이후 ALU_Writeback을 통하여 rd register에 값을 저장하였다.

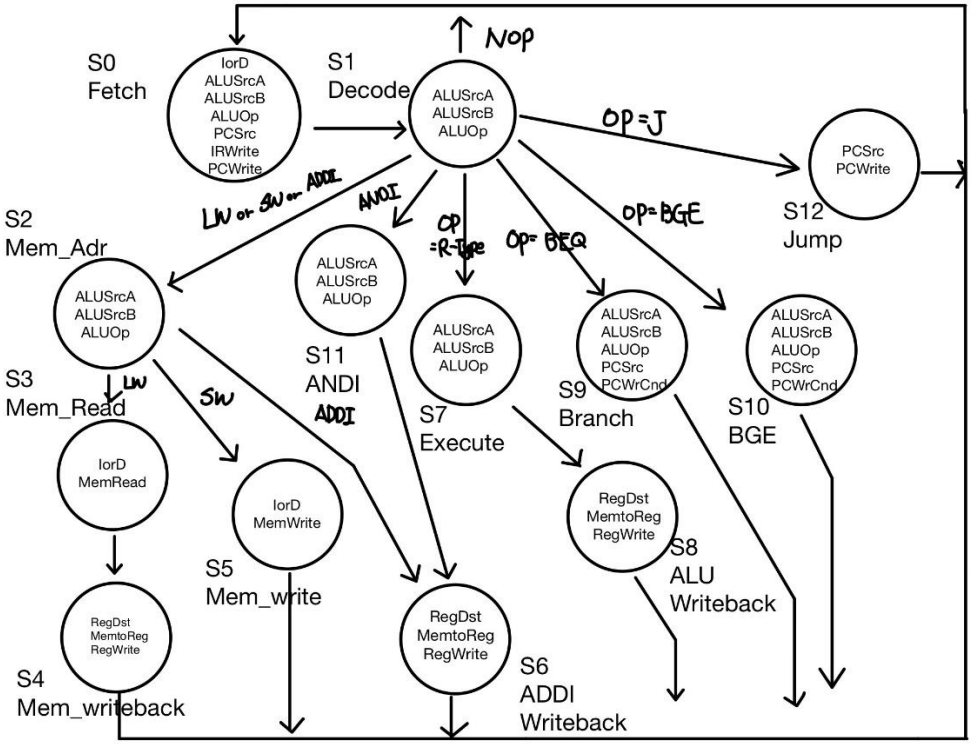
- **BLTZ**

  BLTZ 명령어는 Branch를 활용하여 구현하였다. ALUSrc를 통하여 rs rt reigster을 지정하였으며 ALUOp로 sub를 하도록 하였다. BLTZ의 경우 rt가 0으로 입력되기 때문에 rs – 0의 값이 음수일 때 Branch의 동작을 하게 된다. 하지만 이번 프로젝트에서는 zero flag가 0일 때와 음수일 때를 or로 묶었기 때문에 rs가 0일 때도 Branch의 동작을 하는 BLEZ의 동작을 하게 된다. 이러한 문제점은 flag를 or로 묶지 않고 선택하여 mux에서 signal을 얻어오면

