CSED311 Report

2023/03/28

LAB2: Single-cycle CPU

과목	컴퓨터 구조
학번	20180340, 20200195
이름	김재진, 이재윤
담당교수	김광선
제출기간	2023-3-28 23:59:00

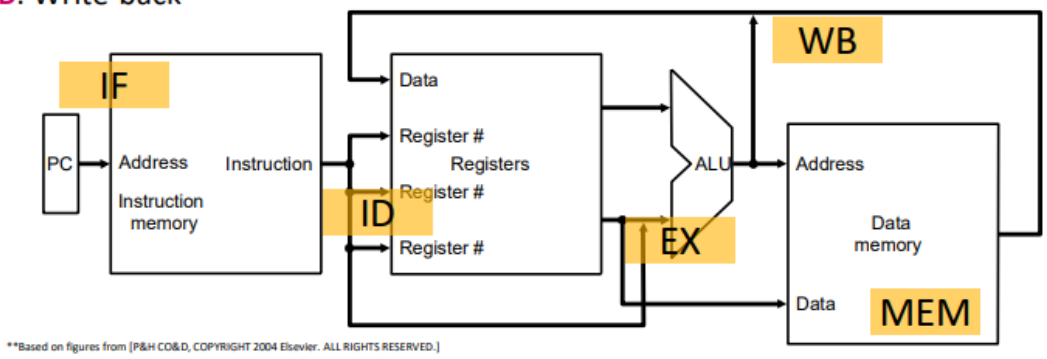
1. Introduction

이번 과제에는 Verilog 이용하여 RISC-V ISA 의 Single-cycle CPU 를 구현해야 한다. Memory.v, RegisterFile.v, cpu.v 의 skeleton code 가 주어지고, Single-cycle CPU 의 Datapath(ALU, RegFIle)와 control unit 을 구현한다. 또한 한 클럭 사이클 당 하나의 명령어(instruction)을 처리하도록 설계한다. 이를 통해 Single-cycle CPU 의 구성요소와 작동원리를 이해하고 RISC-V ISA 에 익숙해지는 것을 기대한다.

2. Design

A. Description of each stage in single-cycle CPU

– **WB**: Write-back



Instruction Processing 의 5 단계는 위 그림에서 보이는 것과 같다. 이는 Single-cycle CPU 의 각각의 stage 에도 해당한다. 후에 다시 언급하겠지만 PVS(PC, Memory, RF)는 clock synchronous 하게 값이 update 되기에 stage 를 PVS 를 기준으로 나눌 수 있다.

