RISC-V Multi-Cycle Processor

구현 보고서

과목 : 컴퓨터구조

학번 : 2023320132, 2023320032

이름 : 이 성민, 최 주영

목차

1. FSM Design


IF
ALUSrcA : 00
ALUSrcB : 10
IRWrite : 1
PCWrite : 1
ResultSrc : 00
Decode
ALUSrcA : 01
ALUSrcB : 01
Op = I-Load || S-type
Op = R-type
Op = I-Arith
Op = I-Lui
Op = J-Jal
Op = I-Jalr
Op = B-type
ExLS
ALUSrcA : 10
ALUSrcB : 01
ResultSrc : 11
ExR
ALUSrcA : 10
ALUSrcB : 00
ExI
ALUSrcA : 10
ALUSrcB : 01
ExLui
ALUSrcA : 11
ALUSrcB : 01
ExJal
ALUSrcA : 01
ALUSrcB : 10
PCWrite : 1
ResultSrc : 10
ExJalr
ALUSrcA : 10
ALUSrcB : 01
PCWrite : 1
ResultSrc : 00
ExB
ALUSrcA : 10
ALUSrcB : 00
ResultSrc : 10
branch : 1
Op = I-Load
Op = S-type
Mem_Read
MemWrite : 0
ResultSrc : 10
Mem_Write
MemWrite : 1
ResultSrc : 10
WReg
RegWrite : 1
ResultSrc : 10
WRegJr
ALUSrcA : 01
ALUSrcB : 10
ResultSrc : 00
RegWrite : 1
WRegL
ResultSrc : 01
RegWrite : 1
*Op : Opcode
*I : I-type
*U : U-type
*J : J-type
*Default Control signal: 1'b0 || 2'b0

2. FSM State Definition

IF state에서 시작하여 여러 state을 거친 후, 다시 IF state로 돌아오도록 FSM을 design하였으며, 각각의 state에서 표기되어 있지 않은 control signal은 signal을 선언한 변수의 크기에 따라 1'b0 또는 2'b0로 할당되어 있다.

해당구현에서 사용되는 control signal은 <ALUSrcA[1:0], ALUSrcB[1:0], IRWrite, PCWrite, ResultSrc[1:0], RegWrite, MemWrite, branch>, 총 8개의 signal이 사용된다.

구현은 아래의 Fig1.을 참고하여 구현하였다. 항상 회로를 따라 여러 값들이 입력으로 들어오지만, 이 값들이 Memory, Register, PC update 등에 사용되는지 여부는 control signal이 조절한다.

Multi-Cycle Processor

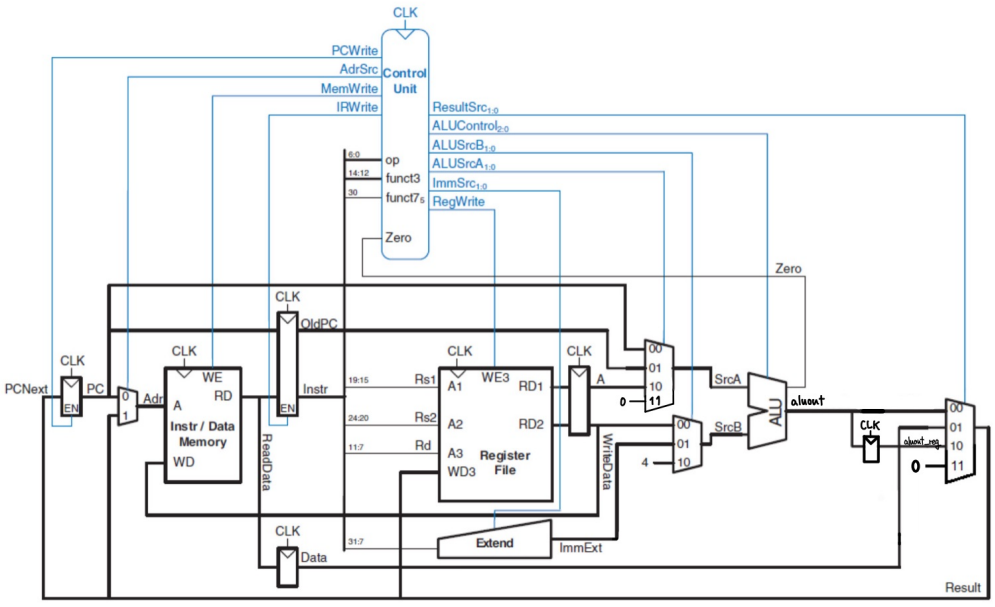

Fig1. Multicycle 회로(TF 강의자료 참고)

각 명령어(R-type, I-Arith, I-Load, I-Jalr, S-type, B-type, U-Lui, J-Jal)에 대한 state 전이는 다음과 같다.

OP_R	IF – Decode – ExR – WReg - IF
OP_I_Arith	IF – Decode – ExI – WReg - IF
OP_I_Load	IF – Decode – ExLS – Mem_Read – WRegL - IF
OP_I_JALR	IF – Decode – ExJalr – WRegJr - IF
OP_S	IF – Decode – ExLS – Mem_Write - IF

OP_B	IF – Decode – ExB - IF
OP_U_LUI	IF – Decode – ExLui – WReg - IF
OP_J_JAL	IF – Decode – ExJal – WReg - IF

A. <IF> state

instruction fetch 단계로 PC을 PC + 4로 설정하고, 다음 단계에서 사용하기 위해 intruction을 inst_reg 레지스터에, pc값을 OldPC 레지스터에 저장한다.