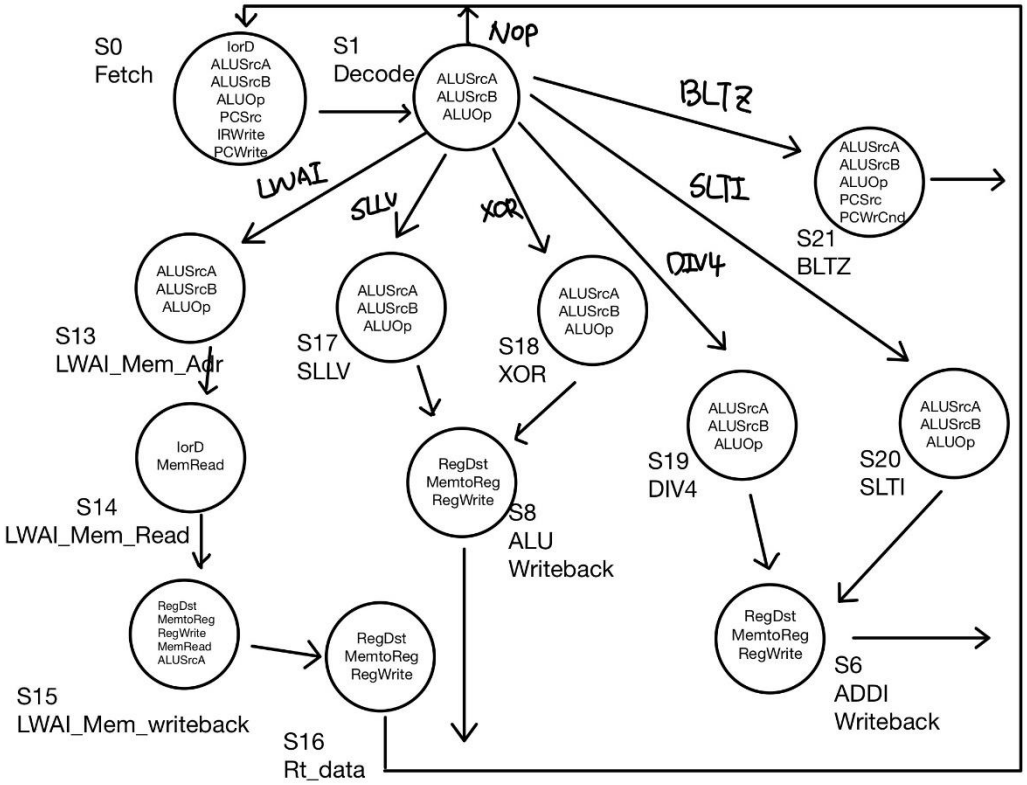
해결할 수 있다.

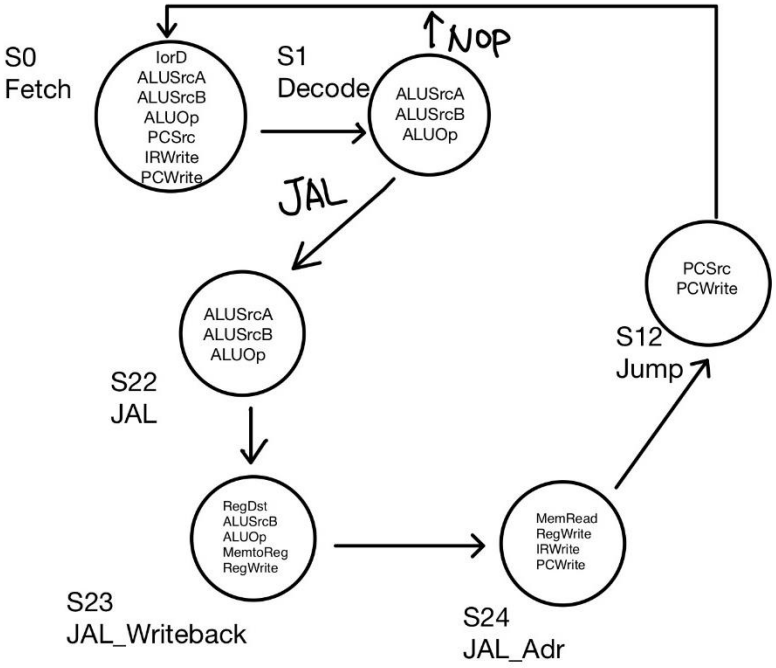
■ 설계 의도와 방법

● 구현한 Multi Cycle CPU FSM Diagram



위 FSM Diagram은 기존 제공된 코드에서 BGE의 동작만을 추가로 한 것이다. 각 명령들은 동작을 끝나면 Fetch로 이동하게 된다.



