# CSED311: Lab2 RTL Design

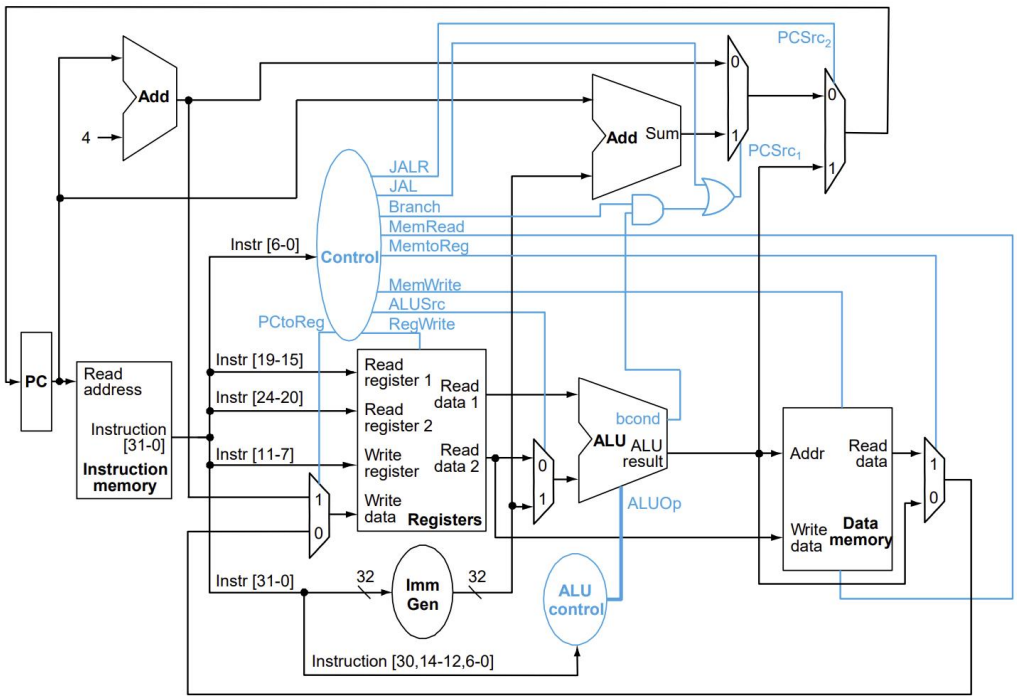
컴퓨터공학과 20220302 김지현

컴퓨터공학과 20220455 윤수인

## Introduction

- 본 과제는 Verilog 를 사용하여 Single-cycle RISC-V CPU (RV32I)를 구현하는 것을 목적으로 하며, 더욱 자세하게는 Datapath 에 해당하는 ALU 와 Register file 관련 모듈, Control unit 과 immediate generator 등의 모듈을 구현하여 3 개의 test(basic_mem, loop_mem, non-controlflow)에 통과하는 CPU 를 제작하는 것을 목적으로 한다.
- Verilator 환경에서 실행하며, Ripes simulator 를 사용하여 각 instruction 에 따른 레지스터 값의 변화를 관찰한다.
- CPU 는 한 사이클(cycle) 당 하나의 instruction 을 수행하여야 한다.

## Design



<그림 1 : CPU diagram - Datapath with control>

구현하여야 할 Single cycle cpu 의 구체적인 동작은 다음과 같다.

먼저 memory 에서 instruction 을 불러와 저장하고, PC 값을 4 만큼 증가시킨다. 다음으로 R-type, I-type 등 연산의 종류에 따라 register file 에서 최대 2 개의 register 값을 읽어와 할당한다. Register 값을 할당하면 미리 읽어 온 opcode 에 따라 alu 를 통해 각기 다른 연산을 수행한다. 만약 추가적인 메모리 접근이 필요한 lw(load), sw(store) 등의 instruction 인 경우, data memory 에 접근하여 데이터를 memory 에서 읽거나 쓰도록 한다. Instruction 의 종류에 따라 다음 PC 값을 설정한다. Branch instruction 의 경우 alu 에서 연산한 target 으로 PC 를 설정하고, 그 외 일반적인 경우는 PC + 4 로 설정한다. 각기 다른 instruction 에 따라 동작을 다르게 수행하는 과정은 Control Unit 에서 총괄한다.