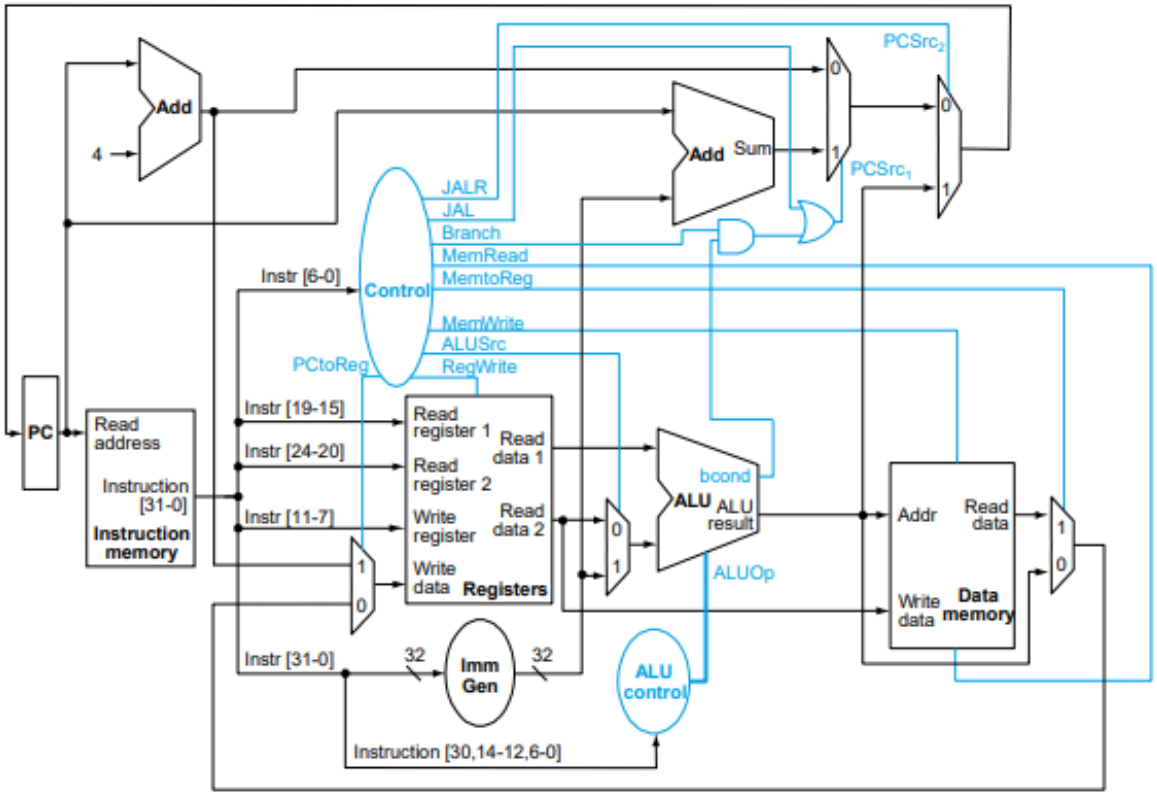
각각의 단계에 대해서 구체적으로 설명을 해보겠다. IF stage 는 Instruction Memory 에서 Instruction 을 Fetch 하는 것이다. ID stage 는 Fetch 한 Instruction 을 decoding 하여 명령을 수행한 operand 를 fetch 하는 단계이다. EX stage 는 ALU 를 사용하여 명령어를 실행하는 단계이다. MEM stage 는 Data

메모리에 접근하여 값을 읽거나 저장하는 단계이다. WB stage 는 Memory 에서 읽은 값 혹은 ALU 의 결과를 레지스터에 저장하고 혹은 경우에 따라서 PC + 4 값을 rd 레지스터에 저장하는 단계이다.

B. Description of whether each module (RF, memory, PC, control unit) is clock synchronous or asynchronous.

위의 항목에서 설명하였듯이 single-cycle CPU 에서 clock synchronous 한 것은 PVS(Programmer Visible state)이다. PVS 의 구성요소로는 PC, Register File (RF), Memory(Instruction memory, Data memory)가 있다. 더 구체적으로 설명하자면, RF 와 메모리에서 read 하는 부분은 clock asynchronous 하고, write 할 때에는 clock synchronous 하다. 반대로 이를 제외한 다른 모듈들(ex. ALU 와 Adder, Control Unit, Immediate Generator, Mux)은 clock asynchronous 하다.

C. Single-cycle CPU design



항목 A 에서 설명한 PVS 를 제외하고 ALU, ALU control, Adder, Immediate Generator, Control Unit, Mux 모듈이 필요하다. 또한 Control Unit 과 ALU Control 에서 Control 신호를 보내줘서 Mux 와 ALU 의 동작을 결정한다.