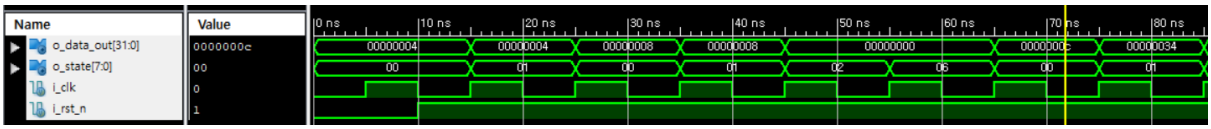


위 FSM Diagram은 나머지 명령어들을 처리하는 Diagram이다. 마찬가지로 모든 동작이 끝나면 Fetch로 돌아가게 된다.

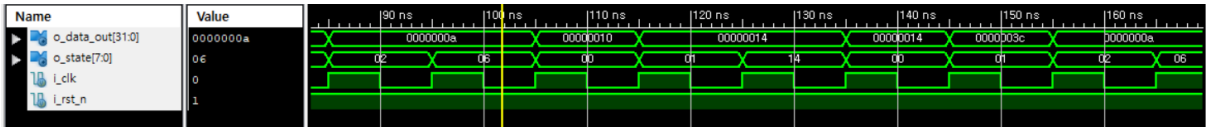
- **시뮬레이션 결과**
  - **inst**

```
NOP
ADDI $v0 $v0 0
ADDI $v1 $v1 A
BLTZL $v1 1
ADDI $a1 $a1 A
SLTI $at $v0 5
NOP
SW $at 4 $zero
LW $s2 4 $zero
NOP
LBU $a1 0 $a2
```

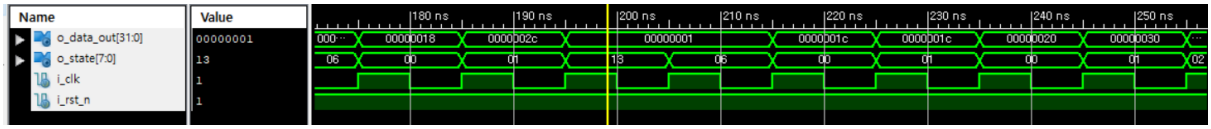
프로젝트에서 주어진 inst파일은 위와 같다.



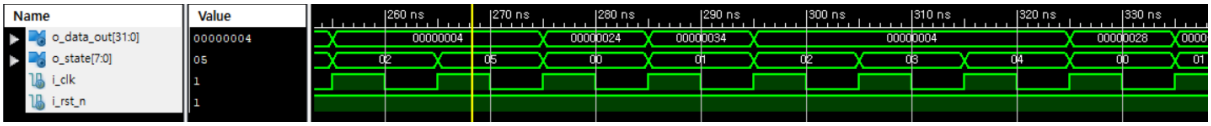
이는 실행 화면이다. o_state를 보면 먼저 NOP이기 때문에 1로 갔다가 다시 0으로 간다. 이후 1로 간 후 2로 가게 되는데 이는 Mem_Adr state이며 ADDI 명령어를 수행하기 위해 지나치는 state이다. O_data_out을 보면 0이 정상적으로 저장되는 것을 확인할 수 있다. 이후 저장이 끝나면 state가 0으로 바뀌며 state는 다시 1로 이동하여 decode를 한다.



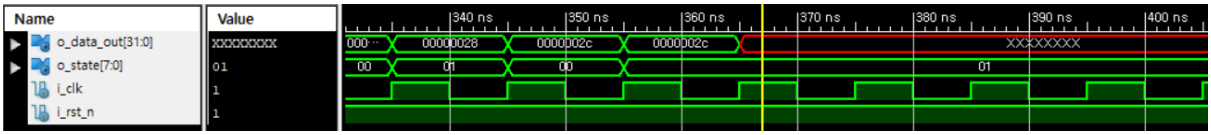
이 화면의 앞부분은 ADDI로 A를 입력한 화면이다. O_data_out에서 a가 입력되는 것을 확인할 수 있으며 저장이 끝나고 state가 0으로 바뀐다. 이후 1로 이동하여 decode를 하여 BLTZ명령어를 수행한다. V1에 저장된 값이 a이기 때문에 Branch의 동작을 하지 않게 된다. 따라서 아무런 변화가 없는 것을 확인하였다. State가 0으로 바뀐 후 1로 이동하여 decode를 하여 ADDI의 동작을 하게 된다.



