Computer Architecture

Assignment #1

담당 교수님 : 이성원 교수님

컴퓨터정보공학부

2018202013

정영민

■ 문제 해석 및 해결 방향

● 각 명령어의 기능과 동작

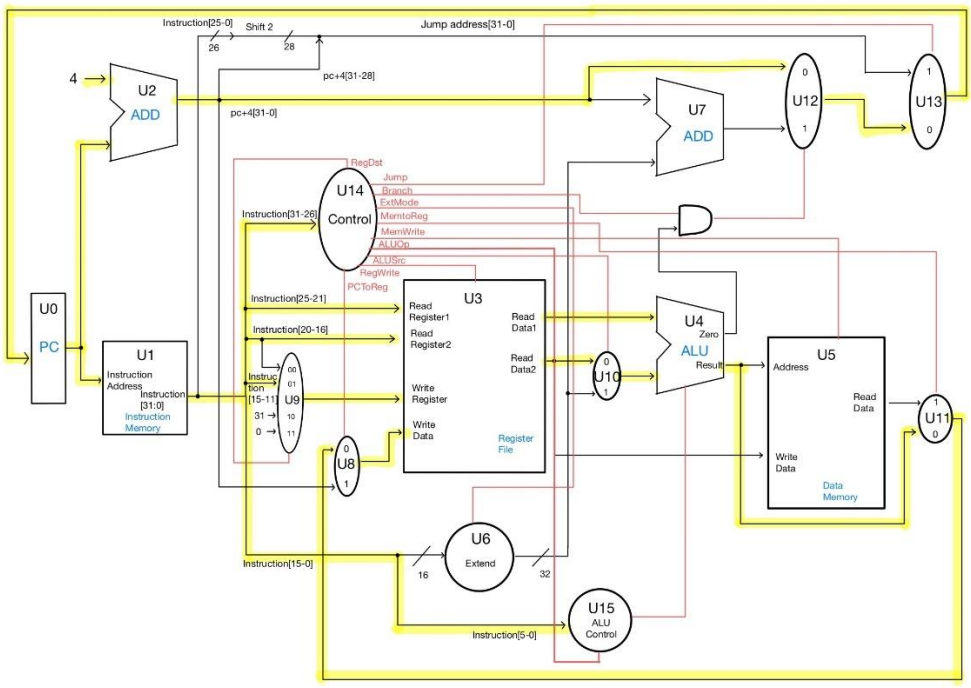
◆ SLL

Opcode(6)	Rs(5)	Rt(5)	Rd(5)	Shamt(5)	Funct(6)

SLL은 R-Type instruction이다. 따라서 입력은 위와 같이 6bit의 opcode, 5bit의 rs, 5bit의 rt, 5bit의 rd, 5bit의 shamt, 6bit의 function으로 이루어져 있다. 이 때, 6bit의 opcode는 전부 0이며 맨 뒤의 6bit의 function bit로 R-Type instruction을 구분하게 된다.

Instruction	Operation
Sll rd, rt, sa	Rd = rt << sa

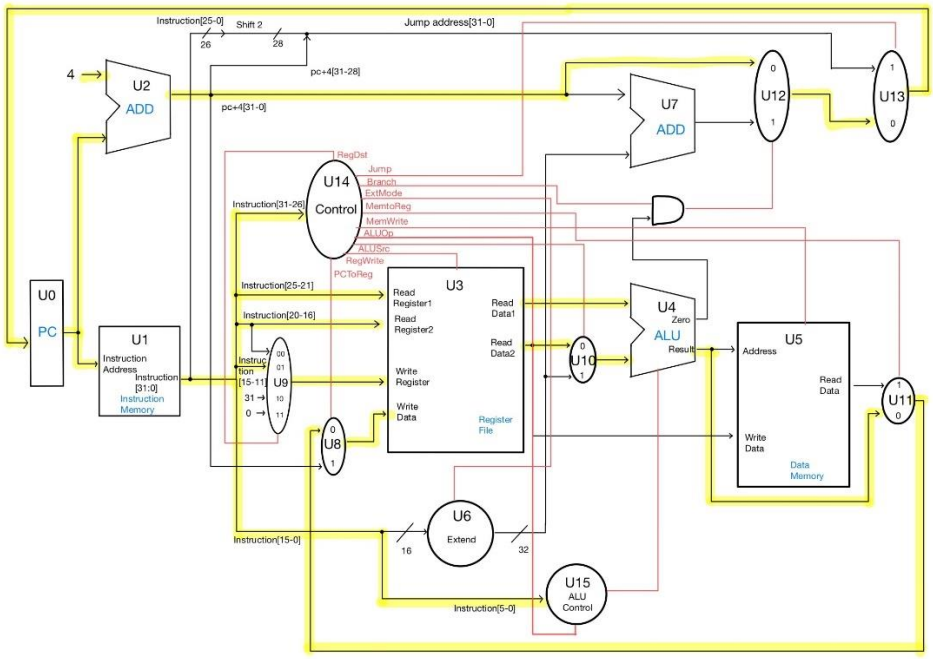
SLL의 function code는 000000이며, sll rd, rt, sa의 형태로 입력된다. 해당 입력의 동작은 rd register에 rt register의 값을 sa만큼 shift left한 값을 저장하게 된다.



- **SRA**

Instruction	Operation
Sra rd, rt, sa	Rd = rt >> sa

SRA은 R-Type instruction이다. SRA의 function code는 000011이며, sra rd, rt, sa 의 형태로 입력된다. 해당 입력의 동작은 rd register에 rt register의 값을 sa만큼 shift right한 값을 저장하게 된다.



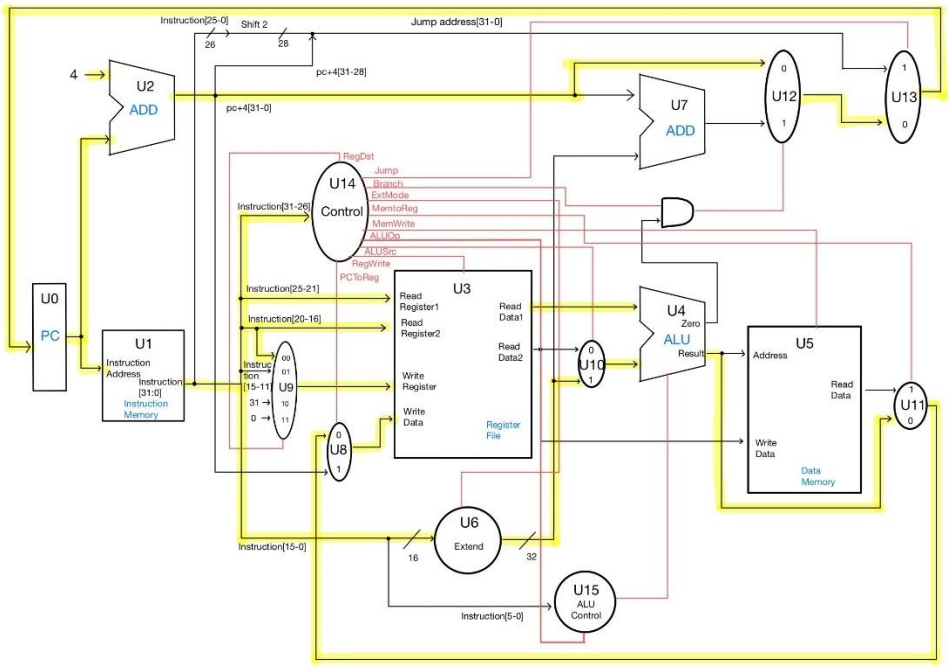
- **ANDI**

Opcode(6)	Rs(5)	Rt(5)	Immediate(16)

ANDI는 I-Type instruction이다. 따라서 입력은 위와 같이 6bit의 opcode, 5bit의 rs, 5bit의 rt, 16bit이 immediate 값으로 이루어져 있다.

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

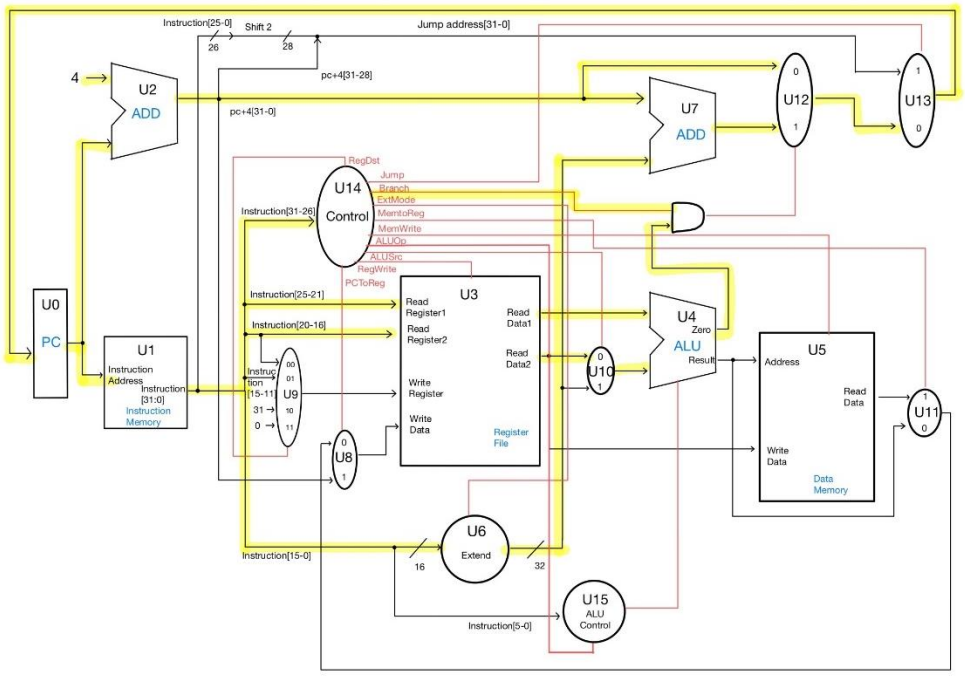
ANDI의 opcode는 001100이며, addi rt, rs, immediate의 형태로 입력된다. 해당 입력의 동작은 rt register에 rs register와 zero extension을 한 immediate 값을 and연산하여 나온 결과값을 저장하게 된다.