각각의 Control 신호에 대해서 간략하게 설명하자면(위의 그림을 기준으로 설명) JALR 신호는 JALR 명령어 일 때 켜지는 신호이다. JAL 은 마찬가지로 JAL 명령어일 때 켜지는 신호이고 Branch 신호 역시 Branch 명령어일 때 켜지는 신호다. MemRead 는 메모리의 값을 읽는 Load 명령어일 때 켜지는 신호이고, MemWrite 는 반대로, 메모리에 값을 쓰는 Store 명령어 일 때 켜지는 신호이다. Regwrite 신호는 명령어가 Store 와 Brach 명령어가 아닐 때 즉, 레지스터에 값을 쓰는 명령어일 때 켜지는 신호이다. MemtoReg 는 메모리에서 읽은 값을 레지스터에 쓰는 Load 명령어일 때 켜진다. ALUSrc 는 2nd ALU input 을 rs2 레지스터에 들어있는 값으로 할 지 아니면 Immediate Generator 에서 생성한 immediate value 로 할 지 결정하는 mux 에 들어가는 control 신호이다. PCtoReg 는 JAL 혹은 JALR 에서 PC + 4 를 레지스터에 쓸 때 켜지며, ALU 의 결과나 메모리에서 읽은 값을 레지스터에 써야 할 때 꺼진다. PCSrc1 과 PCSrc2 는 nextPC(target) 값을 결정하는 mux 에 들어가는 control 신호이다.

이러한 단일 비트 신호 외에도 Multi-bit control 신호들이 존재한다. ALUOp 는 ALU 가 어떠한 연산을 수행할 지 결정해주는 control 신호이며, Immediate Generator 를 통해 생성된 immediate value 역시 instruction format type 에 따라 다르게 선택되는 일종의 Multi-bit control 신호이다.

각각의 Instruction type 마다 통과하는 instruction processing stage 가 다르며, 적절한 control 신호를 통해 data path 를 제어한다.

3. Single-cycle CPU implementation

코드를 보기 편하게 VS code 편집기로 소스파일을 열어 캡처를 하였다.

```
PC pc(···
);

Adder pc_adder1(···
);

// ---------- Instruction Memory ----------
InstMemory imem(···
);

Mux pc_reg_mux(···
);

// ---------- Register File ----------
RegisterFile reg_file (···
);


// ---------- Control Unit ----------
ControlUnit ctrl_unit ( ···
);

// ---------- Immediate Generator ----------
ImmediateGenerator imm_gen(···
);

Mux alu_mux(···
);
```

```
// ---------- ALU Control Unit ----------
ALUControlUnit alu_ctrl_unit (···
);

// ---------- ALU ----------
ALU alu (···
);

Adder pc_adder2(···
);

assign PCsrc1 = (branch && alu_bcond) || is_jal;

Mux pc_mux1(···
);

assign PCsrc2 = is_jalr;

Mux pc_mux2(···
);


// ---------- Data Memory ----------
DataMemory dmem(···
);

Mux Dmem_mux(···
);
```

먼저 cpu.v 에 위 사진과 같이 모듈 (왼쪽 -> 오른쪽)을 추가하여 전체적인 Single-cycle CPU 의 구조를 완성하였다.

```
// Register file
reg [31:0] rf[0:31];

// TODO
// Asynchronously read register file
// Synchronously write data to the register file

assign rs1_dout = rf[rs1];
assign rs2_dout = rf[rs2];
assign x17 = rf[17];

always @ (posedge clk) begin
  if (write_enable && rd != 0) begin
    rf[rd] <= rd_din;
  end
end
```

옆의 사진은 RegisterFile.v 의 코드의 주요 부분이다. clock 에 비동기적으로 RF 에 레지스터 ID 로 접근하여 값을 읽고, clock 의 positive edge 에 맞춰 rd 에 write data 를 할당하는데 여기서 조건이 붙는다. 먼저 write_enable(Regwrite) control 신호가 들어와야 하며, rd 가 x0 일