B. <Decode> state

Instruction을 decode하는 단계로 B-type 또는 J-Jal 명령어에서 target address로 사용할 주소 값을 계산하여 aluout_reg 레지스터에 저장한다. 추가적으로 rs1_data과 rs2를 각각의 레지스터에 할당해야 하지만, 해당 구현에서는 기본적으로 negative edge일 때마다 할당하도록 구현되어 있다.[1]

C. <ExR> state

R-type 명령어를 처리할 때 거치는 단계로, 다음 단계에서 레지스터에 쓸 데이터(rs1 + rs2)을 계산하여 aluout_reg 레지스터에 저장한다.

D. <ExI> state

I-Arith 명령어를 처리할 때 거치는 단계로, 다음 단계에서 레지스터에 쓸 데이터(rs1 + imm)을 계산하여 aluout_reg 레지스터에 저장한다.

E. <ExLui> state

U-Lui 명령어를 처리할 때 거치는 단계로, 다음 단계에서 레지스터에 쓸 데이터(0 + imm)를 계산하여 aluout_reg 레지스터에 저장한다. 다른 type들과의 통일성을 위해 0 + imm[2]의 값을 계산한다.

F. <ExJal> state

J-Jal 명령어를 처리할 때 거치는 단계로, 다음 단계에서 레지스터에 쓸 데이터(OldPC + 4)를 계산하여 aluout_reg에 저장한다. 더불어 Decode 단계에서 계산

[1] rs1_data와 rs2_data는 한 명령어에서 불변이기에, 해당 data을 할당 받은 레지스터의 값을 reset 신호가 활성화 되었을 때만 0으로 설정하고, 이외의 경우 매 클럭마다 data을 저장해준다.

[2] imm값은, 하위 12비트를 0으로 채우고 상위 20비트에 즉시 값을 할당한 값으로 설정되어 있다.

한 target address가 저장되어 있는 aluout_reg 값을 PC에 할당한다.

G. <WReg> state

aluout_reg에 저장되어 있는 값을 rd 레지스터에 저장한다.

H. <ExLS> state

I-Load와 S-type 명령어 즉, lw와 sw을 수행할 때 다음단계에서 메모리에 접근할 주소를 계산하여 계산한 주소 값을 aluout_reg 레지스터에 저장하는 단계이다.

*memory 주소를 계산하는 단계로 ResultSrc의 값이 사용되지 않기 때문에 ResultSrc 신호 값을 11로 설정하여 상수 0값을 rd_data에 할당한다.[3]

I. <MemRead> state

I-Load 명령어를 수행할 때 거치는 단계로, 다음단계에서 레지스터에 데이터를 쓰기 위해 현재 aluout_reg에 저장되어 있는 주소 값(rs1 + imm)을 선택하여 해당 주소에 해당하는 메모리 값을 읽어 MemRdata_reg에 저장한다.

J. <WRegL> state

I-Load 명령어를 수행할 때 거치는 단계로, 메모리에서 읽어온 데이터인 MemRdata[4]을 rd 레지스터에 저장한다.

K. <MemWrite> state

S-type 명령어를 수행할 때 거치는 단계로, 현재 aluout_reg에 저장되어 있는 주소값(rs1 + imm)을 선택하여 해당 메모리 주소에 MemWdata 값(rs2 값을 MemWdata에 할당)을 저장한다.

L. <ExJalr> state

J-Jal 명령어를 처리할 때 거치는 단계로, Jal 명령에서 사용할 target address(rs1 + imm)를 계산하고 해당 값을 사용해 PC을 바로 target address로 설정한다.

M. <WRegJr> state[5]

[3] ExLS 단계에서 ResultSrc을 11로 설정하여 rd_data을 0로 설정하지 않고 00으로 설정하여 rd_data에 aluout이 저장될 때, 보드에서의 타이머 상승 속도가 반으로 줄어드는 현상이 발생한다. 시뮬레이션 상에서도 처음 sec0로 분기해야 할 때 분기하지 않는 문제가 발생한다.

[4] <RV32I> 모듈에서 input값인 MemRdata을 MemRdata_reg에 clock negative edge마다 할당하며, MemRdata_reg값을 <datapath> 모듈의 input인 MemRdata로 할당한다.

[5] J-Jal명령어와 같이 <WReg>에서 처리하지 못하는 이유는, Jalr 명령어가 rs1 + imm값에 해당하는 주소로 분기해야 하기에 decode단계에서 계산한 target address을 사용하지 못하기 때문이다. 따라서 <ExJalr> state에서

I-Jalr 명령어를 수행할 때 거치는 단계로, OldPC + 4를 계산하고 해당 값을 rd 레지스터에 저장한다.

N. <ExB> state

B-type 명령어를 처리할 때 거치는 단계로, rs1과 rs2의 값의 차를 바탕으로 분기 여부를 결정하며, 분기 조건에 부합할 때, Decode 단계에서 계산한 target address가 저장되어 있는 aluout_reg 값을 PC에 할당한다.