- **BNE**

Instruction	Operation
Bne rs, rt, label	If (rs != rt) PC += immediate << 2

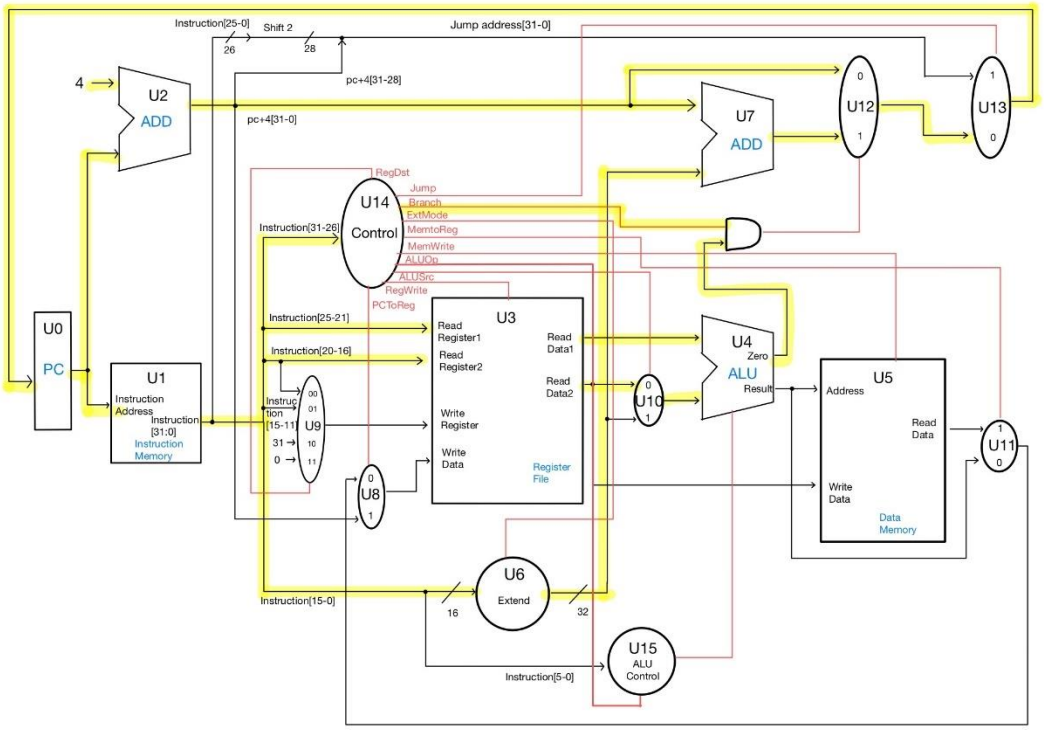
BNE는 I-Type instruction이다. BNE의 opcode는 000101이며, bne rs, rt, label의 형태로 입력된다. 해당 입력의 동작은 만약 rs register의 값과 rt register의 값이 다를 경우 pc에 immediate값을 shift left 2 bit한 값을 더하게 된다.



- **BEQ**

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

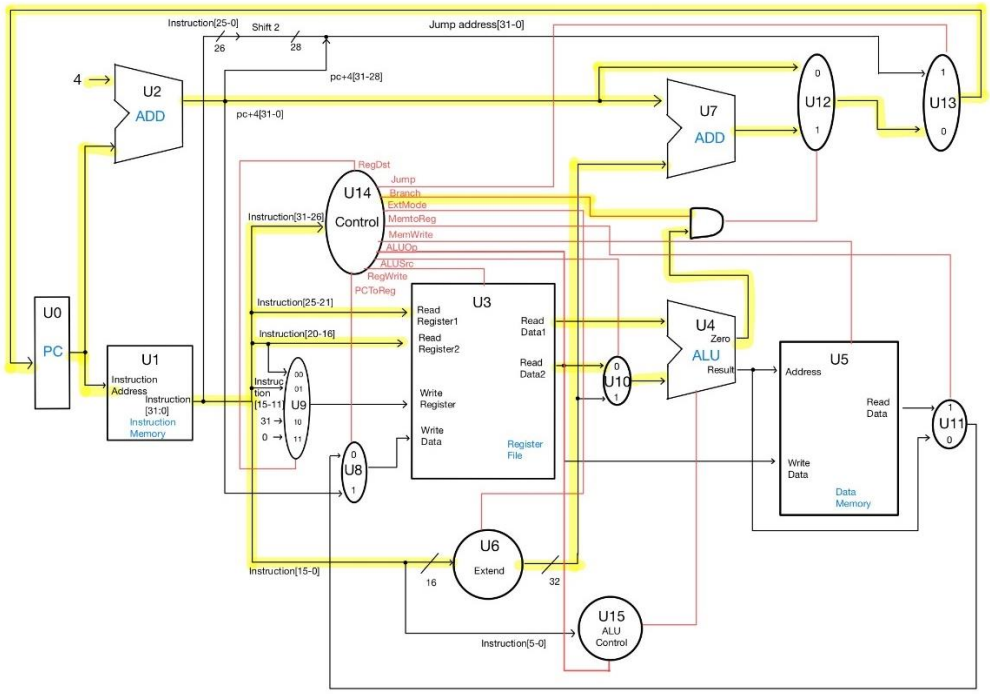
BEQ는 I-Type instruction이다. BEQ의 opcode는 000100이며, beq rs, rt, label의 형태로 입력된다. 해당 입력의 동작은 만약 rs register의 값과 rt register의 값이 같을 경우 pc에 immediate값을 shift left 2 bit한 값을 더하게 된다.



- **MULTU**

Instruction	Operation
Multu rs, rt	Hi:lo = rs * rt

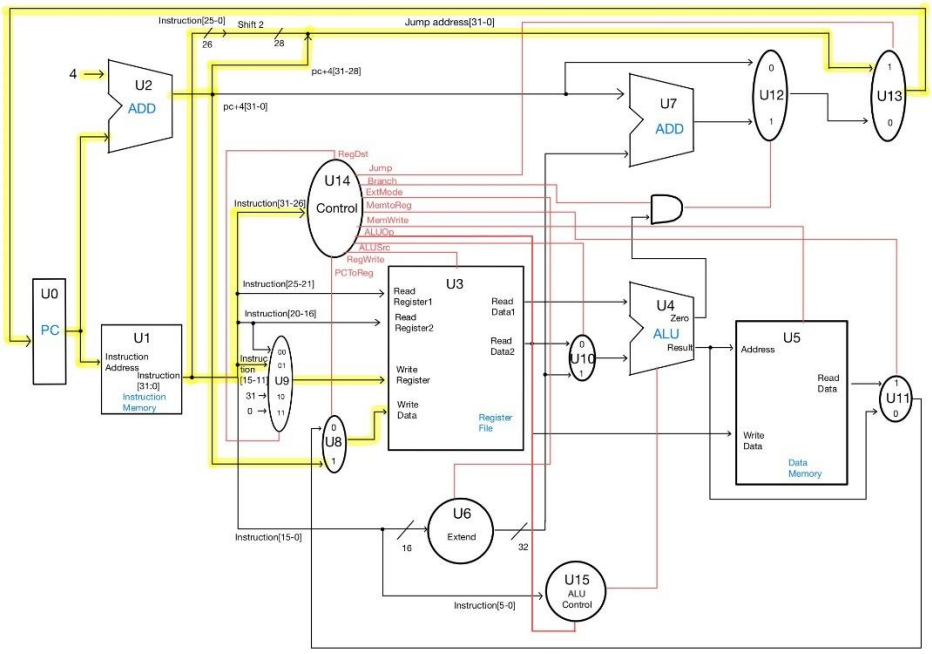
MULTU는 R-Type instruction이다. MULTU의 function code는 011001이며, multu rs, rt의 형태로 입력된다. 해당 입력의 동작은 rs register의 값과 rt register의 값을 곱하여 상위 32bit는 high register에 담고, 하위 32bit는 low register에 담게 된다. 해당 연산은 unsigned로 진행된다.