경우, x0 은 항상 0 값이 들어있어야 하므로 이 경우에는 레지스터에 값을 쓰는 것이 불가능하다. 따라서 if 문으로 이러한 경우를 고려한다.

```
integer i;
// Data memory
reg [31:0] mem[0: MEM_DEPTH - 1];
// Do not touch dmem_addr
wire [31:0] dmem_addr;
assign dmem_addr = {2'b00, addr >> 2};

// TODO
// Asynchrnously read data from the memory
// Synchronously write data to the memory
// (use dmem_addr to access memory)

assign dout = (mem_read) ? mem[dmem_addr] : dout;

always @ (posedge clk) begin
  if (mem_write) begin
    mem[dmem_addr] <= din;
  end
end
```

다음으로 data memory 를 살펴보면, 먼저 input 으로 들어온 addr 을 그대로 사용하는 것이 아니라 unsigned 상태로 4 로 나눠서 data memory 에 접근할 수 있는 주소 값으로 만든다. 또한 RF 와 같이 read 는 비동기적으로 write 는 동기적으로 작성한다. Read 의 경우는 mem_read 신호가 켜졌을 때 값을 읽을 수 있게 하였고 Write 는 clock 의 positive edge 이고 mem_write 신호가 켜졌을 때 값을 쓸 수 있게 하였다. Instruction memory 는 위 구조와 유사하므로 넘어가겠다.

```
always @(*) begin
    case(opcode)
        `ARITHMETIC_IMM: begin
            if(funct3 == `FUNCT3_SLL || funct3 == `FUNCT3_SRL) begin // shift
                imm_gen_out = {{27{1'b0}}, part_of_inst[24:20]};
            end
            else begin
                imm_gen_out = {{20{part_of_inst[31]}}, part_of_inst[31:20]};
            end
        end

        `LOAD: begin
            imm_gen_out = {{20{part_of_inst[31]}}, part_of_inst[31:20]};
        end

        `STORE: begin
            imm_gen_out = {{20{part_of_inst[31]}}, part_of_inst[31:25], part_of_inst[11:7]};
        end

        `BRANCH: begin
            imm_gen_out = {{20{part_of_inst[31]}}, part_of_inst[7], part_of_inst[30:25], part_of_inst[11:8], 1'b0};
        end

        `JAL: begin
            imm_gen_out = {{12{part_of_inst[31]}}, part_of_inst[19:12], part_of_inst[20], part_of_inst[30:21], 1'b0};
        end

        `JALR: begin
            imm_gen_out = {{20{part_of_inst[31]}}, part_of_inst[31:20]};
        end

        default: imm_gen_out = 0;
    endcase
end
```

위 코드는 ImmediateGenerator.v 에서 Immediate value 를 생성하는 로직이다. Opcode 와 funct3, 7 값에 따라서 Immediate value 를 다르게 생성하였다. Case 문과 결합 연산자와 중복 연산자를 사용하여 비교적 간단하게 구현할 수 있었다.

```
assign is_jal = (part_of_inst == `JAL) ? 1 : 0;
assign is_jalr = (part_of_inst == `JALR) ? 1 : 0;
assign branch = (part_of_inst == `BRANCH) ? 1 : 0;
assign mem_read = (part_of_inst == `LOAD) ? 1 : 0;
assign mem_to_reg = (part_of_inst == `LOAD) ? 1 : 0;
assign mem_write = (part_of_inst == `STORE) ? 1 : 0;
assign alu_src = (part_of_inst != `ARITHMETIC && part_of_inst != `BRANCH) ? 1 : 0;
assign write_enable = (part_of_inst != `STORE && part_of_inst != `BRANCH) ? 1 : 0;
assign pc_to_reg = (part_of_inst == `JAL || part_of_inst == `JALR) ? 1 : 0;

assign is_ecall = (part_of_inst == `ECALL) ? 1 : 0;
```