3. 모듈 분석

A. <maindec> 모듈

i. 모듈의 역할

FSM을 제어한다. 프로세서는 여러 단계로 명령어를 실행하는데, 현재 상태(state)와 입력된 opcode을 기반으로 다음 상태(n_state)을 결정한다. 처음 시작은 IF단계로 하며, 다음 명령어를 가져오기 위해 다시 IF단계로 전이하여 모든 명령어를 수행할 수 있도록 구현하였다.

ii. 모듈의 입출력

input	rst	리셋 신호, 0일 때 활성화
input	clk	클럭 신호, 하강 엣지에서 현재 상태(state)에서 다음 상태(state)로 전이
output	Opcode[6:0]	명령어 type 결정 코드에 비트로 정의되어 있음
output	ALUSrcA[1:0]	ALU의 첫 번째 입력 선택, 2비트 00 : pc 01 : OldPC 10 : rs1_data_reg 11 : 내부 상수 0 (lui 명령어에서 사용)
output	ALUSrcB[1:0]	ALU의 두 번째 입력 선택, 2비트 00 : rs2_data_reg

분기할 주소를 계산하게 되고, 다음 state에서 직접적으로 OldPC + 4를 계산하도록 FSM을 디자인하였다.

		01 : imm 10 : 내부 상수 (4)
output	IRWrite	inst_reg와 OldPC값을 할당할 지 여부를 결정하는 신호(IF 에서 활성화)
output	PCWrite	PC 레지스터 쓰기 활성화 신호, JAL, JALR 에서 활성화
output	ResultSrc[1:0]	레지스터 파일에 쓰일 결과 데이터 선택, 2비트 00 : aluout(not register) 01 : MemRdata 10 : aluout_reg 11 : 0 (ExLS state에서 사용, 2.C ExLS state참고)
output	RegWrite	rd 레지스터에 값 쓰기 활성화 신호, 명령어 실행 후 결과 값을 레지스터에 저장할 때 활성화
output	MemWrite	데이터 메모리에 쓰기 활성화 신호, store 명령어에서 활성화
output	branch	분기 명령어의 조건이 참이면 활성화
output	ALUOp (register)	ALU가 수행할 연산의 기본 연산을 지정하는 신호, 1일 경우 덧셈 연산을 수행[6]

iii. 모듈 내부 로직

a. state 변수 선언

모든 state을 4비트 상수로 정의한다. 총 state는 14개(4비트로 표현 가능하다)

코드상 정의는 다음과 같다

```
localparam IF              = 4'b0000;
localparam Decode          = 4'b0001;
localparam ExR             = 4'b0010;
```

[6] 덧셈을 수행하는 것이 확실한 state일 경우 alucontrol을 미리 설정하여 <aludec> 모듈에서 조건 부합 여부를 판단할 필요가 적어진다.

```
localparam ExI          = 4'b0011;
localparam ExLui = 4'b0100;
localparam ExJal = 4'b0101;
localparam WReg    = 4'b0110;
localparam ExLS      = 4'b0111;
localparam Mem_Read    = 4'b1000;
localparam WRegL = 4'b1001;
localparam Mem_Write = 4'b1010;
localparam ExJalr = 4'b1011;
localparam WRegJr = 4'b1100;
localparam ExB      = 4'b1101;
```

b. Control signal의 분배

assign 문을 활용하여 controls을 각 control signal에 분배해준다.

ALUSrcA (2비트):	ALU의 첫 번째 입력 소스 선택(00 - PC || 01 - OldPC || 10 – rs1_data_reg || 11 – 정수 0).
ALUSrcB (2비트):	ALU의 두 번째 입력 소스 선택(00 - rs1_data_reg || 01 - imm || 10 – 정수 4 || 11 – 경우 없음(default 0)).
IRWrite (1비트):	IR(Instruction Register && OldPC register)에 쓰기 활성화 신호.
PCWrite (1비트):	PC(Program Counter)에 쓰기 활성화 신호.
ResultSrc (2비트):	데이터 선택(메모리 || PC || 레지스터에 쓸 데이터, 00 - aludata || 01 - MemRdata || aluout_reg - 10 || 11 – 정수 0).
RegWrite (1비트):	레지스터(rd)에 데이터 쓰기 활성화 신호.
MemWrite (1비트):	메모리에 데이터 쓰기 활성화 신호.
branch (1비트):	B-type 명령어일 때 활성화

c. state 로직 및 전이 로직

clock negative edge 또는 reset negative edge마다 state을 n_state로 할당(상태 전이)하고, reset 신호가 활성화 되면, state을 IF단계로 설정한다. 하단에 명시되어 있지 않은 control signal은 해당 state에서 활성화되어 있지 않은 값이기에 1비트 0 또는 2비트 00으로 할당되어 있다. 또한 ALUOp는 alucontrol signal을 미리 표기해 놓은 값으로 add연산을 해야

하는 것이 확실한 state에서 미리 ALU operation을 결정하여 <aludec> 모듈에서 불필요한 조건문 사용이 없도록 한다.

IF	
ALUSrcA = 00	PC+4(32비트)의 연산을 위해 PC값 선택
ALUSrcB = 10	PC+4(32비트)의 연산을 위해 내부상수 4값 선택
IRWrite = 1	Inst_reg에 inst을 저장, OldPC에 pc 값 저장
PCWrite = 1	PC 업데이트 활성화
ResultSrc = 00	alu연산 결과를 바로 사용하기 위해 aluout(not register)을 선택
ALUOp = 1	PC + 4을 위해 add연산자 선택

➔ alu 연산결과로 pc을 바로 업데이트하며, 다른 state에서 사용할 instruction와 pc값을 레지스터에 저장한다.