A가 정상적으로 저장되었으며 다음으로 SLTI를 입력 받는다. V0의 값이 0이므로 5보다 작아 Set된 값이 출력되는 것을 확인하였다. 이후 NOP를 하고 다음 명령어에 대해 decode를 수행한다.

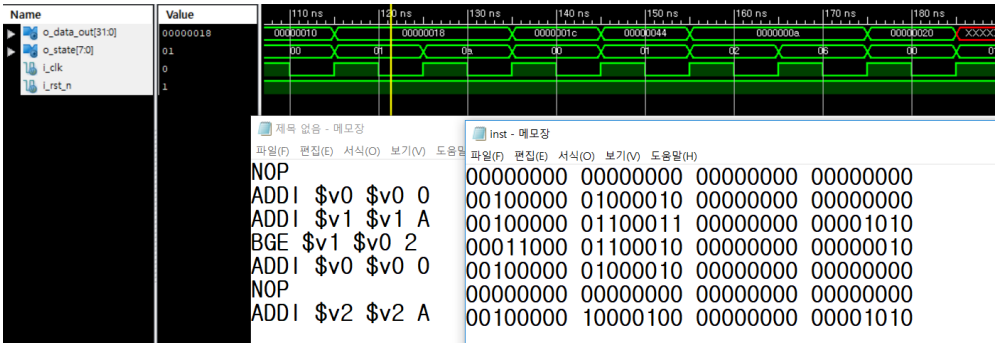


SW를 수행하고 이후 LW를 수행하는 것까지 확인하였다.



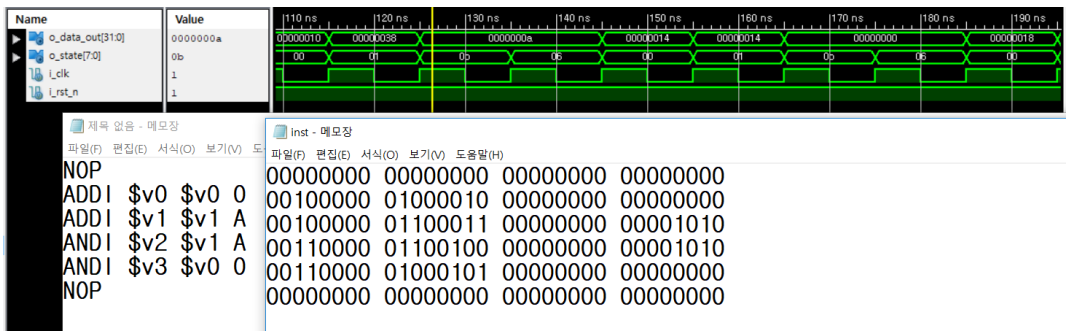
이후 NOP를 하고 LBU를 받았을 때 구현하지 않는 command이기 때문에 state가 변하지 않으며 data의 out이 빨간색으로 출력되지 않는 것을 확인하였다.

- **BGE**



화면은 BGE 명령을 받았을 때부터 실행하는 화면이다. V1의 값은 A이고 V2의 값은 0이므로 조건이 성립하여 immediate값 2를 2bit shift left한 값이 pc에 더해진다. 따라서 BGE 아래 있는 ADDI와 NOP는 무시되며 ADDI V2 V2 A가 실행되는 것을 확인하였으며 정상적으로 작동한다. Decoding부터 종료까지 2clk이 소모된다.