- **JALR**

Instruction	Operation
Jalr rd, rs	Rd = PC+4, PC = rs

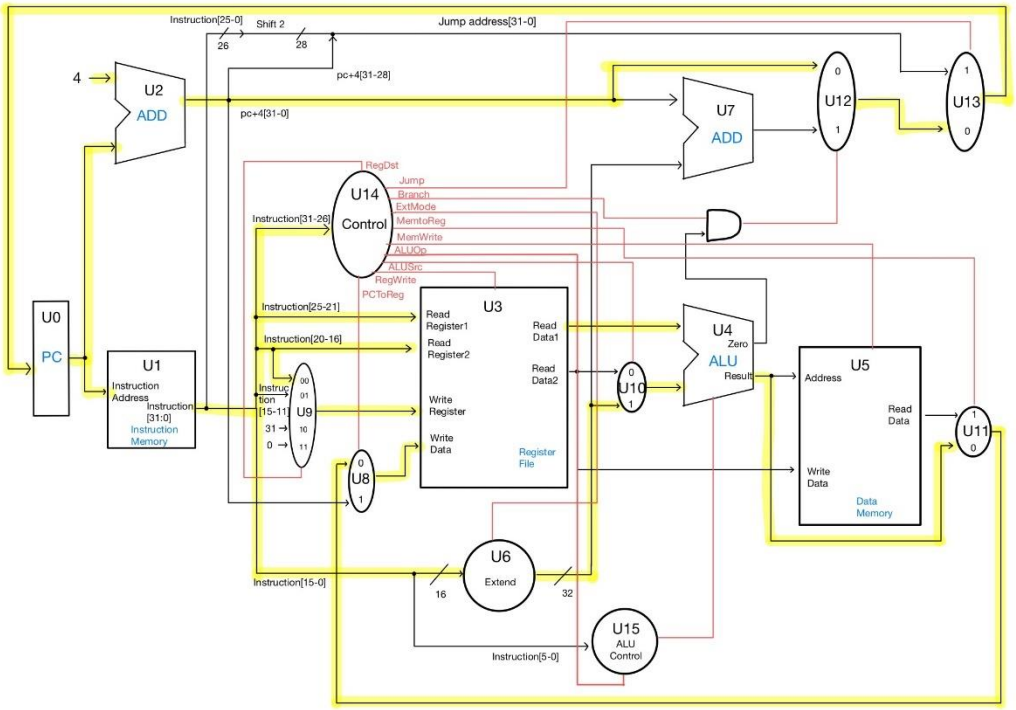
JALR은 R-Type instruction이다. JALR의 function code는 001001이며, jalr rd, rs의 형태로 입력된다. 해당 입력의 동작은 rd register에 PC+4의 값을 넣고, PC에 rs register의 값을 넣는다. 여기서 rd는 destination register이며, rs는 jump할 address를 가진 register이다.



- **XORI**

Instruction	Operation
xori rt, rs, immediate	Rt = rs ^ ZeroExtension(immediate)

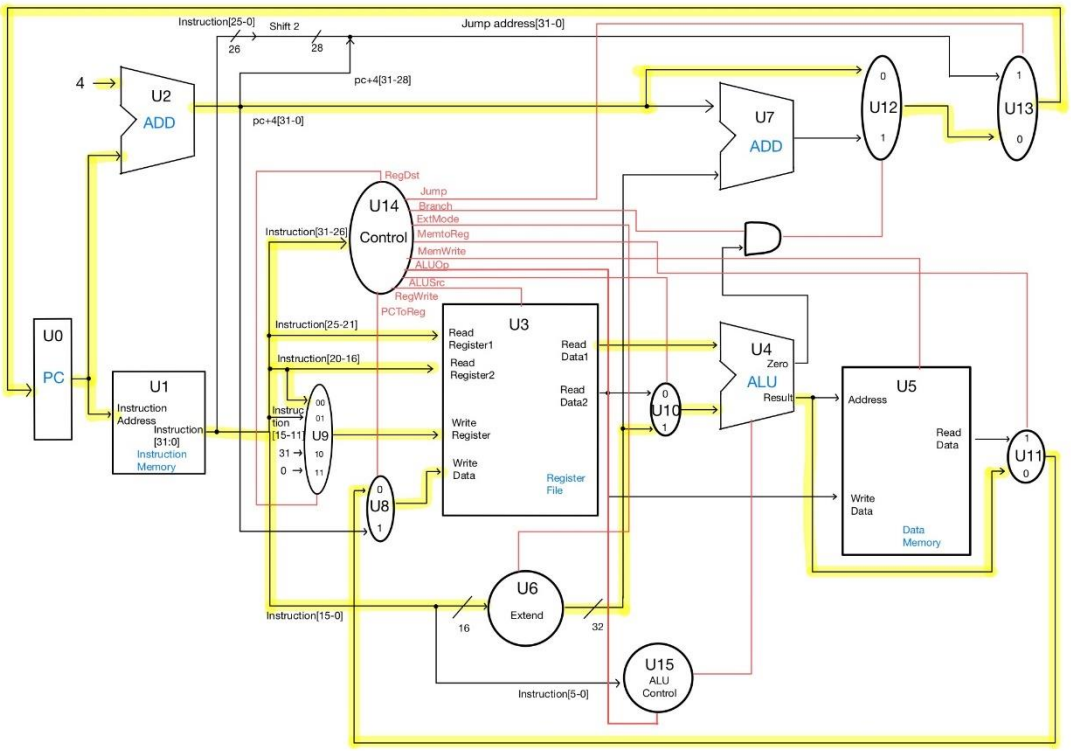
XORI는 I-Type instruction이다. XORI의 opcode는 001110이며, xori rt, rs, immediate의 형태로 입력된다. 해당 입력의 동작은 rt register에 rs register의 값과 zero extension을 한 immediate값을 xor연산한 결과 값을 저장하게 된다.



- **SLTI**

Instruction	Operation
Slti rt, rs, immediate	Rt = (rs < SignExtension(immediate))

SLTI는 I-Type instruction이다. SLTI의 opcode는 001010이며, slti rt, rs, immediate의 형태로 입력된다. 해당 입력의 동작은 rs register의 값이 sign extension한 immediate값보다 작을 경우 rt register에 1을 저장한다. 그 외엔 0을 저장하게 된다.



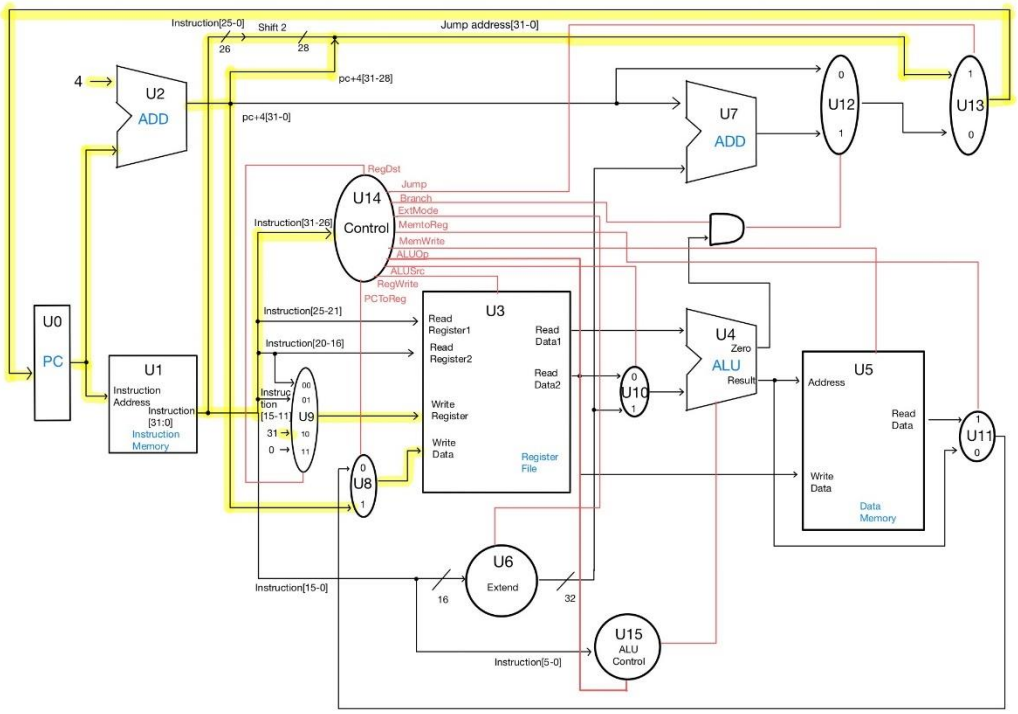
- **JAL**

Opcode(6)	Address(26)

JAL은 J-Type instruction이다. 따라서 입력은 위와 같이 6bit의 opcode와 26bit의 target으로 이루어져 있다.

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

JAL의 opcode는 000011이며, jal label의 형태로 입력된다. 해당 입력의 동작은 31번 register(destination register)에 PC+4의 값을 저장하고, PC의 상위 4bit와 offset을 shift left 2bit한 비트를 이은 32bit를 PC에 저장하게 된다.