## Implementation

- **pc.v**

```
module pc (
    input reset,
    input clk,
    input [31:0] next_pc,
    output reg [31:0] current_pc);

    always @(posedge clk) begin
        if (reset) begin
            current_pc <= 32'b0;
        end
        else begin
            current_pc <= next_pc;
        end
    end

endmodule
```

<그림 2 : pc.v 모듈의 구성>

pc.v 모듈에서는 reset 이 1 인 경우 current_pc 의 값을 0 으로 업데이트하고, reset 이 1 이 아닌 경우에는 next_pc 의 값으로 업데이트한다. 이 과정은 clock synchronous 하게 이루어진다.

- **adder.v**

```
module adder (
    input[31:0] in1,
    input [31:0] in2,
    output reg [31:0] out);

    assign out = in1 + in2;

endmodule
```

<그림 3: adder.v 모듈의 구성>

adder.v 모듈은 PC 값을 업데이트하기 위해서 필요한 모듈이다. pc_4 의 경우 pc = pc + 4 의 형태로, pc_imm 은 pc = pc + immediate 의 형태로 adder 를 사용한다. 이 과정은 clock asynchronous 하게 이루어진다.

- **instruction_memory.v**

```
module instruction_memory #(parameter MEM_DEPTH = 1024) (input reset,
                                                         input clk,
                                                         input [31:0] addr,   // ad
                                                         output [31:0] dout); // in
  integer i;
  // Instruction memory
  reg [31:0] mem[0:MEM_DEPTH - 1];
  // Do not touch imem_addr
  wire [9:0] imem_addr;
  assign imem_addr = addr[11:2];
  // Do not touch or use this wire
  wire _unused_ok = &{1'b0,
                      addr[31:12],
                      addr[1:0],
                      1'b0};

  // TODO
  // Asynchronously read instruction from the memory
  // (use imem_addr to access memory)
  assign dout = mem[imem_addr];

  // Initialize instruction memory (do not touch except path)
  always @(posedge clk) begin
    if (reset) begin
      for (i = 0; i < MEM_DEPTH; i = i + 1)
        // DO NOT TOUCH COMMENT BELOW
        /* verilator lint_off BLKSEQ */
        mem[i] = 32'b0;
        /* verilator lint_on BLKSEQ */
        // DO NOT TOUCH COMMENT ABOVE

      // Provide path of the file including instructions with binary format
      $readmemh("C:/Users/tndls/OneDrive/Desktop/윤수인/POSTECH/3-1/컴퓨터구조/Lab2/
    end
  end

endmodule
```

<그림 4: instruction_memory.v 모듈의 구성>

instruction_memory.v 모듈은 reset 이 1 인 경우 clock synchronous 하게 instruction memory 를 초기화한다. Reset 이 1 이 아닌 경우, asynchronous 하게 instruction 주소를 받아와 메모리에 저장된 instruction 을 dout 에 저장한다.