Decode	
ALUSrcA = 01	OldPC + imm 연산을 위해 OldPC값 선택
ALUSrcB = 01	OldPC + imm 연산을 위해 imm값 선택
ALUOp = 1	OldPC + imm을 위해 add연산자 선택

➔ 분기 주소로 사용할 값을 ALU 연산을 하여 계산하고 다음 state (ExB || ExJal) 에서 사용할 수 있도록 aluout_reg에 저장한다.

ExR	
ALUSrcA = 10	rs1_data_reg과 rs2_data_reg의 ALU 연산을 위해 rs1_data_reg 선택
ALUSrcB = 00	rs1_data_reg과 rs2_data_reg의 ALU 연산을 위해 rs2_data_reg 선택

➔ rd 레지스터에 저장할 값을 ALU 연산을 하여 계산하고 다음 state(WReg)에서 저장에 사용할 수 있도록 aluout_reg에 저장한다.

ExI	
ALUSrcA = 10	rs1_data_reg과 imm 값의 ALU 연산을 위해 rs1_data_reg 선택

ALUSrcB = 01	rs1_data_reg과 imm 값의 ALU 연산을 위해 imm 선택

➔ rd 레지스터에 저장할 값을 ALU 연산을 하여 계산하고 다음 state(WReg)에서 저장에 사용할 수 있도록 aluout_reg에 저장한다.

ExLui	
ALUSrcA = 11	0 + auipc_lui_imm ALU 연산을 위해 0 (32비트)을 선택
ALUSrcB = 01	0 + auipc_lui_imm ALU 연산을 위해 lui imm을 선택(하위 12비트 0, 상위 20비트 lui_imm으로 설정되어 있음)
ALUOp = 1	0 + auipc_lui_imm ALU 연산을 위해 add연산자 선택

➔ rd 레지스터에 저장할 값을 ALU 연산을 하여 계산하고 다음 state(WReg)에서 저장에 사용할 수 있도록 aluout_reg에 저장한다.

ExJal	
ALUSrcA = 01	jump 수행 후, 돌아올 주소 OldPC + 4을 rd에 저장하기 위해 OldPC[7] 값 선택
ALUSrcB = 10	돌아올 주소 OldPC+4를 계산하기 위해 4 선택
PCWrite = 1	PC값을 Decode 단계에서 계산한 OldPC+imm로 설정
ResultSrc = 10	Decode 단계에서 저장해 놓은 OldPC+imm을 선택
ALUOp = 1	OldPC + imm을 위해 add연산자 선택

➔ rd 레지스터에 저장할 값을 ALU 연산을 하여 계산하고 다음 state(WReg)에서 저장에 사용할 수 있도록 aluout_reg에 저장한다. 또한, Decode state에서 계산한 jump주소값이 저장되어 있는 aluout_reg의 값을 바탕으로 pc을 update한다.

WReg	
ResultSrc = 10	aluout_reg 레지스터에 저장된 값을 rd 레지스터에

[7] 현재 pc값을 저장하고 있는 레지스터

	저장
RegWrite = 1	aluout_reg의 값을 rd 레지스터에 저장하기 위해 설정됨

➔ aluout_reg에 저장되어 있는 값을 rd 레지스터에 저장한다.

ExLS	
ALUSrcA = 10	Base address + offset ALU 연산을 위한 rs1_data_reg(base address) 선택
ALUSrcB = 01	Base address + offset ALU 연산을 위한 imm 값 (offset) 선택
ResultSrc = 11	기본적으로 해당신호는 PC write이나, memory access register write을 하지 않기 때문에, 필요가 없는 신호이다. 다만, rd_data을 지속적으로 저장하게 되는데, 이때 해당신호 값이 00이면 aluout을 저장하게 된다. 이 경우, 보드의 타이머 상승 속도가 제공된 영상에서의 약 반정도의 빠르기로 줄어들게 되는 문제가 발생한다. 따라서, 해당 신호를 11로 설정하여 rd_data에 단순히 0의 값을 할당함으로써 불필요한 값의 할당을 방지하여 속도를 빠르게 유지한다.
ALUOp = 1	rs1_data_reg + imm을 위해 add연산자 선택

➔ 메모리 접근에 사용할 주소 값을 ALU 연산을 하여 계산하고 다음 state(Mem_Read || Mem_Write)에서 메모리 접근에 사용할 수 있도록 aluout_reg에 저장한다.

Mem_Read	
ResultSrc = 10	다음 단계에서 register에 계산된 메모리주소의 값을 쓰기 위해 ExLS 단계에서 저장해 놓은 Base address + offset 선택 -> aluout_reg aluout_reg에 해당하는 메모리 주소의 값을 읽어서 MemRdata_reg에 저장 *Memread 신호는 ~MemWrite로 정의함(<RV32I> 모듈설명 참고)

➔ aluout_reg에 저장되어 있는 주소 값을 바탕으로 메모리에 접근하여 주소에 해당하는 값을 읽어온다. (읽어온 값은 MemRdata_reg에 저장되고, 이 값은 <RV32I> 모듈에서 <datapath>모듈의 MemRdata변수에 전달된다.)