- **실험 내용에 대한 설명**

이 하드웨어는 16개의 unit들로 구성되어 있다. U0 PC는 현재 수행하는 명령어의 주소 값을 저장하는 Unit이다. U2의 Adder을 이용하여 4를 더하며 주소 값을 증가시켜 이를 다시 PC에 넣어 다음 명령어를 수행할 수 있도록 이동하게 된다. U1 Instruction Memory에서 instruction들을 저장하고, 해당 instruction들을 opcode의 값을 보며 U14 Control Unit이 Signal을 변경하며 데이터의 흐름을 제어한다. U3 Register File은 32x32bit의 register을 가지는 Unit이다. 해당 Unit을 통하여 데이터를 Register에 저장하거나, 읽어올 수 있다. U4 ALU는 연산을 진행하는 Unit이며 Register File에서 읽어온 Register에 대한 값을 읽어와 이전 U15 ALU Control에서 ALUOp signal에 따라 전달한 ALU_funct 값을 이용하여 지정한 연산을 진행하게 된다. U5 Data Memory는 연산 된 값을 주소 값에 저장하거나, 읽어올 수 있는 Unit이다. U6 Extend은 입력 받은 16bit의 immediate값을 zero extension할 지 sign extension할 지 결정하는 Unit이다. U7 Adder은 Branch 명령어를 입력 받았을 때 PC+4의 값에 sign extension한 immediate값을 더하여 해당 값을 PC에 넣을 때 사용되는 Unit이다. U8 mux는 register에 PC+4의 값을 저장할 때 사용되는 mux이다. U9 mux는 어떠한 register에 값을 저장할 지 고를 때 사용되는 mux이다. U10 mux는 register의 값을 선택할 지 immediate값을 선택할 지 고를 때 사용되는 mux이다. U11 mux는 ALU에서 나온 연산 값을 바로 register에 보낼 지 혹은 Data Memory에서 읽은 값을 register에 보낼 지 선택할 때 사용되는 mux이다. U12 mux는 Branch 명령어가 사용되지 않을 때는

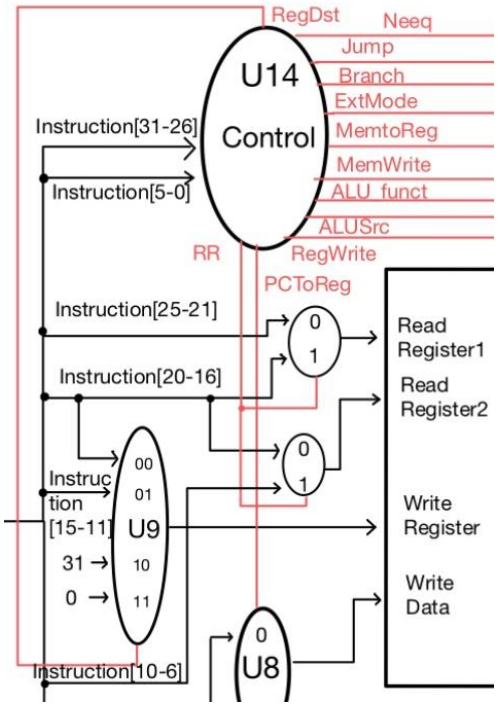
PC+4의 값을 전송하며 Branch명령어를 사용하였을 때는 해당 조건이 만족하는 지 아닌 지에 따라 PC값을 바꾸기 위해 사용되는 mux이다. U13 mux는 jump 명령어를 사용하였을 때 해당 값으로 이동하기 위하여 사용되는 mux이다.

● 문제점 및 개선점

◆ SLL

SLL 명령어를 구현한 방법은 opcode가 0일 때 ALUOp Signal에 0010을 넣어 ALU Control에서 funct bit를 보게 하였고, 이 비트를 통하여 ALU_funct의 값이 00101이 되어 SLL 연산을 ALU에서 하도록 구현하였다. RegDst에 01을 넣어 Rd register에 값을 넣도록 지정하였고 ALUSrc에 0을 넣어 ALU에서 RF의 Read data 끼리 연산을 하도록 하였다. 사용하지 않는 Signal들은 0의 값을 넣었으며 Register에 값을 넣기 위해 RegWrite에 1을 넣었다. 하지만 SLL 명령어는 sll rd, rt, sa로 rs register을 사용하지 않고, rd register에 rt register의 값을 sa만큼 shift 한 값을 저장해야 하는데, 이 하드웨어는 sll의 동작을 수행할 때 rd register에 rs register의 값을 rt register의 값만큼 shift한 값을 저장하게 된다. 즉, 기존 정형화된 R-Type instruction과 같이 데이터를 사용했기 때문에 정상적인 작동을 하지 못하게 된다. 이는 Top file인 SingleCycle의 RF에서 전달받는 인자의 비트 자릿수가 고정되어 있기 때문이다.

이러한 문제를 해결하기 위해선 MyControl의 동작과 mux 2개를 추가해야 한다. MyControl의 동작으로는 Signal을 하나 추가해야 하는데, 이 Signal은 RF에 값을 전달하는 mux(추가한 2개의 mux)에서 사용된다. MyControl의 전달 인자로 function bit도 추가하여 sll 명령어가 입력되었을 때 추가한 Signal을 알맞게 조정하여 RF에 전달할 register가 적혀 있는 bit를 전달한다. 즉 MyControl의 동작 추가에선 Signal ALUOp를 삭제하고 MyControl과 ALUControl을 합쳐서 동작시키면 된다.



위 방법에 따라 수정한 블록도의 부분이다. ALUControl를 삭제하고 해당 동작을 Control에서 하도록 설계하였다.

- **SRA**

SRA 명령어를 구현한 방법은 SLL과 마찬가지로 Opcode가 0일 때 ALUOp Signal에 0010을 넣어 ALU Control에서 ALU_funct의 값을 01000으로 바꾸어 SRA 연산을 ALU에서 하도록 구현하였다. 그 외 Signal들은 sll과 같다. SRA의 문제점 또한 SLL과 같다. Sra rd, rt, sa의 명령을 받아 rd에 rt register의 값을 sa만큼 >>>해야 하지만, rd에 rs register의 값을 rt register의 값만큼 >>>되어 제대로 작동을 하지 않는다.

해결방법 또한 sll과 같은 Signal을 이용하여 RF에 전달할 register을 정상적으로 입력해야 한다. 따라서 수정된 블록도는 위 sll에서 올린 사진과 같다.

- **ANDI**

ANDI 명령어를 구현한 방법은 Opcode가 001100일 때 ALUOp의 값을 바꾼다. 해당 Signal을 AND 동작을 하도록 지정을 하고, RegDst는 00을 넣어 rt register에 값을 저장하도록 하였다. ALUSrc에 1을 넣어 Immediate값을 사용할 수 있도록 하였고 ExtMode에 0을 넣어 Zero Extension을 하도록 하였다. RegWrite에 1

을 넣어 레지스터에 값을 넣고 그 외에 사용하지 않는 Unit의 Signal엔 0을 넣었다. 하지만 이 하드웨어에서 ANDI는 정상 동작을 하지 않는다. ANDI 명령어가 정상적으로 동작을 하지 않는 이유는 ALUControl에서 ALUOp Signal의 값에 따라 AND연산을 하는 부분이 없기 때문이다. 따라서 ANDI가 정상적으로 동작을 할 수 있도록 하기 위해서는 ALUOp조건을 추가하여 AND연산을 하는 부분을 추가하면 된다. 이는 조건을 추가해야 하는 것이기 때문에 블록도가 수정되지는 않는다.

- **BNE**

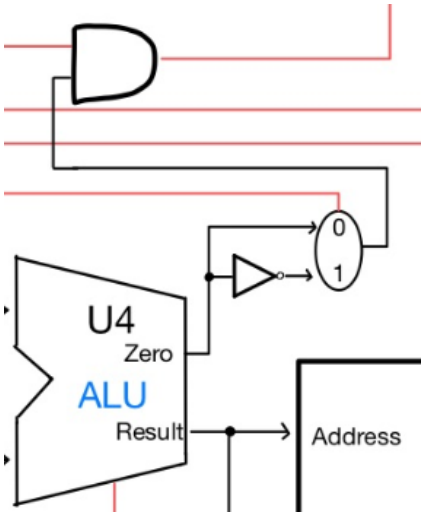
  BNE 명령어를 구현한 방법은 Opcode가 000101일 때 ALUOp의 값을 0001로 바꾼다. 이는 Subtract로 빼기 연산을 한 후 ALU의 Zero flag를 Branch Signal과 AND 연산하여 조건에 충족할 시 Branch로 이동하기 위함이다. 따라서 Branch Signal에 1의 값을 넣는다. BNE는 rs와 rt register의 값을 비교하기 때문에 ALUSrc는 0을 넣어 두 register 그대로 비교를 하였다. Register에 값을 넣는 것이 아니기 때문에 RegWrite는 0을 넣고 RegDst값은 Don't care이다. ExtMode는 immediate 값을 Sign Extension하기 때문에 1이 값을 넣었다. 그 외에 사용하지 않는 Unit의 Signal은 0이다.

  이 하드웨어에서는 ALU의 Zero flag가 반대로 설정되어 있어 BNE가 정상 작동하게 된다.

- **BEQ**

  BEQ 명령어를 구현한 방법은 Opcode가 000100일 때 ALUOp의 값을 0001로 바꾸어 Bne와 같이 동작한다. 그 외의 나머지 Signal들도 BEQ와 같게 설정하였다. 하지만 이 하드웨어에서는 BEQ 명령어가 정상적으로 작동하지 않게 된다. 그 이유는 ALU의 Zero flag가 반대로 설정되어 있기 때문이다.

  BEQ가 정상적으로 작동하기 위해선 Zero flag의 값을 정상적으로 설정하면 되지만, BNE 명령어와 같이 동작할 수 없게 된다. 따라서 이 두 명령어들을 같이 작동할 수 있게 하려면 Zero flag의 값을 두 갈래로 나누어 하나는 convert하여 mux에 연결하고, 나머지 하나는 그대로 연결하여 Opcode를 이용하여 값을 선택하여 전달하면 된다.



수정한 블록도는 위와 같다. Mux에 연결된 signal은 beq와 bne를 구분하는 signal이며, 그에 따라 zero값을 바꿔 전달하게 된다. 이와 같이 구현하면 정상적으로 동작할 수 있게 된다.

- **MULTU**

  MULTU 명령어를 구현한 방법은 Opcode가 0일 때 ALUOp의 값을 0010으로 바꾸어 ALUControl에서 function bit를 확인하여 ALU_funct 신호를 바꾸어 ALU에서 multu연산을 하도록 구현하여야 하지만 mult 명령어만 구현이 되어있고 multu 명령어는 구현이 되어있지 않기 때문에 정상적으로 동작을 하지 않게 된다. 그 외의 Signal들은 sll과 같게 된다.