- **register_file.v**

```
module register_file(input  reset,
                     input clk,
                     input [4:0] rs1,          // source register 1
                     input [4:0] rs2,          // source register 2
                     input [4:0] rd,           // destination register
                     input [31:0] rd_din,      // input data for rd
                     input write_enable,          // RegWrite signal
                     output reg [31:0] rs1_dout,   // output of rs 1
                     output reg [31:0] rs2_dout,   // output of rs 2
                     output [31:0] rf17,
                     output [31:0] print_reg [0:31]);
  integer i;
  // Register file
  reg [31:0] rf[0:31];
  // Do not touch or use print_reg
  assign print_reg = rf;

  // TODO
  assign rf17 = rf[17];
  // Asynchronously read register file
  // Synchronously write data to the register file
  assign rs1_dout = rf[rs1];
  assign rs2_dout = rf[rs2];


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

  // Initialize register file (do not touch)
  always @(posedge clk) begin
    // Reset register file
    if (reset) begin
      for (i = 0; i < 32; i = i + 1)
        // DO NOT TOUCH COMMENT BELOW
        /* verilator lint_off BLKSEQ */
        rf[i] = 32'b0;
        /* verilator lint_on BLKSEQ */
        // DO NOT TOUCH COMMENT ABOVE

      // DO NOT TOUCH COMMENT BELOW
      /* verilator lint_off BLKSEQ */
      rf[2] = 32'h2ffc; // stack pointer
      /* verilator lint_on BLKSEQ */
      // DO NOT TOUCH COMMENT ABOVE
    end
  end
endmodule
```

<그림 5: register_file.v 모듈의 구성>

register_file.v 모듈은 register 의 위치를 받아와 해당 register 의 값을 읽거나 쓰는 역할을 한다. Register 값을 읽는 과정은 clock asynchronous 하고, 5-bit input 인 rs1 과 rs2 각각에 해당하는 register 의 값을 32-bit output 인 rs1_dout, rs2_dout 에 각각 저장한다.

Register 값을 쓰는 과정은 clock synchronous 하고, write_enable 시그널이 1 이고 destination register(rd)가 존재할 때 32-bit input data 인 rd_din 을 rd 위치에 해당하는 register 에 저장한다.

- **control_unit.v**

```
`include "opcodes.v"

module control_unit (
  input [6:0] part_of_inst,
  output reg is_jal,
  output reg is_jalr,
  output reg branch,
  output reg mem_read,
  output reg mem_to_reg,
  output reg mem_write,
  output reg alu_src,
  output reg write_enable,
  output reg pc_to_reg,
  output reg is_ecall);

  always @(*) begin
    is_jal = 0;
    is_jalr = 0;
    branch = 0;
    mem_read = 0;
    mem_to_reg = 0;
    mem_write = 0;
    alu_src = 0;
    write_enable = 0;
    pc_to_reg = 0;
    is_ecall = 0;

    case(part_of_inst[6:0])
      `JAL: begin
        is_jal = 1;
        write_enable = 1;
        pc_to_reg = 1;
      end

      `JALR: begin
        is_jalr = 1;
        write_enable = 1;
        pc_to_reg = 1;
        alu_src = 1;
      end

      `BRANCH: begin
        branch = 1;
      end

      `LOAD: begin
        mem_read = 1;
        write_enable = 1;
        mem_to_reg = 1;
        alu_src = 1;
      end

      `STORE: begin
        mem_write = 1;
        alu_src = 1;
      end

      `ARITHMETIC_IMM: begin
        write_enable = 1;
        alu_src = 1;
      end

      `ARITHMETIC: begin
        write_enable = 1;
      end

      `ECALL: begin
        is_ecall = 1;
      end

      default: begin

      end
    endcase
  end
endmodule
```

<그림 6: control_unit.v 모듈의 구성>

control_unit.v 모듈은 part_of_inst(opcode)에 따라 시그널의 값을 활성화/비활성화하여 CPU 의 동작을 결정한다. 이 과정은 clock asynchronous 하게 이루어진다.

