컴퓨터 구조

2차 프로젝트 multi cycle

학 과: 컴퓨터정보공학부

담당교수: 이성원교수님

학 번: 2017202087

성 명: 홍 세 정

1. 문제의 해석 및 해결 방향

구현해야 할 명령어(BGE, ANDI, LW, LWAI, JAL, DIV4, SLLV, BLTZ, XOR)의 기능과 동작을 설명한다.

- BGE

Type, opcode(func)	I-type, opcode : 000100
명령어 구조	BGE rt, rs, imme 000100 / rs(5) / rt(5) / imme(16)
명령어 설명	rs값이 rt보다 크거나 같으면 pc + 4 + (imme<<2)을 수행한다. IF : 명령어를 access하고(IR = MEM[pc]), pc = pc+4를 수행한다. ID : 명령어에 맞게 rs와 rt를 선택한다. EX : alu control에서 sge를 선택하여 rs와 rt의 값을 비교한다. MEM : EX에서 비교한 rs, rt의 값을 보고 imme <<2를 더해준다.

- ANDI

Type, opcode(func)	I-type, opcode : 001000
명령어 구조	ANDI rt, rs, imme 001000 / rs(5) / rt(5) / imme(16)
명령어 설명	and 연산을 하는 명령어이다. rt = rs (and) imme IF : 명령어를 access하고(IR = MEM[pc]), pc = pc+4를 수행한다. ID : 명령어에 맞게 control한다. EX : rs와 imme의 연산을 한다. MEM : ALU에서 rs imme and계산한다. WB : and연산된 결과를 rt에 저장.

- LW

Type, opcode(func)	I-type, opcode : 100011
명령어 구조	lw rt, imme(rs) 100011 / rs(5) / rt(5) / imme(16)
명령어 설명	Rs의 값을 가져와 +imme address해서 rt에 저장한다. Rt = MEM[rs+imme] IF : 명령어를 access하고(IR = MEM[pc]), pc = pc+4를 수행한다. ID : 명령어에 맞게 control한다. A=rs EX : rs와 imme를 add 계산한다. MEM : memory data register에 memory[alu연산결과]를 저장한다. WB : rt에 memory data register을 저장한다.

- LWAI

Type, opcode(func)	R-type, opcode : 001110
명령어 구조	lwai rd, rs, rt 001110/ rd(5) / rs(5) / rt(5)
명령어 설명	Lwai는 원래는 없는 명령어이므로 따로 구현해주어야 한다. Load와 비슷한 기능을 하지만 해줘야 할 기능이 더 많기 때문에 clock이 더 많이 들어가게 된다. rd = MEM[rs+rt], rt = rt+4 rs와 rt를 더해 rd에 저장하고(), rt는 rt+4의 값을 넣어준다. IF : 명령어를 access하고(IR = MEM[pc]), pc = pc+4를 수행한다. ID(rs+rt) : 명령어에 맞게 control한다. A=rs B=rt ID(rt) : A = rt, B = 4 EX1 : rd에 저장하기 위한 rs, rt를 연산을 수행한다. EX2 : rtd에 저장하기 위해 rt+4 연산을 수행한다. MEM1 : memory data register에 memory[alu연산결과]를 저장한다. MEM2 : rt에 EX2연산을 저장한다. WB : rd에 memory data register(MEM1값)r을 저장한다.

- JAL

Type, opcode(func)	J-type, opcode : 000011
명령어 구조	JAL imme 000011 / imme(26)
명령어 설명	Imme의 address로 jump하고,ra(31번째 레지스터)에 pc +4의 값을 저장한다. Pc = pc +4 + (imme <<2) , ra = pc+4 IF : 명령어를 access하고(IR = MEM[pc]), pc = pc+4를 수행한다. ID : 명령어에 맞게 control한다. EX : pc를 업데이트한다. MEM : ra 값에 pc+4의 값을 저장한다.