WRegL	
ResultSrc = 01	메모리에서 읽은 값을 rd 레지스터에 쓰기 위해 MemRdata 레지스터 선택
RegWrite = 1	MemRdata(MemRdata_reg 값을 <RV32I> 모듈로부터 받음)의 값을 rd 레지스터에 저장

➔ 상위 state(Mem_read)에서 읽은 메모리 data를 rd레지스터에 저장한다.

Mem_Write	
ResultSrc = 10	계산된 주소에 register(rs2_data_reg == MemWdata)의 값을 쓰기 위해 상위 state(ExLS)에서 계산한 aluout_reg (Base address + offset) 선택
MemWrite = 1	Memory에서 aluout_reg에 해당하는 주소에 rs2_data_reg(MemWdata)을 저장

➔ aluout_reg에 저장되어 있는 주소 값을 바탕으로 메모리에 접근하여 주소에 해당하는 위치에 값을 쓴다.

ExJalr	
ALUSrcA = 10	rs1 + se_Jalr_imm을 계산하기 위해 rs1_data_reg 선택
ALUSrcB = 01	rs1 + se_Jalr_imm을 계산하기 위해 se_Jalr_imm 선택
PCWrite = 1	PC을 rs1_data_reg + se_jalr_imm로 업데이트
ResultSrc = 00	해당 state에서 바로 pc값을 alu 연산 결과로 설정하기 위해 aluout을 선택
ALUOp = 1	rs1_data_reg + se_jalr_imm을 위해 add연산자 선택

➔ jump할 주소에 대한 ALU연산을 수행하고, 그 결과값인 aluout에 해당하는 주소로 바로 pc을 update한다.

WRegJr	
ALUSrcA = 01	rd 레지스터에 jump 후 돌아올 주소인 OldPC + 4(32 비트)을 계산하기 위해 OldPC 선택
ALUSrcB = 10	OldPC + 4(32비트) 계산을 위해 4(32비트) 선택
RegWrite = 1	rd에 OldPC + 4 저장
ResultSrc = 00	해당 state에서 바로 rd 레지스터에 alu 연산 결과를 저장하기 위해 aluout을 선택
ALUOp = 1	OldPC + 4을 위해 add연산자 선택

➔ jalr 명령어에서 rd에 저장할 OldPC + 4(jalr명령어의 pc)을 ALU연산하고 그 결과를 즉시 rd레지스터에 저장한다.

ExB	
ALUSrcA = 10	rs1_data_reg과 rs2_data_reg 비교를 위해 rs1_data_reg 선택 (rs1 data – rs2 data의 결과를 계산)
ALUSrcB = 00	rs1_data_reg과 rs2_data_reg 비교를 위해 rs2_data_reg 선택 (rs1 data – rs2 data의 결과를 계산)
branch = 1	분기 활성화
PCWrite = 0	만약 branch 조건이 성립하지 않으면 PCWrite는 활성화 되면 안됨, 1이라면 조건이 성립하지 않아도 활성화되어 무조건 분기를 하게 된다.
ResulSrc = 10	Decode 단계에서 계산한 oldpc + imm을 pc로 설정하기 위해 aluout_reg 레지스터를 선택 (분기 조건에 설립할 때)

➔ B-type 명령어의 조건이 참일 경우 Decode state에서 계산한 target 주소가 저장되어 있는 aluout_reg의 값으로 pc를 update한다.

각 명령어에 대한 상태 전이 로직은 다음과 같다. (2. FSM State Definition의 내용과 동일)

OP_R	IF – Decode – ExR – WReg - IF
OP_I_Arith	IF – Decode – ExI – WReg - IF
OP_I_Load	IF – Decode – ExLS – Mem_Read – WRegL - IF

OP_I_JALR	IF – Decode – ExJalr – WRegJr - IF
OP_S	IF – Decode – ExLS – Mem_Write - IF
OP_B	IF – Decode – ExB - IF
OP_U_LUI	IF – Decode – ExLui – WReg - IF
OP_J_JAL	IF – Decode – ExJal – WReg - IF

B. <datapath> 모듈

i. 모듈의 역할

회로를 직접적으로 구현한 것이며, Fig1.의 회로상 필요한 PC로직, rd_data 로직, 여러 Mux 등을 직접적으로 구현해 놓은 모듈이다. 또한 해당 모듈에서 직접적인 alu연산을 진행하지 않지만 alu연산에 필요한 여러 값들을 <alu> 모듈에 선별 제공하며, 31개의 레지스터가 정의되어 있는 <regfile> 모듈과 값들을 교환하며 레지스터의 값을 읽어오고, 기록할 data을 전달한다.