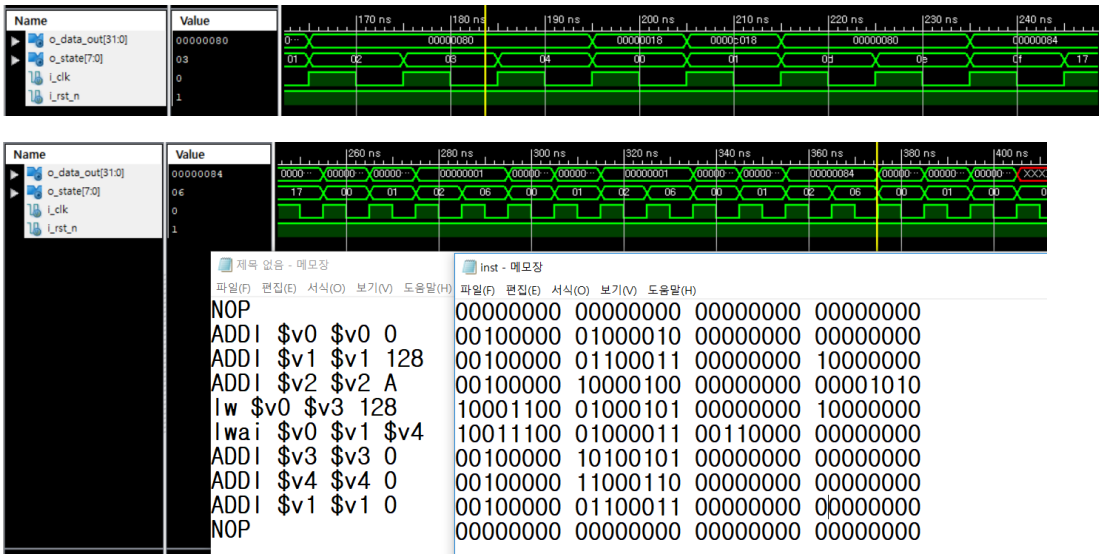
- **ANDI**



이는 ANDI명령을 받았을 때부터 실행하는 화면이다. A값인 v1과 A를 ANDI하여 결과가 A가 나오는 것을 확인하였으며 0값인 v0와 0을 ANDI하여 결과가 0이 나오는 것을 확인하였다. ANDI MIPS instruction에서 $v2 $v1의 위

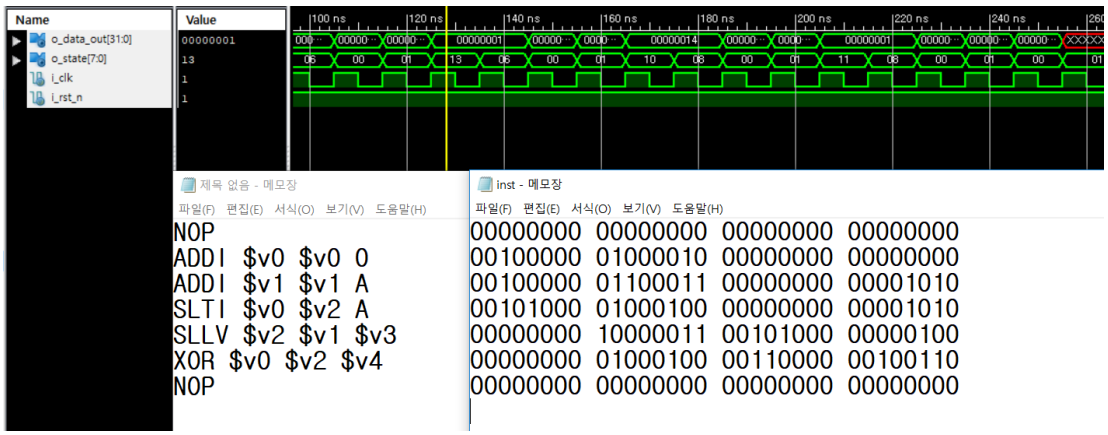
치와 $v3 $v0를 바꾸어 잘못 적었다. decoding부터 종료까지 3clk이 소모된다.

- **LW & LWAI**



LW와 LWAI명령어의 실행 화면이다. Sw가 구현이 되지 않아 data.txt에 저장되어 있는 데이터를 사용하였다. 우선 lw로 메모리의 128번째에 있는 데이터를 V3에 저장하였다. 또 lwai로 128번째에 있는 데이터를 V4에도 저장하였다. 결과 화면을 보면 v3과 v4가 1을 출력하는 것을 확인할 수 있다. lwai 명령어로 인하여 v1의 값을 4만큼 증가시키는 것을 확인하였으며 decoding부터 종료까지 lw는 4clk, lwai는 5clk을 소모하는 것을 확인하였다.

