2024 Spring CSED311 Lab 3 Report

Team ID: 67735
20220312 박준혁, 20220871 홍지우

명예서약 (Honor Code)

나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.

I completed this programming task without the improper help of others.

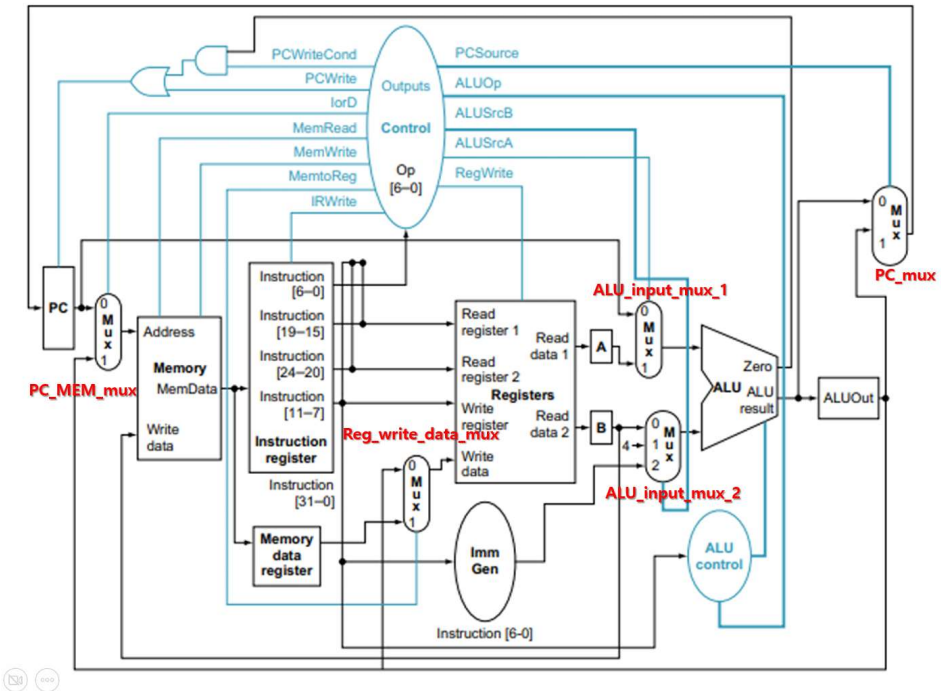
1 Introduction

각 스테이지별로 쪼개어 연산하는 Multi-Cycle CPU를 Finite State Machine 기반으로 구현한다.

각 연산별로 요구 사이클이 다르기 때문에 빠른 연산은 빨리 끝낼 수 있다;
오래 걸리는 연산에 사이클을 맞추던 싱글사이클보다 빠르게 동작한다.

2 Design

2.1 전체 구조



기존 싱글 사이클에서 A,B, inst, mem. aluout 레지스터, mux들이 추가된 형태이다.

2.2 모듈 동기화 여부

Synchronous:

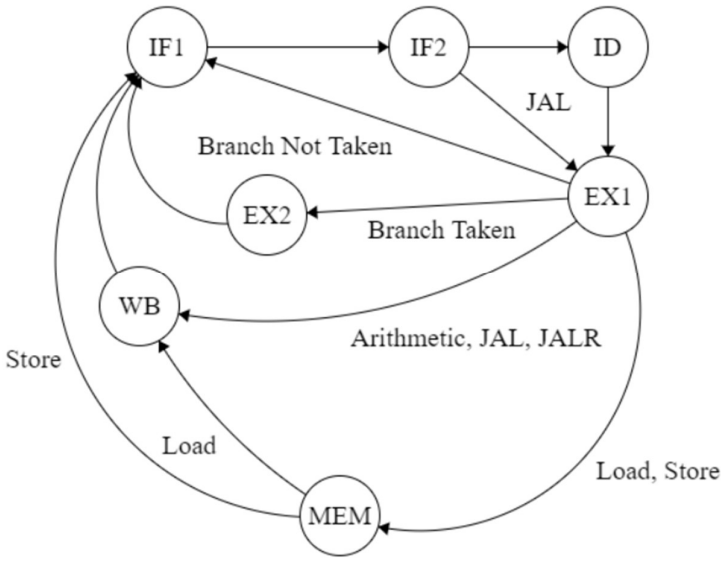
PC, Register File, Memory, IR, ALUOut, MDR, Control Unit

Asynchronous

ALU, ALU Control, ImmGen

Asynchronous read from memory and register

2.3 FSM Design



IF를 두단계, EX를 두단계, MEM을 한단계로 구현하였다.