- DIV4

Type, opcode(func)	I-type, opcode : 011011
명령어 구조	DIV4 rt, rs, 4 011011 / rs(5) / rt(5) / 0000 0000 0000 0100
명령어 설명	Rt = rs /4 Rs의 값을 나누기 4한 후 rt에 저장한다. 원래는 없는 명령어이므로 조건에 맞게 우리가 구현해주어야 한다. IF : 명령어를 access하고(IR = MEM[pc]), pc = pc+4를 수행한다. ID : 명령어에 맞게 control한다. A=rs EX : ALU연산 rs/4의 연산을 한다. MEM : 연산 결과를 rt에 저장할 수 있다.

- SLLV

Type, opcode(func)	R-type, opcode : 000000 (funct:000100)
명령어 구조	SLLV rd, rt, rs 000000 / rd(5) / rt(5) / rs(5) / shamt(5) / 000100 -> 보통은 rs를 먼저 써주는데 SLLV명령어는 rt를 먼저 써준다.
명령어 설명	Rd = rt <<rs Rt를 움직이는 rs만큼 shift한다. IF : 명령어를 access하고(IR = MEM[pc]), pc = pc+4를 수행한다. ID : 명령어에 맞게 control한다. A=rs, B=rt EX : ALU연산 rt << rs 연산을 해준다. MEM : 연산 결과를 rd에 저장한다.

- BLTZ

Type, opcode(func)	I-type, opcode : 000001
명령어 구조	BLTZ rs, imme (rt=00000) 000001 / 00000rt(5) / rs(5) / imme(16)
명령어 설명	rs값이 imme보다 작으면 pc + 4 + (imme<<2) IF : 명령어를 access하고(IR = MEM[pc]), pc = pc+4를 수행한다. ID : 명령어에 맞게 control한다. A = rs, ALUOut 값을 연산해둔다. EX : rs와 imme 값을 slt연산을 통해 연산한다.

- XOR

Type, opcode(func)	R-type, opcode : 000000 (func:100110)
명령어 구조	XOR rd, rs, rt 000000 / rd(5) / rs(5) / rt(5) / shamt(5) / 100110
명령어 설명	Rd = rs ^ rt Rs와 rt를 xor한 값을 rd에 저장한다. IF : 명령어를 access하고(IR = MEM[pc]), pc = pc+4를 수행한다. ID : 명령어에 맞게 control한다. A=rs, B=rt EX : ALU연산 rs, rt연산을 해준다. MEM : 연산 결과를 rd에 저장한다.

문제점 해결 방향

- BGE, BLTZ

  BGE와 BLTZ 명령어는 branch 명령어이다. 조건을 따져서 pc를 이동한다. 맞는 ALUop를 선택하여 BGE는 rt와 rs를 비교하고, BLT는 rs와 imme와 비교하여 zero 값이 나오면 이동하도록 구현할 수 있었다. 따라서 ID(instruction decode)에서 BGE는 A=rs, B=rt를 선택, BLTZ는 A=rs, B=imme 값을 선택하고 EX 동작에서 A와 B를 비교하여 이동할 수 있도록 조건문을 만들어주었다.

- ANDI, DIV4

  다음 두개의 명령어는 I-type에서 단순 연산을 해주는 명령어이다. Execution을 수행할 때 A=rs, B=imme를 선택하여 각 명령어에 맞는 ALUop(ANDI:011, DIV4:111)를 선택하여 계산할 수 있도록 구현하였다. ALUout값을 rt에 저장하는 것은 같은 control을 사용하므로 연산을 한 이후에는 같은 state로 이동하여 rt에 값을 업데이트할 수 있었다.

  DIV4의 명령어는 나누기 4하는 명령어이다. 기존 기능에는 나누기하는 명령어가 없기 때문에 DIV4 명령을 받으면 나누기를 실행하는 ALUcontrol(4'b1111)을 만들어 주었다.