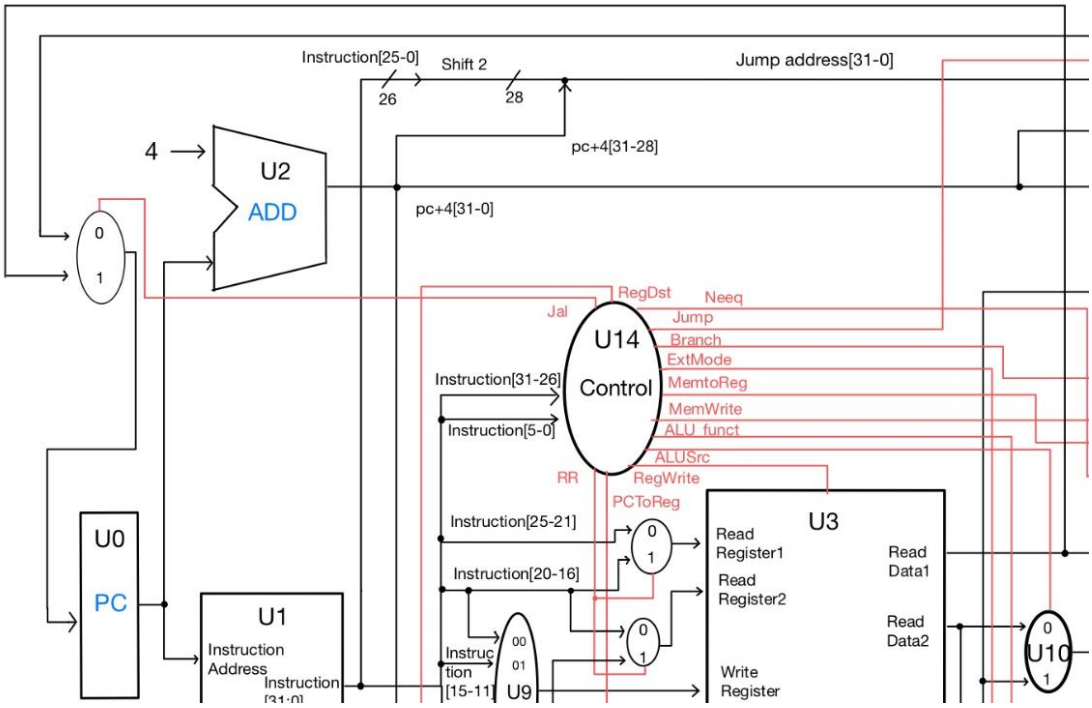
  따라서 이를 해결하기 위해선 funct bit의 조건을 하나 추가하여 multu 연산을 하는 부분을 추가한 후에 ALU에서 해당 Signal에 맞춰서 multu연산을 추가하면 된다. 이는 조건을 추가하면 해결되는 명령어이므로 블록도가 수정되지 않는다.

- **JALR**

  JALR 명령어를 구현하는 방법은 Opcode가 0일 때 function bit를 읽어 해당 명령어를 우선 인식을 해야 한다. 이 하드웨어에서 JALR은 정상적으로 작동하지 않는데 그 이유는 opcode가 0인 모든 입력은 Signal을 같도록 하지만 JALR은 R-Type Instruction들과는 다른 Signal값이 필요하기 때문이다. 대표적인 예시로 Jump signal을 사용하기 때문에 정상적으로 작동하지 않는다.

  JALR 명령어를 정상적으로 작동시키기 위해서 우선 ALU관련 Signal들은 사용하지 않으므로 Don't care값을 넣고, RegDst에 01을 넣어 Rd register에 값을 입력

하도록 한다. RegWrite엔 1을 넣고 Memory 관련 Signal은 0을 넣어 사용하지 않는다. Jump는 1을 넣고 Branch와 ExtMode는 Don't care, PCToReg는 1을 넣어 Register에 PC+4의 값을 입력할 수 있도록 한다. 마지막으로 PC값에 Rs register의 값을 넣을 수 있도록 해야 하는데, 이를 위해선 RF에 새로운 path를 추가하여 Read Data1 값을 PC에 넣을 수 있도록 해야 한다. 이를 위해선 mux가 하나 더 필요하며, 이 mux에 들어가는 입력 값은 이전 PC+4와 Rs register의 값이다. 특별한 Signal에 따라 어떤 값을 출력할 지 결정하도록 하면 정상적으로 작동할 것이다.



이와 같이 수정된 Control과 Read data를 mux에 연결하여 PC에 rs register의 값을 전달할 수 있게 하고, 앞서 말했던 것과 같이 jalr 명령어에 맞게 signal들을 설정하면 정상적으로 작동할 수 있게 된다.

- **XORI**

XORI 명령어를 구현한 방법은 Opcode가 001110일 때 ALUOp의 값에 0011을 넣어 ALU_funct의 값을 01011로 바꾸어 ALU에서 XOR연산을 하도록 구현하였다. RegDst엔 00을 넣어 rt register을 지정하였고, ALUSrc에 1을 넣어 immediate값을 사용할 수 있도록 하였다. ExtMode엔 0을 넣어 zero extension을 하였고 MemtoReg엔 0을 넣어 ALU의 연산 값을 Register에 넣었다. 이때 RegWrite는 1이다. 그 외에 사용하지 않는 Unit의 Signal들은 0을 넣었다.

- **SLTI**

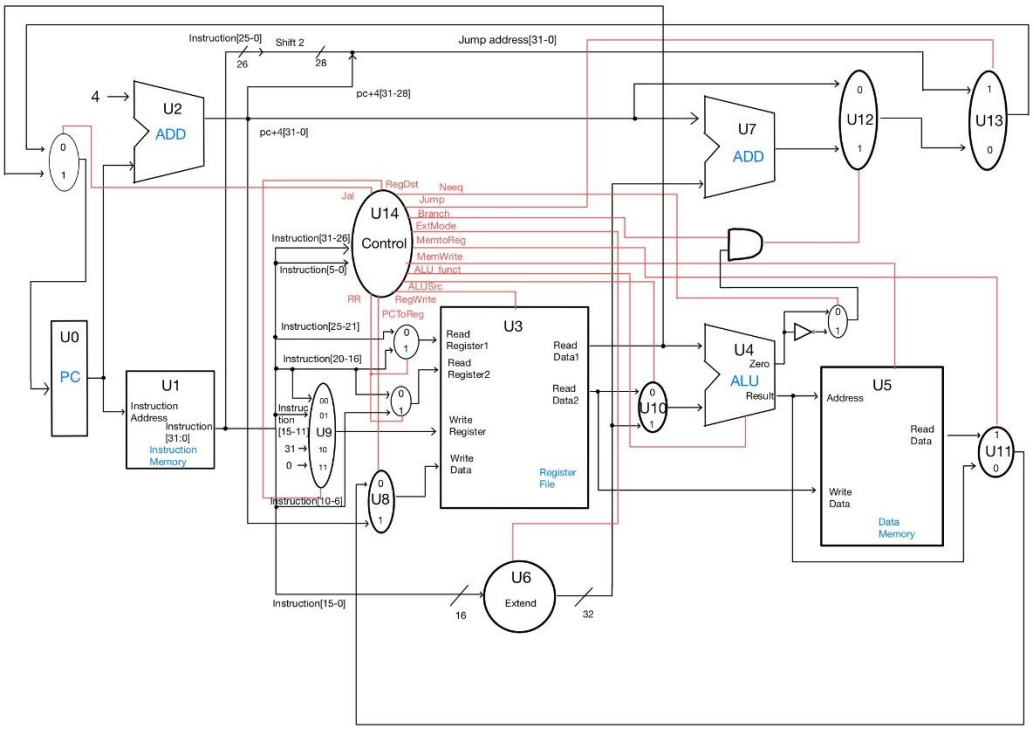
  SLTI 명령어를 구현한 방법은 Opcode가 001010일 때 ALUOp의 값에 0100을 넣어 ALU에서 SLT연산을 하도록 구현하였다. RegDst엔 00을 넣어 rt register을 지정하였고, ALUSrc에 1을 넣어 immediate값을 사용할 수 있도록 하였다. ExtMode엔 1을 넣어 Sign Extension을 하였고 MemtoReg엔 0을 넣어 ALU의 연산 값을 Register에 넣었다. RegWrite는 1이며 그 외 사용하지 않은 Unit의 Signal들은 0을 넣었다.