EX2의 경우 branch taken을 처리하는 용도로 추가하였다.

나머지 타입에 대한 연산은 EX1에서 끝난다

2.4 Microcode Control Diagram

State	Flow	R/I	LD	SD	Bxx	JALR	JAL
IF1	next	-	-	-	-	-	-
IF2	go to	ID	ID	ID	ID	ID	EX1
ID	next						
EX1	go to	WB	MEM	MEM	EX2	WB	WB
EX2	go to				IF1		
MEM	go to		WB	IF1			
WB	go to	IF1	IF1			IF1	IF1
CPI		5	6	5	5	5	4

3 Implementation

3.1 Control Unit

3.1.1 State Transition (mictrocode controller)

```
always @(*) begin
  NextState=`IF_1;
  case (State)
    `IF_1: begin
      NextState=`IF_2;
    end
    `IF_2: begin
      if(op==`JAL)
        NextState=`EX_1;
      else if (op==`ECALL)
        NextState=`IF_1;
      else
        NextState=`ID;
    end
    `ID: begin
      NextState=`EX_1;
    end
    `EX_1: begin
      case (op)
        `ARITHMETIC,`ARITHMETIC_IMM:
          NextState=`WB;
        `LOAD, `STORE:
          NextState=`MEM;
        `BRANCH: begin
          if(isTaken)
            NextState=`EX_2;
          else
            NextState=`IF_1;
        end
        `JALR, `JAL:
          NextState=`WB;
        default:
          NextState=`IF_1;
      endcase
    `EX_2: begin
      NextState=`IF_1;
    end
    `MEM: begin
      if(op==`LOAD)
        NextState=`WB;
      else
        NextState=`IF_1;
    end
    `WB: begin
        NextState=`IF_1;
    end
    default: begin
        NextState=`IF_1;
    end
  endcase
end
```

3.1.2 Module Port Declaration, Synchronization

```
module ControlUnit (
    input reset,
    input clk,
    input [6:0] op,
    input isTaken,
    output reg is_jal,
    output reg is_jalr,
    output reg mem_read,
    output reg mem_to_reg,
    output reg mem_write,
    output reg alu_srcA,
    output reg [1:0] alu_srcB,
    output reg [1:0] alu_control,
    output reg write_enable,
    output reg pc_to_reg,
    output reg pc_source,
    output reg pc_write,
    output reg pc_write_cond,
    output reg i_or_d,
    output reg ir_write,
    output reg is_ecall
);
```

```
// change state synchronously
always @(posedge clk) begin
        if (reset) begin
            State <= `IF_1;
        end
        else begin
            State <= NextState;
        end
end
```

3.1.3 Setting Control Values: Init, IF1, IF2

```
// setting control values
always @(*) begin
  mem_read=0;
  mem_to_reg=0;
  mem_write=0;
  pc_to_reg=0;
  pc_source=0;
  pc_write =0;
  pc_write_cond=1;
  alu_srcA=0;
  alu_srcB=0;
  alu_control = `ALU;
  i_or_d=0;
  ir_write=0;
  write_enable=0;
  is_ecall=0;
```

```
case (State)
  `IF_1: begin
    // IR = MEM[PC]
    mem_read=1;
    ir_write=1;
    i_or_d=1;
    alu_control = `ALU_NOP;
  end
  `IF_2: begin
    mem_read=1;
    ir_write=1;
    alu_control = `ALU_NOP;
    if(op==`JAL) begin
      // ALUOut = PC+4
      alu_srcA=0;   alu_srcB=1;
      alu_control = `ALU_ADD;
    end
    if(op==`ECALL) begin
      // PC=PC+4
      alu_srcA = 0; alu_srcB = 1;
      alu_control = `ALU_ADD;
      pc_source = 0;
      pc_write = 1;
    end
  end
```

`ALU 시그널은 기존과 동일하게 다양한 연산을 하도록 한다.