- LW, LWAI

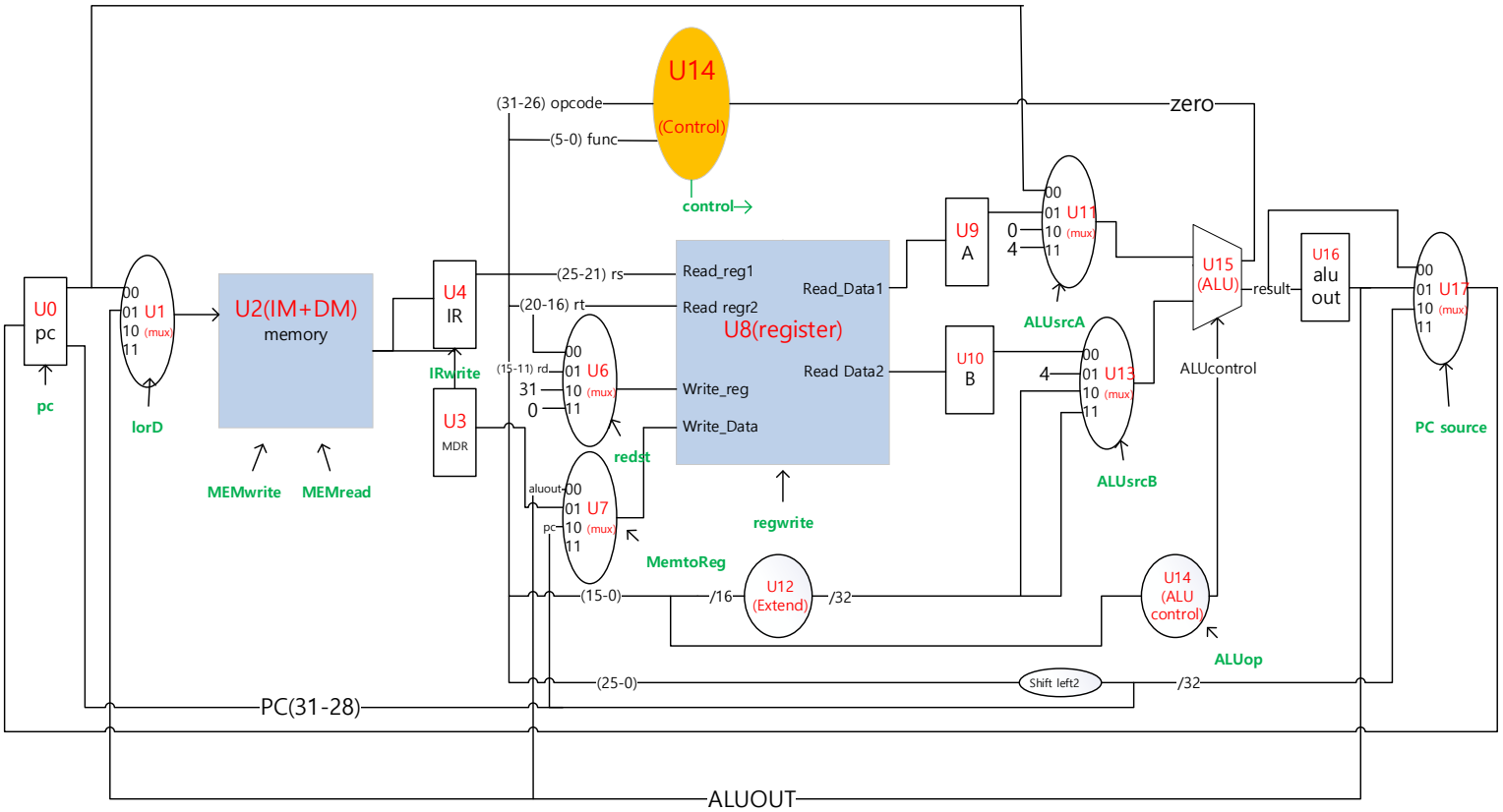
  Load를 하는 명령어이다. LW는 기존 load명령어를 통해 구현 가능하다. LWAI는 기본적으로 load하는 것과 같은 기능을 하지만 rs+rt를 Memory address를 한다. 따라서 그에 맞게 ALUSrcA, ALUSrcB를 선택하고 load를 할 수 있다. 그 후 rt = rt + 4명령어를 실행하기 위해 state를 추가한다. 구현하기 위해 A = 11, B = 00을 선택하여 rt의 값을 업데이트할 수 있다.

- JAL

  jump하는 명령어이다. 기본 jump하는 명령어에 ra에 pc + 4를 저장하는 state를 추가하여 구현할 수 있다. 31번째 레지스터는 RegDst = 10으로 가져올 수 있으며 MemtoReg = 10 (pc+4)을 write할 수 있다.