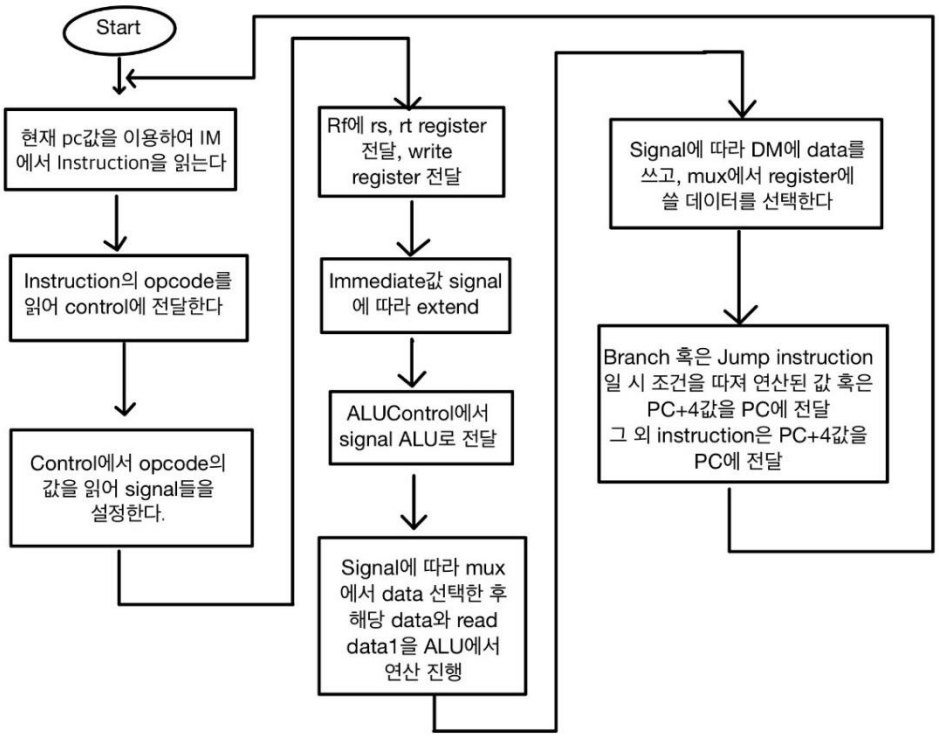
- **JAL**

  JAL 명령어를 구현한 방법은 Opcode가 000011일 때 ALUOp의 값은 don't care이며 RegDst에 10을 넣어 31번 register에 값을 저장하고 이때 RegWrite는 1이다. Jump엔 1을 넣으며 Register에 PC+4의 값을 저장해야 하기 때문에 PCToReg는 1이고 그 외 Signal은 0 혹은 don't care값을 넣었다.