`ALU_ADD, ALU_SUB는 ALU가 더하기와 빼기만 하도록 한다.

`ALU_NOP는 아무런 결과도 생성하지 않도록 막는 역할을 한다.

```
`ID: begin
  // ALUOut = PC+4
  alu_srcA=0;   alu_srcB=1;
  alu_control = `ALU_ADD;
end
`EX_1: begin
  case (op)
    `ARITHMETIC: begin
      // ALUOut = A+B
      alu_srcA=1;   alu_srcB=0;
      alu_control = `ALU;
    end
    `ARITHMETIC_IMM: begin
      // ALUOut = A+Imm
      alu_srcA=1;   alu_srcB=2;
      alu_control = `ALU;
    end
    `LOAD, `STORE: begin
      //ALUOut = A+Imm
      alu_srcA=1;   alu_srcB=2;
      alu_control = `ALU_ADD;
    end
    `BRANCH: begin
      // calculate cond: A-B
      alu_srcA=1;   alu_srcB=0;
      alu_control = `ALU_SUB;

      pc_source=1;
      pc_write_cond=0;
      if(isTaken==0) begin
        // PC=ALUOut
        pc_write=1;
      end
    end
    `JAL: begin
      // rd = ALUOut
      write_enable = 1;
      mem_to_reg = 0;
    end
    `JALR: begin
      // rd = ALUOut
      write_enable = 1;
      mem_to_reg = 0;
    end
    default: begin
      alu_control = `ALU_NOP;
    end
  endcase
end
```

```
`EX_2: begin
  if(op==`BRANCH) begin
    // PC = PC+Imm
    alu_srcA=0;   alu_srcB=2;
    alu_control = `ALU_ADD;
    pc_source=0;
    pc_write=1;
  end
end
```

ID, EX 부분

Branch의 경우 EX1에서 조건을 계산하고, 만약 not taken이면 값을 쓰고 IF로 돌아간다.

만약 taken이면 EX2로 넘어가 PC=PC+imm을 하고 IF로 돌아간다.

Branch를 제외한 다른 연산은 EX1에서 연산이 끝난다.

```
`MEM: begin
  i_or_d=1;
  if(op==`LOAD) begin
    // MDR = MEM[ALUOut]
    mem_read=1;
  end
  if(op==`STORE) begin
    // MEM[ALUOut] = B
    mem_write=1;

    // PC=PC+4
    alu_srcA = 0; alu_srcB = 1;
    alu_control = `ALU_ADD;
    pc_source = 0;
    pc_write = 1;
  end
end
```

MEM 부분: Load, Store의 경우에만 작동한다.

Store의 경우에는 WB를 거치지 않고 여기서 끝난다.

```
`WB: begin
  case (op)
    `ARITHMETIC,`ARITHMETIC_IMM: begin
      // rd = ALUOut
      write_enable = 1;
      mem_to_reg = 0;

      // PC=PC+4
      alu_srcA = 0; alu_srcB = 1;
      alu_control = `ALU_ADD;
      pc_source = 0;
      pc_write = 1;
    end
    `LOAD: begin
      // rd = MDR
      write_enable = 1;
      mem_to_reg = 1;

      // PC=PC+4
      alu_srcA = 0; alu_srcB = 1;
      alu_control = `ALU_ADD;
      pc_source = 0;
      pc_write = 1;
    end
    `JAL: begin
      // PC = PC+Imm
      alu_srcA=0;   alu_srcB=2;
      alu_control = `ALU_ADD;
      pc_source=0;
      pc_write=1;
    end
    `JALR: begin
      // PC = A+Imm
      alu_srcA=1;   alu_srcB=2;
      alu_control = `ALU_ADD;
      pc_source=0;
      pc_write=1;
    end
    default: begin
      alu_srcA=0;   alu_srcB=0;
    end

    // store, branch does not requrie WB
  endcase
end
```

WB 부분

각 Type에 맞는 PC 값을 업데이트 해 준다.