- SLLV, XOR

  두 명령어는 R-type 명령어이다. 기존 R-type명령어에 function에 따라서 각 ALU control을 조정하여 명령어를 구현할 수 있다. SLLV명령어는 rt<<rs 만큼 shift하는 연산을 ALU module에서 동작하도록 만들어 주었다. (o_result <= (i_data2 << i_data1);)

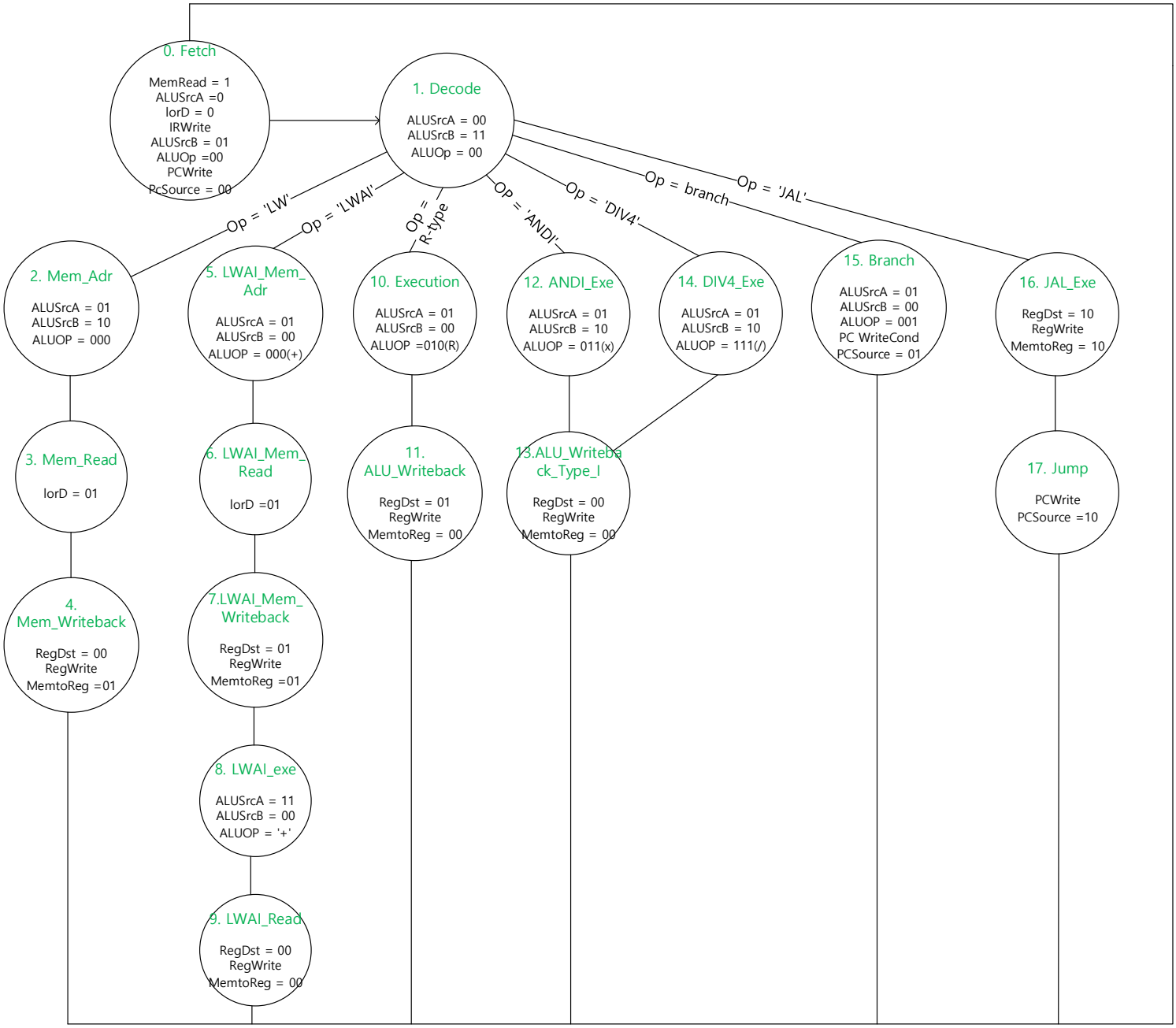


Multicycle를 topology diagram으로 나타낸 것이다.