ii. 모듈의 입출력

input	clk reset_n	clock신호와 reset 신호
input	inst	명령어에 대한 machine code
input	regwrite	register write 여부에 대한 control signal(<maindec> 모듈에서 생성)
input	alucontrol	alucontrol code(<aludec> 모듈에서 생성)
input	branch	분기 여부에 대한 control signal(<maindec> 모듈에서 생성)
input	PCWrite	PC에 값을 할당할 지 여부를 결정하는 제어 신호
input	IRWrite	inst_reg와 OldPC값을 할당할 지 여부를 결정하는 신호 (IF에서 활성화)
input	ALUSrcA	Mux로, 현재 pc, oldpc, rs1_data_reg, 상수 값 0 중에 선택 (alu 첫번째 입력)

input	ALUSrcB	Mux로, rs2_data_reg, 즉시 값, 상수 값 4 중에 선택 (alu 두번째 입력
input	ResultSrc	Mux로 aluout, MemRdata, aluout_reg, 상수 값 0중에 선택
output	PC	현재 실행 중인 명령어의 주소
output	rd_data	rd 레지스터에 저장할 데이터
input	OldPC	현재 수행중인 명령어의 PC값을 저장하고 있는 레지스터
output	aluout	alu연산 결과(alu에서 생성)
output	MemWdata	memory에 쓸 데이터 rs2_data_reg로 설정
input	MemRdata	memory로부터 읽은 데이터

iii. 모듈 내부 로직

a. 값의 할당

beq_taken	branch 일 때, f3beq가 set되고, Zflag가 1일 때 설정된다.
bne_taken	branch 일 때, f3bne가 set되고, Zflag가 0일 때 설정된다.
blt_taken	branch 일 때, f3blt가 set되고, Nflag != Vflag 일 때 설정된다.
bgeu_taken	branch 일 때, f3bgeu가 set되고, Cflag가 1일 때 설정된다.
MemWdata	mem에 쓸 데이터로서 rs2_data를 할당한다.
jal_imm	instruction의inst[31],inst[19:12],inst[20],inst[30:21] 해당하는 20비트 값을 할당한다. machine code의 분배형식에 맞춰 재조립한 것이다.
se_jal_imm	{11{jal_imm[20]}},jal_imm[20:1],1'b0} 로 표현 함으로서, 상위 11비트를 signExt으로 확장하여 표현한 뒤, LSB을 0로 설정하여, left shift가 된 것과 동일한 결과를 저장하게 된다.
jalr_imm	instruction의 12비트 imm값을 저장한다.
se_jalr_imm	signExt과 leftshift을 한 값을 저장한다.(실제 leftshift 기능이 적

	용된 것은 아니지만, 동일한 결과를 보여준다.)
br_imm	{inst[31],inst[7],inst[30:25],inst[11:8]}에 해당하는 12비트 값을 할당한다. machine code의 분배형식에 맞춰 재조립한 것이다.
se_br_imm	signExt과 leftshift을 한 값을 저장한다.(실제 leftshift 기능이 적용된 것은 아니지만, 12: 1까지 저장되어 있는 값과 LSB을 0로 설정함으로써 동일한 결과를 보여준다.)
se_imm_itype	I-type의 imm값을 signExt한다. MSB을 복사하여 확장한다.
se_imm_stype	S-type의 imm값을 signExt한다. MSB을 복사하여 확장한다.
auipc_lui_imm	{inst[31:12],12'b0} 하위 12비트를 0으로 채우고 상위 20비트를 imm값으로 채운다.
rs1	instruction의 19:15의 5bits값을 저장하여, rs1의 주소를 나타낸다.
rs2	instruction의 24:20의 5bits값을 저장하여, rs2의 주소를 나타낸다.
rd	instruction의 11:7의 5bits값을 저장하여, rd의 주소를 나타낸다.
funct3	instruction의 14:12의 3bits값을 저장하여, func3 값을 나타낸다.
f3beq	funct3가 000일 때 설정한다.
f3bne	funct3가 001일 때 설정한다.
f3blt	funct3가 100일 때 설정한다.
f3bgeu	funct3가 111일 때 설정한다.

b. PC 로직

reset_n 신호가 활성화 되었을 시 pc 초기화, PCWrite 신호가 활성화 되거나 특정 분기 조건이 참이면, PC를 rd_data(pc 업데이트 필요한 값들이 ResultSrc signal에 따라 할당되어 있음)로 업데이트

c. alusrc1 로직

ALUSrcA[1:0]	ALU의 첫 번째 입력 선택, 2비트 00 : alusrc1 = pc 01 : alusrc1 = OldPC 10 : alusrc1 = rs1_data_reg 11 : alusrc1 = 32d'0 (lui 명령어 연산에서 사용)

d. alusrc2 로직

ALUSrcB[1:0]	ALU의 두 번째 입력 선택, 2비트 00 : alusrc2 = rs2_data_reg 01 : alusrc2 = imm (opcode에 따라 각 명령어에 해당하는 imm을 할당한다.) 10 : alusrc2 = 32'd4 11 : 경우 없음, default 값 0 할당

e. rd_data 로직

ResultSrc[1:0]	rd 레지스터에 쓰일 데이터 선택, 2비트 00 : rd_data[31:0] = aluout(not a register value) 01 : rd_data[31:0] = MemRdata 10 : rd_data[31:0] = aluout_reg(register value) 11 : rd_data[31:0] = 32'b0, (ExLS state에서 불필요한 rd_data 할당 방지 – 타이머 증가 속도에 영향)

f. rd1_data 레지스터의 할당 로직

clock의 하강 엣지 마다 rs1_data_reg을 always 구문이 실행되며, rs1_data로 설정하고, reset_n 신호의 하강 엣지 마다 rs1_data_reg을 0으로 설정[8]