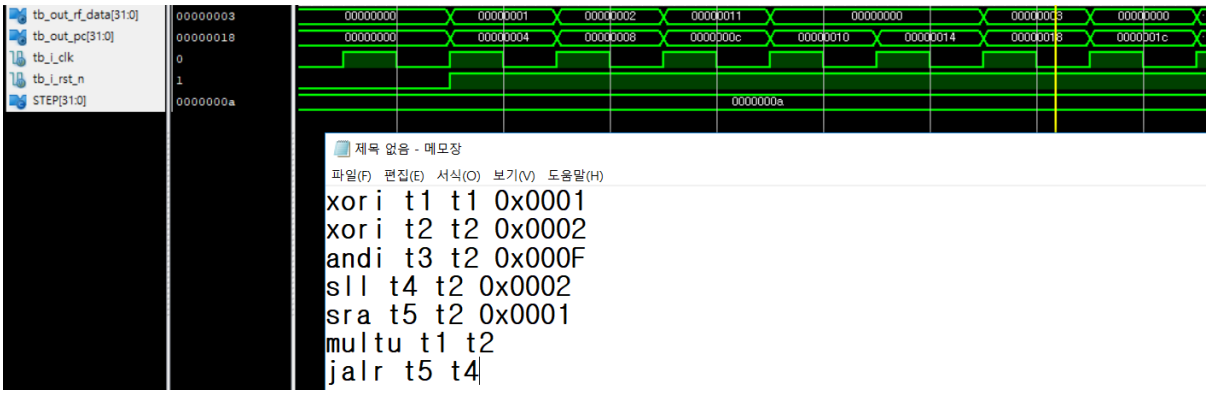
■ 설계 의도와 방법

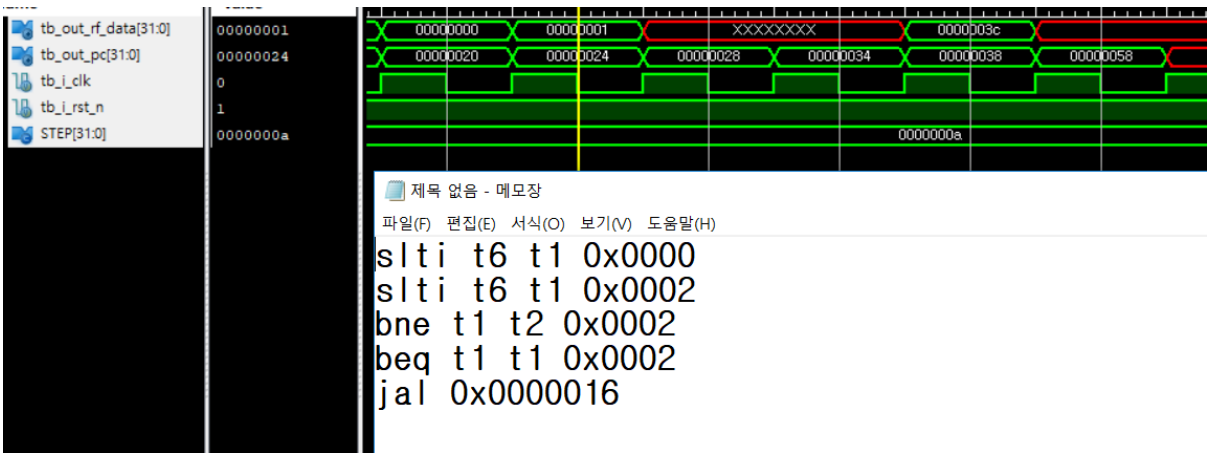
● 구현한 Single Cycle CPU 블록도



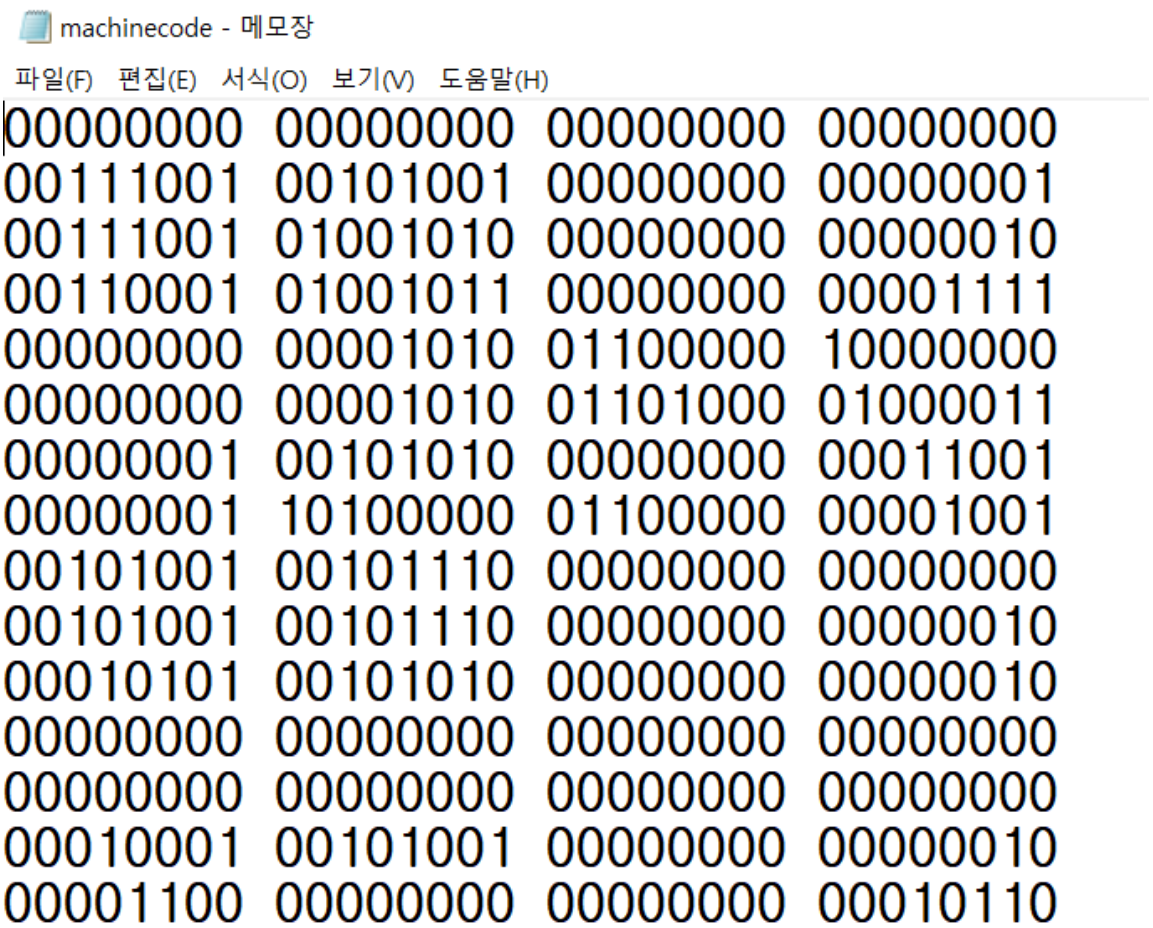


- **전체 testbench 비교 분석**





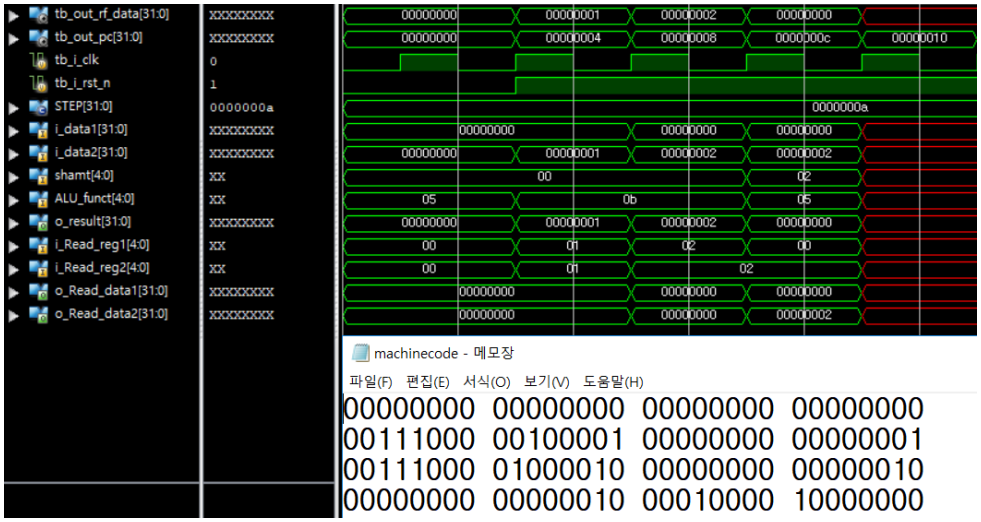
이는 추가한 모든 명령어를 수행할 수 있는 machinecode를 실행한 결과창이다. XORI를 통하여 정상적으로 register에 값이 들어가 있고, ANDI의 연산은 구현이 안되어 있어 ADDI의 동작을 하도록 설정하여 ADDI의 동작을 하는 것을 확인할 수 있다. Sll과 sra는 0을 shift했기 때문에 결과 값이 0으로 오작동 되는 것을 확인할 수 있으며 multu는 연산이 구현되지 않아 ALUOp인 0010의 앞에 0을 붙여 ALU_funct가 00010이 되어 ADD 연산을 수행하는 것을 확인하였다. Jalr의 경우 jump도 되지 않고, pc+4의 값이 register rd에 써지지도 않는 것을 확인하였다. Slti 명령어는 1의 값을 가진 t1 register과 0, 2를 비교하여 결과값을 비교하였다. 제대로 출력되는 것을 확인할 수 있다. Bne instruction의 경우 정상적으로 작동하여 주소 값이 PC+4에 immediate 값을 shift 2하여 더한 값이 제대로 출력되는 것을 확인할 수 있다. Beq의 경우는 정상적으로 작동하지 않고 다음 instruction으로 넘어가게 된다. 마지막으로 jal instruction에서 PC+4의 값을 31번 레지스터에 정상적으로 저장하여 0x0000003c가 저장되는 것을 확인할 수 있으며, 그 후 전달한 label로 정상적으로 jump하는 것을 확인할 수 있다.



위는 machinecode이며 bne와 beq 사이에 주소 값의 변경이 일어나기 때문에 32'b0을 넣어 해당 주소 값을 채워 넣었다.

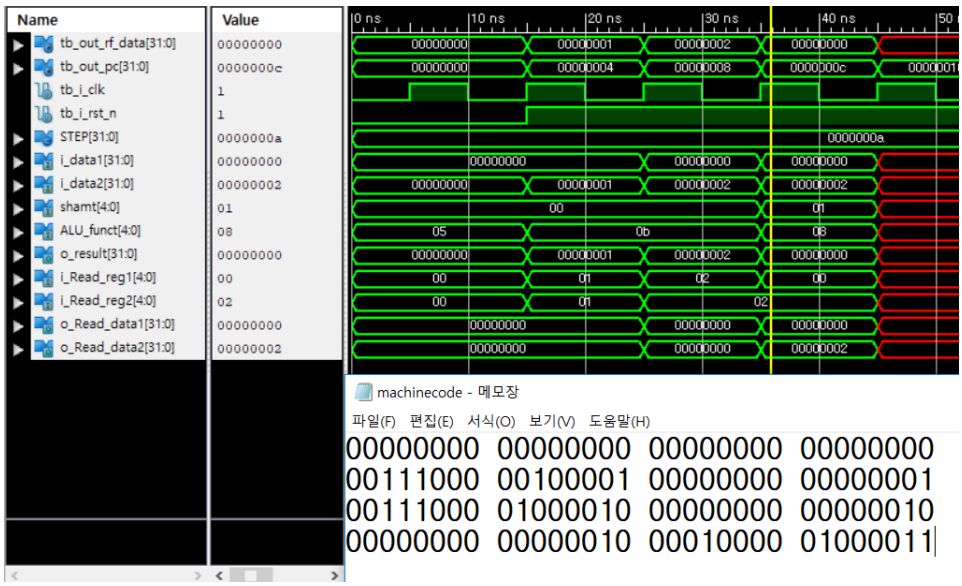
- **시뮬레이션 결과와 예상 결과 비교 분석**

  - **SLL**



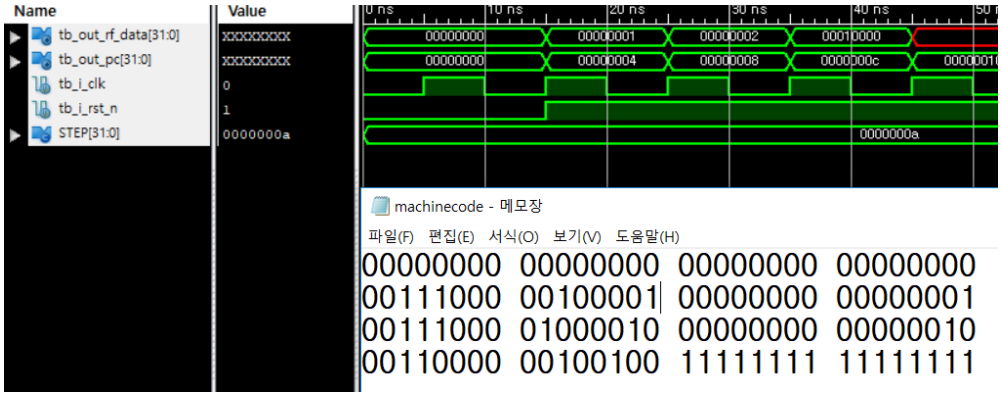
이는 XORI를 이용하여 1번째 register에 1, 2번째 register에 2의 값을 넣은 후, SLL instruction을 입력한 화면이다. rs비트에 0을 넣었고, rt와 rd에 00010을 넣었고 shamt에 2를 넣었는데, SLL이 i_data1을 보면 rs의 값을 가져왔고, i_data2를 보면 rt의 값을 가져와 0을 2만큼 shift하는 것을 볼 수 있다. 따라서 0을 shift했기 때문에 0이 결과로 출력되는 것을 확인할 수 있다. 예상 값은 2에 4를 곱한 8의 숫자가 나와야 하지만, 하드웨어 구성의 오류로 인하여 예상 값과 다르게 나온 것을 확인할 수 있다.

  - **SRA**



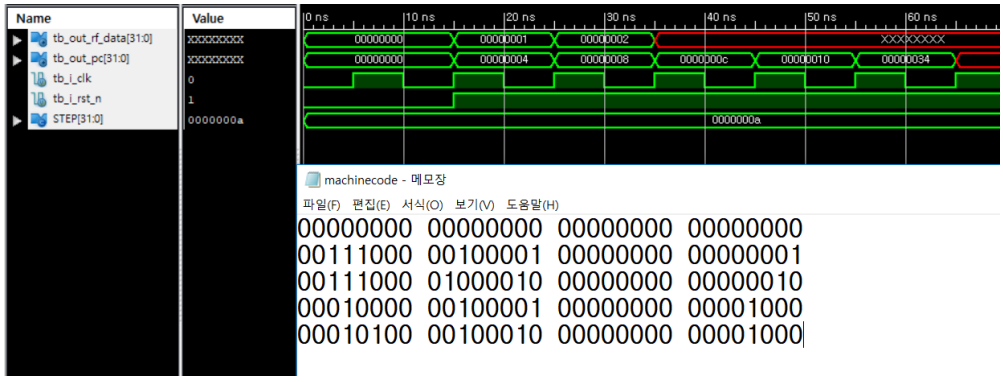
이는 XORI를 이용하여 1번째 register에 1, 2번째 register에 2의 값을 넣은 후, SRA instruction을 입력한 화면이다. rs비트에 0을 넣었고, rt와 rd에 00010을 넣었고 shamt에 1을 넣었는데, SRA가 i_data1을 보면 rs의 값을 가져왔고, i_data2를 보면 rt의 값을 가져와 0을 1만큼 shift하는 것을 볼 수 있다. 따라서 0을 shift했기 때문에 0이 결과로 출력되는 것을 확인할 수 있다. 예상 값은 2를 1bit >>> 하여 1의 값을 예상하였지만, 이 또한 하드웨어의 구성 오류로 인하여 예상 값과 다르게 나왔다.

- **ANDI**



이는 XORI를 이용해 값을 넣은 후, ANDI instruction을 입력한 화면이다. Rs register에 00001, rt register에 00100을 넣고 immediate값을 0xFFFF로 입력하였는데, ALU에 ANDI동작을 하기 위한 부분이 없기 때문에 제대로 출력되지 않은 것을 확인할 수 있다. 예상 값은 rs값이 충분히 작기 때문에 rs register의 값인 00001이 나올 것을 예상했지만, 하드웨어의 오류로 예상 값과 다르게 나오게 된 34다.