총 17개의 모듈을 사용하여 multicycle을 구현할 수 있다. 다음 diagram에서는 cycle이 필요한 module이 있고 필요하지 않은 module이 있다. Clock이 필요한 모듈(U0, U2, U4, U8, U9, U10, U16)을 지나갈 때마다 clock주기가 하나씩 늘어난다.

초록색으로 들어가는 input은 U14(control)에서 나오는 input이다. 다음과 같은 control을 명령어에 맞게 조정하여 명령어를 구현할 수 있다.

2. 설계 의도와 방법



다음과 같이 FSM을 설계하고 module을 구현할 수 있었다.

총 19개의 state로 이번 과제를 구현할 수 있었다.

18번 state는 모든 control을 reset시키는 Reset state를 만들 수 있었다.

모든 명령어는 기본적으로 IF(state. 0)와 ID(state .1)을 수행하고 그 후 state는 명령어에 따라서 움직이게 된다.

명령어를 state순으로 나열해보면 다음과 같다.

LW : 0 – 1 – 2 – 3 – 4 - 18

LWAI : 0 – 1 – 5 – 6 – 7 – 8 – 9 -18

R-type(SLLV, XOR) : 0 – 1 – 10 – 11 – 18

ANDI : 0 – 1 – 12 – 13 – 18

DIV4 : 0 – 1 – 14 – 13 – 18

Branch(BGE, BLTZ) : 0 – 1 – 15 – 18

JAL : 0 – 1 – 16 – 17 – 18

0. Fetch

MemRead = 1 // 32bit 명령어를 읽어온다.

ALUSrcA = 0 // pc를 선택

ALUSrcB = 1 // 4를 선택

ALUOp = 000 // + 연산을 수행

IRWrite = 1 // IR = Memory[pc]를 수행

1. Decode

ALUSrcA = 0 // pc를 선택

ALUSrcB = 1 // imme를 선택

ALUOp = 00 // + 연산을 수행. Branch 값을 미리 만들어준다.

2. Mem_Adr

ALUSrcA = 01 // rs를 선택

ALUSrcB = 10 // imme를 선택

ALUOp = 000 // + 연산을 수행. Memory address한다.

3. Mem_Read

IorD = 01 // load하기 위해 ALU의 연산 값을 가져온다.

4. Mem_Writeback

RegDst = 00 // rt를 선택

MemtoReg = 01 // Memory address 한 값을 선택

RegWrite = 1 // rt에 write한다.

5. LWAI_Mem_Adr

ALUSrcA = 01 // rs를 선택

ALUSrcB = 00 // rt를 선택

Mem_Adr과 같은 역할이지만 A와 B 선택만 다르게 해준다.

6. LWAI_Mem_Read

IorD = 01 // load하기 위해 ALU의 연산 값을 가져온다.

State3와 같은 역할.

7. LWAI_Mem_Writeback

RegDst = 01 // rd를 선택

MemtoReg = 01 // Memory address 한 값을 선택

RegWrite = 1 // rt에 write한다.

State4.와 같은 역할 RegDst만 다르게 선택해준다.

8. LWAI_exe 9. LWAI_Read

Rt = rt+4를 해주기 위해서 A = 11, B = 00을 선택해서 RegWrite할 수 있다.

10. Execution 11. ALU_Writeback

ALUSrcA = 01 // rs를 선택

ALUSrcB = 00 // rt를 선택

ALUOp = 010 // R-type function을 보고 ALUcontrol을 판단.

12. ANDI_Exe 14. DIV4.Exe

ALUSrcA = 01 // rs를 선택

ALUSrcB = 10 // imme를 선택

ALUOp = 011(ANDI), ALUOp = 111(ANDI) // 다음과 같은 연산을 수행.

13. ALU_Writeback_Type_I

RegDst = 00 // rt를 선택

MemtoReg = 00 // ALUout 값을 선택

RegWrite = 1 // rt에 write한다.