위 코드는 ControlUnit.v 의 일부분으로 control signal 을 생성하는 로직이다. part_of_inst 는 여기서 opcode 를 의미한다. 직관적으로 볼 수 있듯 이 opcode 를 조합하여 Design 항목에서 서술한 control signal 을 생성하여 output 으로 내보낸다.

```
always @(*) begin

    case(part_of_inst[6:0]) // according to opcode

        `ARITHMETIC: begin

            case(part_of_inst[14:12]) // according to funct3
                `FUNCT3_ADD: begin // `FUNCT3_SUB
                    if (part_of_inst[31:25] == `FUNCT7_SUB)
                        alu_op = `OP_SUB;
                    else
                        alu_op = `OP_ADD;
                end

                `FUNCT3_SLL: begin
                    alu_op = `OP_SLL;
                end

                `FUNCT3_XOR: begin
                    alu_op = `OP_XOR;
                end

                `FUNCT3_OR: begin
                    alu_op = `OP_OR;
                end

                `FUNCT3_AND: begin
                    alu_op = `OP_AND;
                end

                `FUNCT3_SRL: begin
                    alu_op = `OP_SRL;
                end
            endcase
        end
```

옆의 코드는 ALUControlUnit.v 의 일부분이다. 이중 case 문과 if else 를 활용하여 opcode 와 funct3, func7 에 따라서 다른 alu_op 를 생성하고 이를 ALU 에 output 으로 보내준다. ADD 와 SUB 연산이 조금 독특한데 먼저, R-type 에서는 ADD 와 SUB 연산이 구분되고 I-type 에서는 ADD 와 SUB 이 구분이 되지 않는다. 그 이유는 I-type 에서는 immediate value 의 부호를 음으로 바꿔서 ADD 연산 하나로 SUB 까지 구현할 수 있기 때문이다. 옆의 사진은 R-type 의 소스코드로 funct7 값에 따라서 ADD 연산과 SUB 연산을 구분한다.

이외에 나머지 모듈들은 코드가 상대적으로 간단하기에 보고서에는 기술하지 않도록 하겠다.

4. Discussion

```
`OP_BLT: begin
    alu_result = alu_in_1 - alu_in_2;

    if ($signed(alu_result) < 0)
        alu_bcond = 1;
    else
        alu_bcond = 0;
end

`OP_BGE: begin
    alu_result = alu_in_1 - alu_in_2;

    if ($signed(alu_result) >= 0)
        alu_bcond = 1;
    else
        alu_bcond = 0;
end
```

이 코드는 ALU.v 의 코드이다. 처음에 $signed 함수 없이 alu_result 와 0 을 비교했더니 BLT instruction 이 정상적으로 작동되지 않았다. 그래서 alu_result 에 $signed 함수를 사용하였더니 정상적으로 작동하였다. Verilog 에서는 기본적으로 신호가 unsigned 타입으로 간주되어 0 보다 작은 것을 비교할 때는 signed type 으로 casting 을 해줘야 한다. BGE 명령어 역시, 음수가 들어왔을 때 이를 unsigned 로 인식하여 잘못되게 작동하는 것을 막고자 $signed 함수로 casting 을 하였다.

```
always @ (posedge clk) begin
  if (write_enable && rd != 0) begin
    rf[rd] <= rd_din;
  end
end
```

이 코드는 RegisterFile.v 에 있는 코드이다. 레지스터에 값을 쓰는 것에 관한 코드인데 if 의 조건문에 rd != 0 이라는 조건이 빠졌을 때 정상적으로 작동하지 않았다. 그 이유는 testbench 로 주어진 어셈블리 언어에서 JAL X0, Imm 형식의 명령어가 있기 때문이다. x0 는 hardwired zero 이므로 항상 0 이 들어있는 레지스터이다. 이는 return address 를 기록하지 않고 Jump 하라는 명령어이다. 따라서 write register 가 x0 일 때는 값을 쓰면 안 되므로 rd ! =0 조건을 걸어줘야 한다.

5. conclusion

이번 과제를 통해 RISC-V ISA 의 Single-cycle CPU 를 Verilog 로 구현해보면 CPU 의 datapath, control signal 그리고 어떻게 clock 에 동기적/비동기적으로 작동하는 지를 알 수 있었다.