[8] 기본적으로 register에 값을 저장하는 이유는 Clock 신호가 흐르더라도 값을 유지하기 위해 즉, 다른 state에

g. rd2_data 레지스터의 할당 로직

clock의 하강 엣지 마다 rs2_data_reg을 always 구문이 실행되며, rs2_data로 설정하고, reset_n 신호의 하강 엣지 마다 rs2_data_reg을 0으로 설정

h. aluout 레지스터의 할당 로직

clock의 하강 엣지 마다 aluout_reg을 always 구문이 실행되며, aluout로 설정하고, reset_n 신호의 하강 엣지 마다 aluout_reg을 0으로 설정

i. <regfile> 모듈과의 값 교환

<regfile> 모듈은 32개의 레지스터가 정의되어 있는 모듈로, clk신호와 regwrite신호, rs1, rs2, rd, rd_data를 받아 rd에 rd_data을 쓰거나 , <datapath> 모듈에서 사용할 rs1_data와 rs2_data를 <datapath> 모듈로 전달한다.

j. <alu> 모듈과의 값 교환

alusrc1, alusrc2, alucontrol을 받아 alu연산을 하고, 그 결과를 <datapath> 모듈의 aluout에 전달한다. 또한, B-type명령어에서 사용할 조건 플래그를 <datapath> 모듈로 전달한다.

C. <RV32I > 모듈

i. 모듈의 역할

앞선 두개의 모듈 포함 RV32I_Core의 최상위 모듈로서 clk신호, reset신호, instruction, MemRdata을 직접적으로 받으며, 해당 값들을 하위차원 모듈로 전달하고, 외부 메모리 접근에 필요한 여러 값들을 외부로 출력한다.

ii. 모듈의 입출력

input	clk reset_n	clock신호와 reset 신호
input	inst	명령어에 대한 machine code
output	pc	현재 실행 중인 명령어의 pc값
output	be	바이트 활성화 신호

서 저장된 값을 사용하기 위해 레지스터에 저장한다.

output	Memwrite	메모리 쓰기 제어 신호
output	Memread	메모리 읽기 제어 신호
output	Memaddr	메모리 참조주소
input	MemRdata	메모리에서 읽은 데이터
output	MemWdata	메모리에 쓰일 데이터

iii. 모듈 내부 로직

a. Memread 신호 할당 로직

Memread 신호는 ~MemWrite로 설정한다. 동시에 읽으면서 쓰는 작업은 존재할 수 없고, 읽는 작업은 언제나 할 수 있기 때문에 다음과 같이 할당한다.

b. Inst_reg, OldPC register 할당 로직

reset신호가 활성화 되면, inst_reg와 OldPC을 0으로 설정한다. 또한 IRWrite신호가 활성화 된다면, inst_reg <= inst로 OldPC <= pc로 설정한다. 해당과정은 IF단계에서 현재 pc값과 현재 pc값에 해당하는 명령어를 레지스터에 저장하기 위해 진행한다.

c. MemRdata register 할당 로직

reset 신호가 활성화 되면. MemRdata_reg을 0으로 설정한다. 이외의 경우 MemRdata_reg을 MemRdata로 설정한다.

d. <Controller> 모듈과의 값 교환

<maindec> 모듈과 <aludec> 모듈에서 사용할 여러 값들을 <Controller> 모듈로 전달하고 <RV32I> 모듈에서 사용할 값들을 받는다.

e. <datapath> 모듈과의 값 교환

<datapath> 모듈에서 사용할 여러 값들을 전달하고 <RV32I> 모듈에서 받는다.

*이 때 rd_data가 memaddr로 설정되어, memory에 access할 때 사용된다.[9]

4. TROUBLE SHOOTING

A. PC 로직 문제

i. 기존 코드

```
always @(negedge clk, negedge reset_n)
  begin
    if (!reset_n)
            pc <= 0;
    else if (PCWrite | beq_taken | bne_taken | blt_taken | bgeu_taken)
    begin
            if (ResultSrc == 2'b00) pc <= aluout;
            else if (ResultSrc == 2'b01) pc <= MemRdata;
            else if (ResultSrc == 2'b10) pc <= aluout_reg;
            else pc <= 32'b0;
    end
  end
```

ii. 문제점

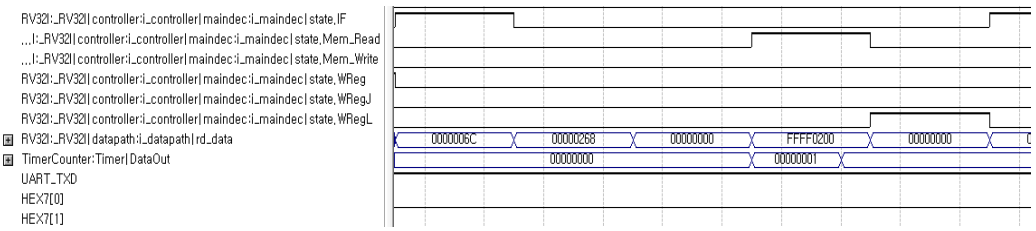

Fig2. Quartus Simulation Report – TimerCounter:Timer|DataOut의 1값 유지 문제 수정 전

기존 코드에서는 Simulation Report가 Fig2.와 같았고, DataOut을 보면 TimerCounter 모듈[10]에서 CompareR == CounterR일 때, StatusR을 1로 설정하고,

[9] rd data는 resultsrc 신호에 따라 다르게 할당되어, fig1. 회로 그림을 참고하면, 해당 값이 memory 접근 주소로 사용됨을 할 수 있다.