15. Banch

ALUSrcA = 01 // rs

ALUSrcB = // BGE는 rt값을 선택하고 BLTZ는 imme값을 선택한다.

ALUOP = 100(BLTZ), ALUOP=101(BGE) // sge, slt연산을 통해서 같은지 판단.

PC WriteCond

PCSource = 01

16. JAL_Exe

RegDst = 10 //ra선택

RegWrite // write

MemtoReg = 10 //pc 선택

17. Jump

PCWrite //해당 imme address로 jump한다.

PCSource =10

a) 어셈블리 코드

00000000 00000000 00000000 00000000

00010000 01100010 00000000 00000000

//BGE $v0, $v1, 0

00100000 01001000 00000000 00000001

//ANDI $s0 $v0, 1

10001100 01001000 00000000 00000100

//LW $s0, 4($v0)

00111000 01000011 01000000 00000000

//LWAI $s0, $v0, $v1

00001100 00000000 00000000 00000000

//JAL 10

01101100 01001000 00000000 00000100

//DIV4 $s0 &v0. 4

00000000 01000011 01000000 00000100

//SLLV $s0, $v1, $v0

00000100 01000000 00000000 00000000

//BLTZ $v0, 0

00000000 01000011 01000000 00100110

//XOR $s0, $v0, $v1