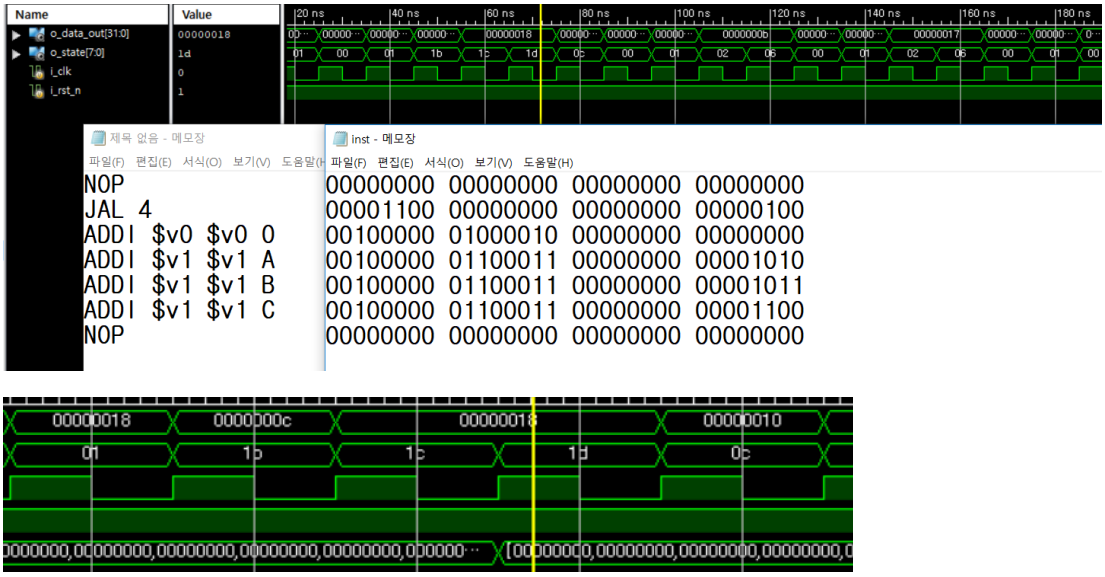
- **SLTI & SLLV & XOR**



SLTI, SLLV, XOR명령어의 실행 화면이다. 우선 v0에 0, v1에 A를 저장하였고,

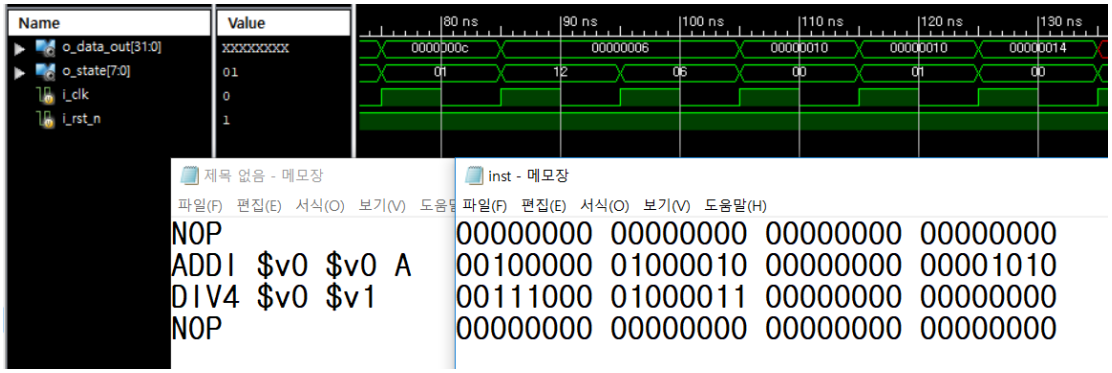
SLTI에서 v0와 A를 비교하여 그 값을 v2에 넣었다. V2에 1이 저장된 것을 확인하였고 정상 작동하였다. SLLV에서 V1을 V2만큼 Shift left한 값을 V3에 저장하였다. V1이 A, V2가 1이므로 V3엔 20을 의미하는 0x14가 저장되어 있어야 하고, 정상적으로 작동하는 것을 확인하였다. XOR에서 0과 1을 넣었으므로 1이 저장되어야 하고, 정상적으로 저장된 것을 확인하였다. 각 명령어는 모두 decoding부터 끝까지 3clk을 소모한다.