[10] TimerCounter 모듈 (ARM_System/Timer/TimerCounter.v)

이 값을 clock의 negative edge마다 DataOut로 저장하게 되는데, 이 때 해당 모듈에서 StatusR을 다시 0으로 초기화 시킨 값인 0이 DataOut에 전달되어 rd_data에 0이 적힘으로써, Assembly 코드의 bne 명령어가 조건이 성립하여 무한 loop을 돌게 된다[11]. 이 때문에 sec0로 jump하지 못하였고, 보드에 A-00 0000만 표기된 채 타이머가 증가하지 않는 문제가 발생하였다.

iii. 수정 코드 및 해결방법

```
always @(negedge clk, negedge reset_n)
  begin
    if (!reset_n)
            pc <= 0;
    else if (PCWrite | beq_taken | bne_taken | blt_taken | bgeu_taken)
    begin
            pc <= rd_data
    end
  end
```

rd_data가 할당 받은 값이 pc가 할당 받을 값과 동일하기 때문에[12] pc값을 바로 rd_data로 할당하였다. rd_data을 할당하는 로직은 always @(*) 블록에 해당하기 때문에, 변수의 변화마다 실행되어 값을 저장할 수 있는데, pc로직은 clock의 negative edge에서 실행되므로, 값의 변화를 즉시 참조하는 과정에서 딜레이가 발생하여 해당문제가 발생한 것으로 이해했다. 따라서 rd_data을 직접적으로 pc에 할당하였더니, rd_data가 1로 적절히 바뀌었고, 따라서 bne 명령어 조건에 부합하지 않아 다음 명령어인 jump가 수행되었고, 때문에 sec0로 분기하여 타이머가 정상적으로 증가할 수 있었다.

수정 후 Simulation Report는 아래 Fig3.와 같다.

[11] Assembly 코드상 rd_data가 1로 설정되어야 bne명령어의 조건에 부합하지 않게 된다.

[12] rd_data 로직 (ARM_System/RV32I_Core/RV32ICPU.v 373th row)

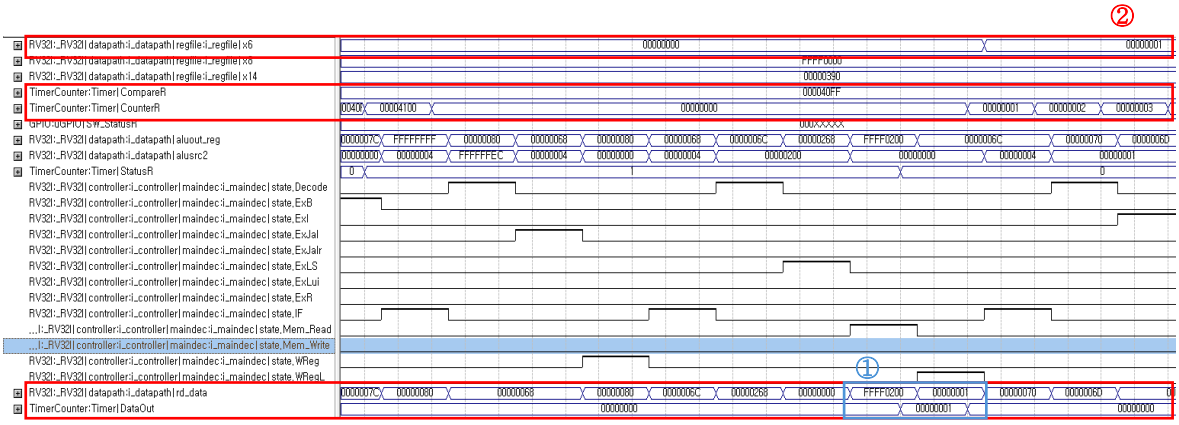

Fig3. Quartus Simulation Report – TimerCounter:Timer|DataOut의 1값 유지 문제 수정 후

DataOut이 WRegL 단계까지 1로 유지가 되어, rd_data가 1로 바뀌었고(① 부분), x6에 1 값이 load 되어 들어오고 있음(② 부분)을 볼 수 있다. 때문에 타이머가 정상적으로 증가하게 되었다.

B. B-Type(ExB state) Control signal 할당 문제

i. 기존 코드

```
ExB: begin
    controls <= 11'b10_00_01_10_001;
    ALUop <= 0;
end
```

ii. 문제점

ExB 단계의 Control signal을 할당하는 과정에서 PCWrite 신호가 1로 할당되어 있다. 따라서, PC로직에서 PCWrite 신호가 1이기에 항상 else if 문[13]을 실행하게 되고, B-type 명령어의 조건에 관계없이 무조건 PC값을 target address로 설정하는 문제가 발생했다. 따라서 bne명령어에서 조건에 부합하지 않아도 무조건 분기를 하게 되는 문제가 발생하였다.

iii. 수정코드 및 해결방법

```
ExB: begin
    controls <= 11'b10_00_00_10_001;
```

[13] PC 로직 (ARM_System/RV32I_Core/RV32ICPU.v 338th row)

```
    ALUop <= 0;
end
```

PCWrite 신호를 0으로 할당함으로써, branch 조건에 부합할 경우 beq/bne/blt/bgeu taken이 활성화 되고, 이 경우에만 PC로직의 else if 문이 실행되며, 조건에 부합하지 않을 때는 분기를 하지 않게 되어 해당 문제를 해결할 수 있었다.