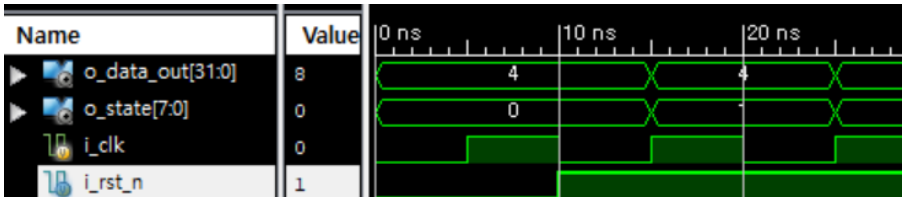
v0 = 2 = 00010

v1 = 3 = 00011

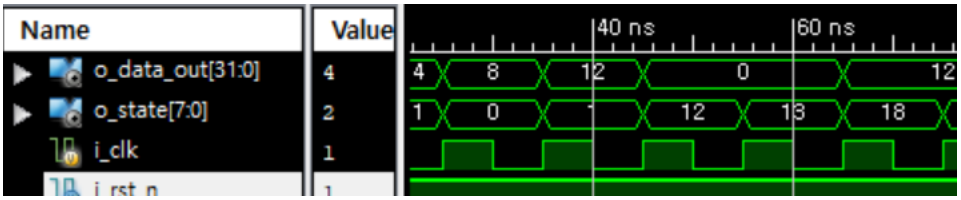
s0 = 16 = 01000

3. 결과

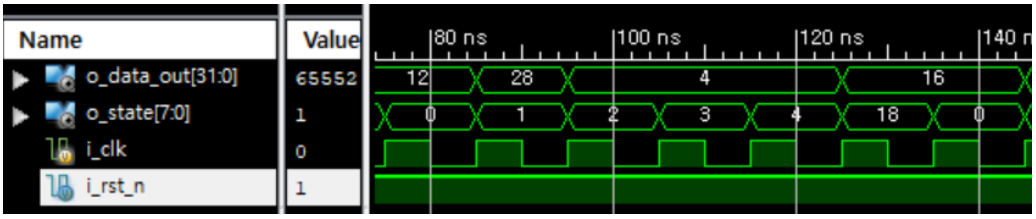
- 초기상태



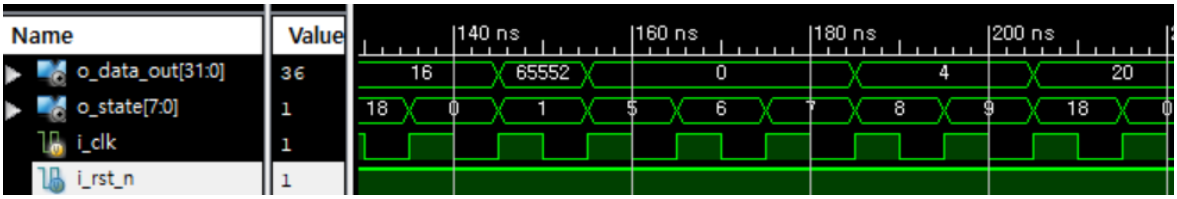
- ANDI



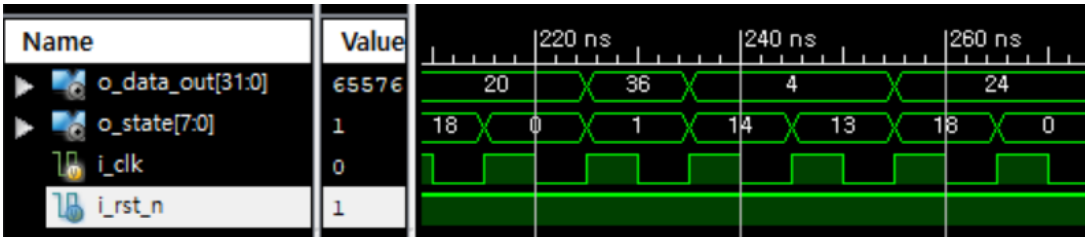
- LW



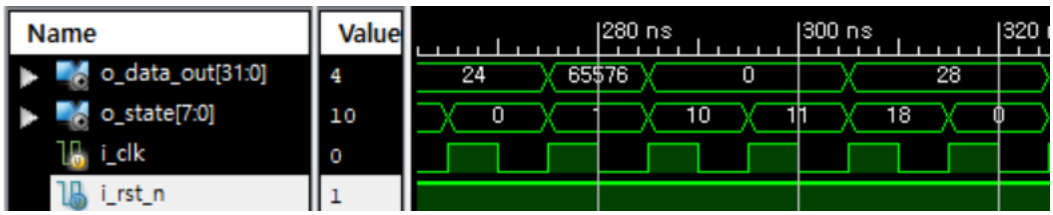
- LWAI



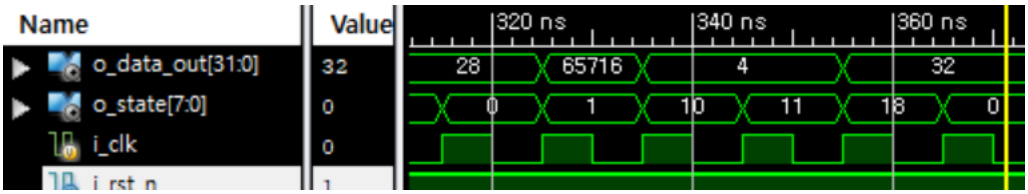
- DIV4



- SLLV

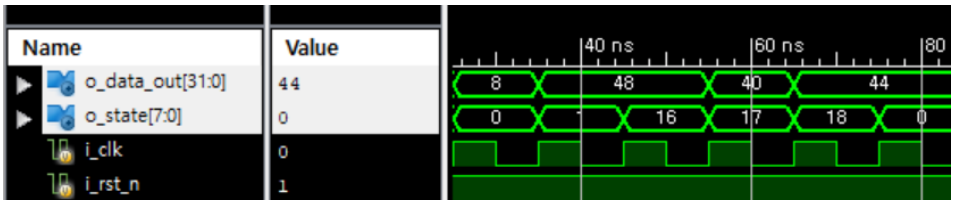


- XOR



- Branch

- Jump



명령어에 따라서 state가 움직이는 것을 확인할 수 있다.

4. 고찰

이번 프로젝트는 multicycle을 구현하는 하드웨어를 동작시키는 과제였다. 이번 과제에서도 많은 오류와 새로 배운 것 들이 있다.

FSM을 구현할 때 main FSM에서 state를 만들고 FSM을 구현하는 것이었지만, main FSM에서 구현하였을 때는 memory 값을 읽어오지 못하였다. 그래서 Top module에서 MyFSM이 아닌 main FSM에 실행되도록 바꿔주었더니, main FSM module이 실행될 수 있었다.

실행했을 때 Non-blocking, blocking이라는 오류가 난 것을 보았다. 다음 오류는 non_blocking(=), blocking(<=) 둘 중 한가지로만 input을 입력해주라는 뜻이라는 것을 알 수 있었다.