- **JAL**



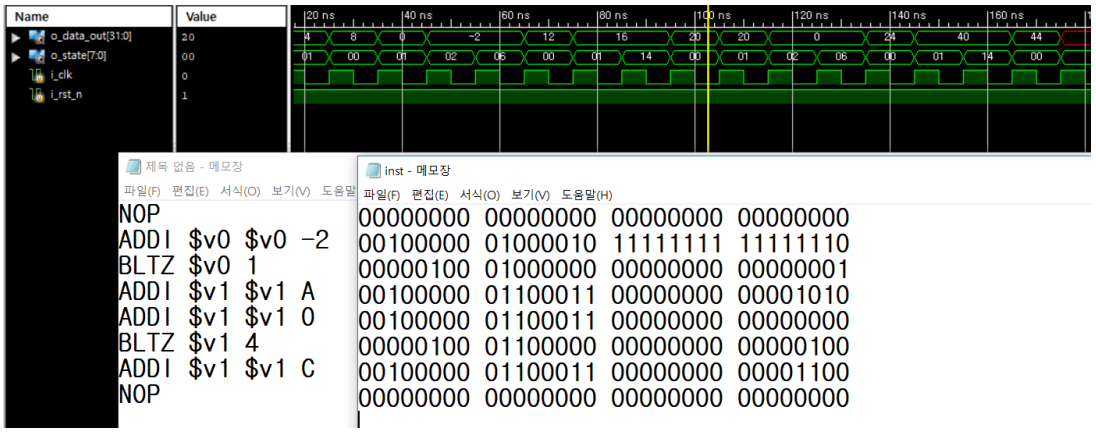
JAL 명령어의 실행 화면이다. JAL 4와 register에 값을 더하는 명령어를 넣어 가장 먼저 실행되는 명령어가 무엇인지 확인하였다. 결과 화면에서 보이듯이 ADDI $v1 $v1 B가 실행되었고, 그 후에 순차적으로 실행이 되는 것을 확인할 수 있다. Writeback state이후 31번 register에 정상적으로 PC값이 저장되는 것을 확인하였다. JAL의 경우 decoding부터 끝까지 5clk이 소모된다.

- **DIV4**



DIV4의 실행 화면이다. V0에 먼저 A값을 저장하였다. DIV4에서 v0의 값을 4로 나눈 값이 v1에 들어가야 한다. 하지만 이번 과제에서는 ALU의 동작이 DIV도 없고, Shift left도 R-type에서만 동작하도록 설계되어 있기 때문에 다른 방법으로 구현하기 위해서 너무 많은 clk이 필요했다. 이는 과제 출제 의도와 거리가 먼 것 같아 동작은 임의의 operation을 하도록 하였으며 보고서에 이 명령어가 제대로 작동하기 위해서 어떻게 고쳐야 하는지에 대해 서술하였다.

- **BLTZ**



BLTZ의 실행 화면이다. V0에 -2값을 저장하였다. 그 이후 BLTZ V0 1을 하게 되면 V0가 0보다 작기 때문에 1을 Shift left 2 bit하여 pc값에 더하게 된다. 따라서 아래 ADDI $v1 $v1 0x000a를 건너뛰게 되며 ADDI $v1 $v1 0x0000이 실행된다. V1에 0이 저장되어 있을 때 BLTZ $v1 4를 하면 원래 pc값에 16을 더하지 않아야 하는데 이번 프로젝트의 zero flag가 다른 방식으로 전달되어져 있기 때문에 0의 값을 받아도 Branch의 동작을 하게 된다. 이 명령어는 Decoding부터 끝까지 2clk만에 종료하게 된다.