4 Discussion

4.1 논의

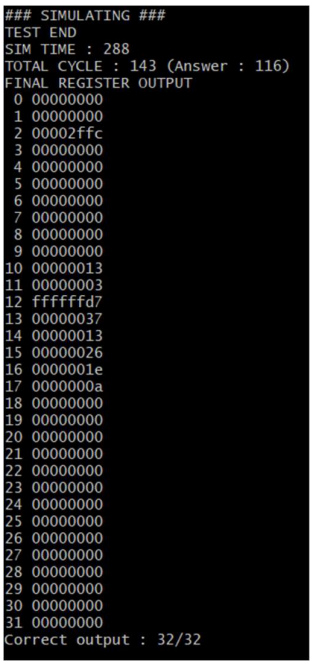
4.1.1 Branch의 계산과 PC를 컨트롤 하는 것이 복잡했다.

4.1.2 state를 여러 개로 쪼갰더니 레지스터에서 값이 사라지거나 하는 문제가 있어, 꼭 나누어야 하는 state를 제외하고는 하나로 합쳤다

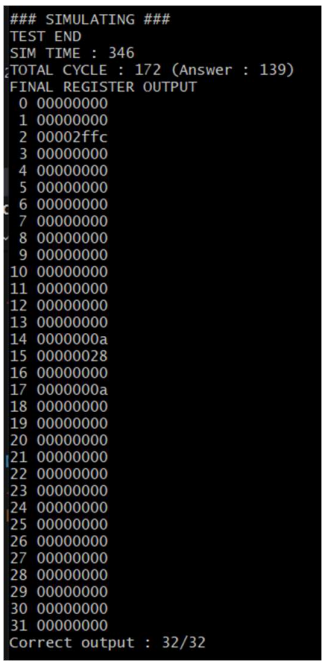
4.1.3 ALU가 원하지 않는 타이밍에 동작하는 것을 막기 위해 ALU_NOP 시그널을 주어 계산하지 않도록 하였다.

5 Conclusion

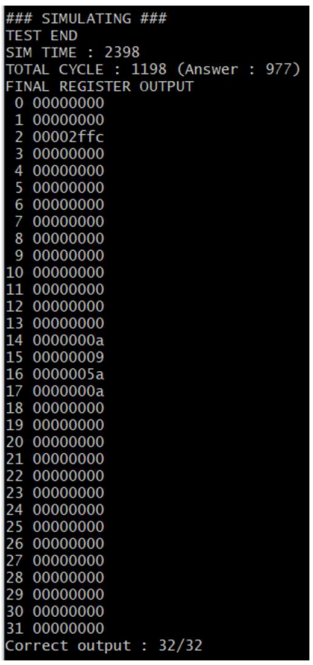
5.1 tb 결과



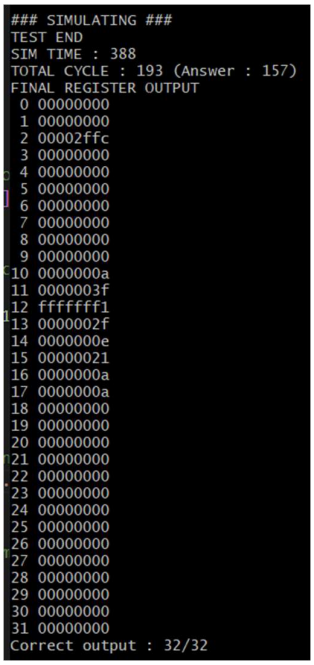
basic_mem

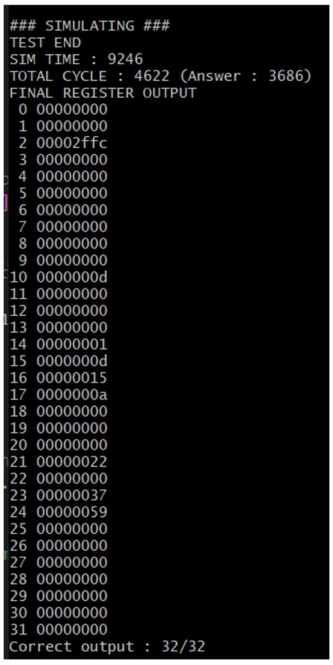


ifelse_mem



loop_mem





non-controlflow_mem recursive_mem

전부 동일한 레지스터 값을 얻을 수 있었다.