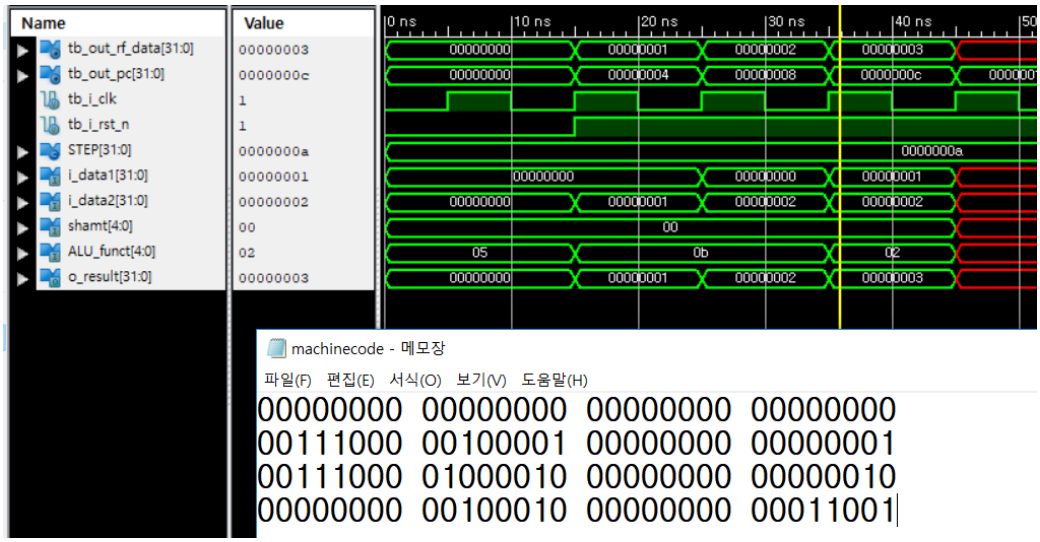
- **BEQ, BNE**



이는 XORI를 이용해 값을 reg에 넣은 후 BEQ와 BNE instruction을 차례로 입력한 화면이다. BEQ엔 rs와 rt를 00001로 같은 register 값을 넣었고, BNE엔 rs에 00001, rt에 00010을 넣어 다른 값을 입력하였다. 결과 값은 BEQ에 동작을 하지 않아 주소 값만 변경되는 것을 확인할 수 있고, BNE에서 정상적으로 동작하게 되어 마지막 주소 값이 34가 되는 것을 확인할 수 있다.

예상 결과 값은 BEQ만 작동하고 BNE는 작동하지 않는다고 예상했지만 하드웨어가 zero flag를 반대로 설계하여 BEQ는 작동하지 않고 BNE만 정상적으로 작동하게 된다.

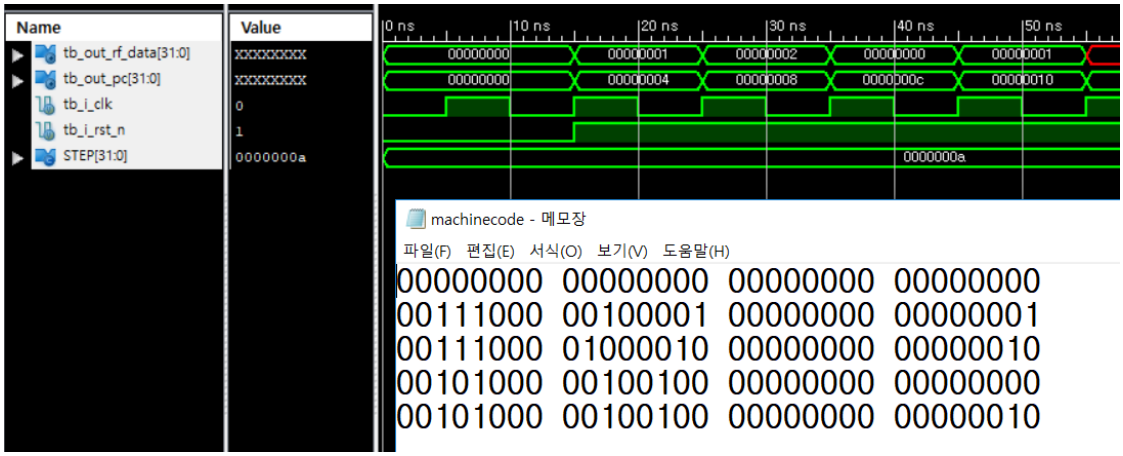
- **MULTU**



이는 XORI를 이용해 reg에 값을 넣은 후 MULTU instruction을 입력한 화면이다. ALU에 MULTU동작을 하는 부분이 없기 때문에 ALU_funct를 보면 의도하지 않은 동작을 하는 것을 확인할 수 있고 결과 값으로 3이 나오게 되었다.

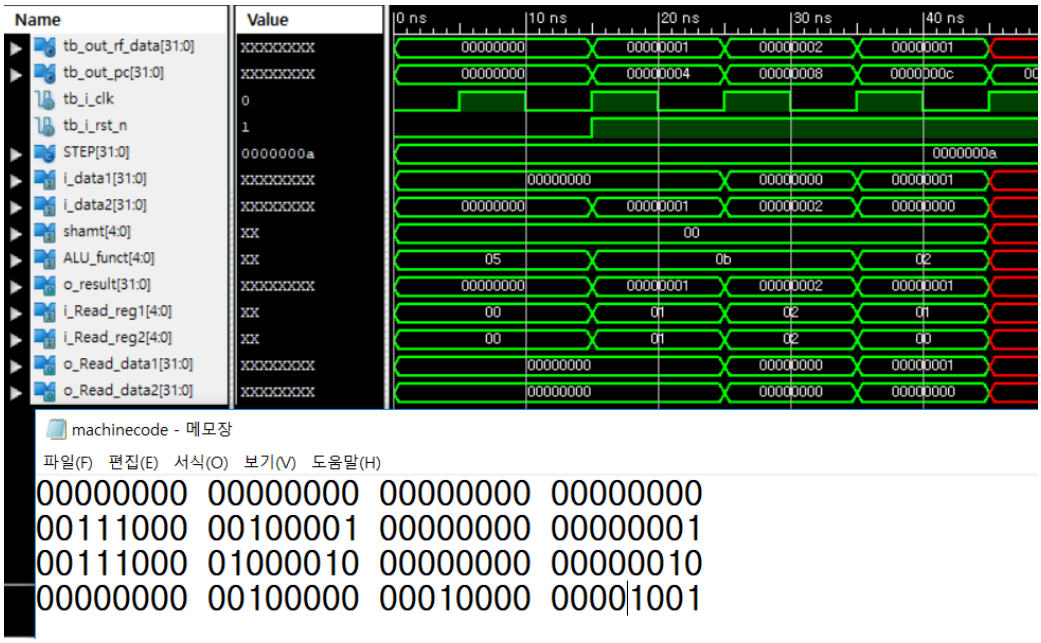
예상 값은 unsigned mult이기에 2를 예상했지만 다른 값이 나왔다.

- **SLTI**



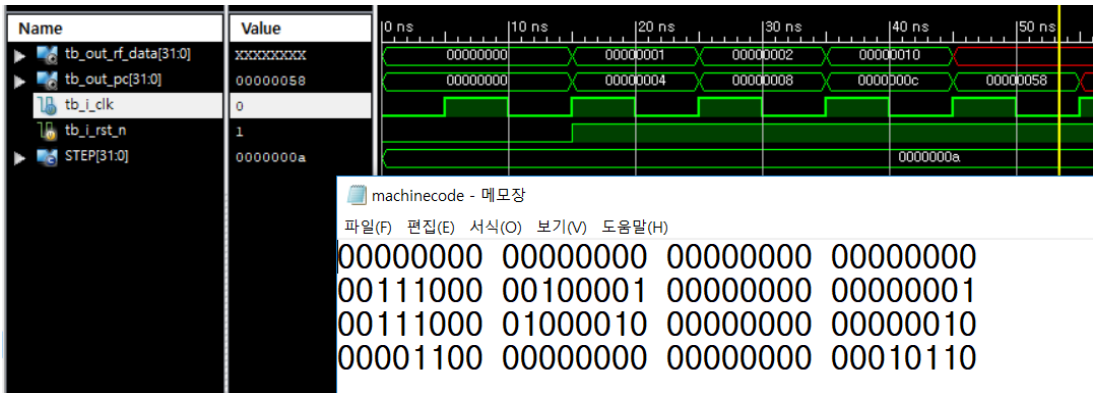
이는 SLTI의 두 가지 경우를 입력한 화면이다. 먼저 입력한 instruction은 00001에 저장된 값 1과 immediate값 0을 비교하여 register의 값이 크기 때문에 0을 출력하는 것을 확인할 수 있으며 이후 입력한 instruction은 1과 immediate값 2를 비교하였다. 1이 2보다 작기 때문에 출력으로 1이 Set된 것을 확인할 수 있다.

- **JALR**



이는 JALR 명령어이며 00010 register에 PC+4를 저장하고 00001 register의 값을 PC에 넣어야 한다. 하지만 JALR을 사용할 수 없게 회로가 구성되어 있기 때문에 Jump를 하지 않아 정상적인 작동을 하지 않은 것을 확인할 수 있다. 예상 값은 rd register에 pc+4의 값을 저장하고 RS값으로 JUMP하는 것이었지만 실험 결과는 다르게 나왔다.

- **JAL**



이는 JAL 명령어이며 31번 Register에 PC+4를 저장하고, 나머지 26개의 bit를 사용하여 연산을 진행한다. 26bit를 2bit shift한 후 instruction의 상위 4비트를 합쳐 32bit를 만들어 해당 주소로 Jump한다. 따라서 output을 보면 rf_data가 10인 부분이 있고 이는 31 register의 값이다. 그 다음 0x00000058번으로 정상적으로 이동이 된 것을 확인할 수 있으며 예상 값과 같게 나왔다.