- **immediate_generator.v**

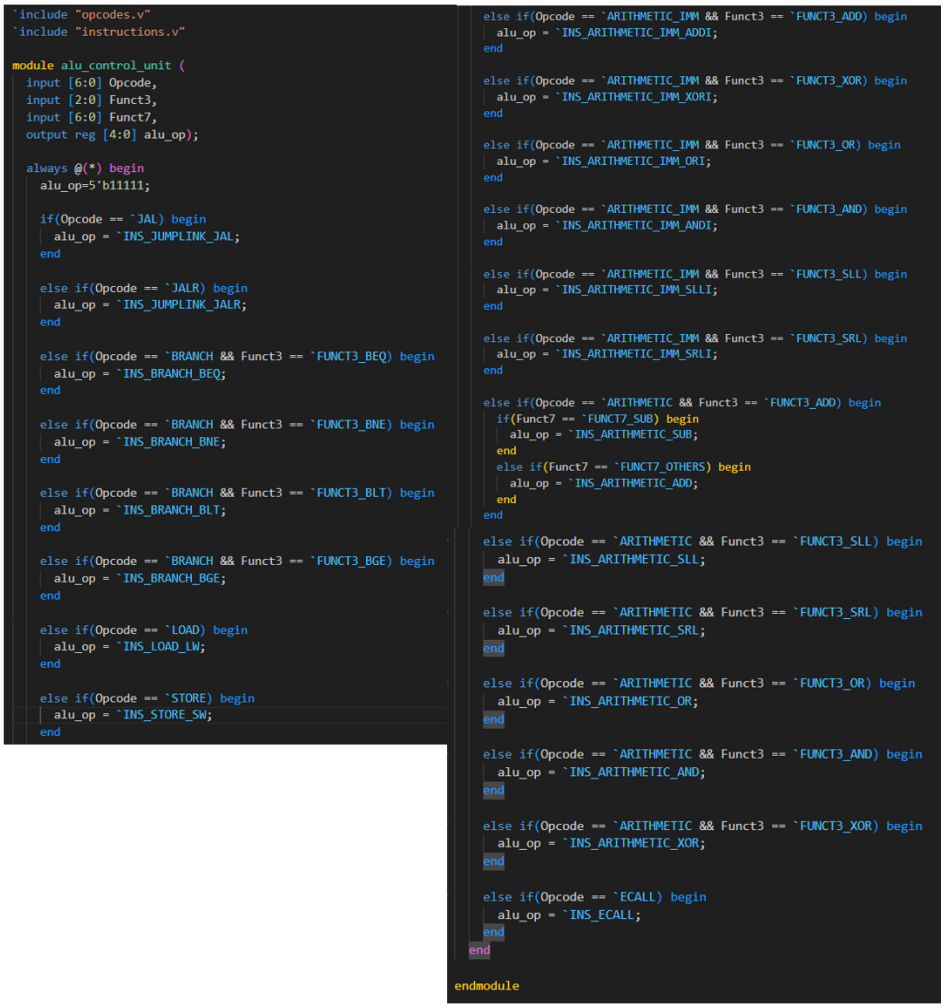
```
`include "opcodes.v"

module immediate_generator (input  reg [31:0]part_of_inst,
                            output reg [31:0] imm_gen_out);

    reg [6:0] opcode;

  always @(*) begin
    imm_gen_out = 32'b0;
    opcode = part_of_inst[6:0];
    case(opcode)
        /*`ARITHMETIC: begin
            imm_gen_out = 32'b0;
        end
        */
        `ARITHMETIC_IMM: begin
            if(part_of_inst[31] == 1) begin  // signed
                imm_gen_out = {20'hfffff, part_of_inst[31:20]};
            end
            else begin
                imm_gen_out = {20'h0, part_of_inst[31:20]};
            end
        end

        `LOAD: begin
            if(part_of_inst[31] == 1) begin  // signed
                imm_gen_out = {20'hfffff, part_of_inst[31:20]};
            end
            else begin
                imm_gen_out = {20'h0, part_of_inst[31:20]};
            end
        end
```

```
        `JALR: begin
            if(part_of_inst[31] == 1) begin  // signed
                imm_gen_out = {20'hfffff, part_of_inst[31:20]};
            end
            else begin
                imm_gen_out = {20'h0, part_of_inst[31:20]};
            end
        end
        `STORE: begin
            if(part_of_inst[31] == 1) begin  // signed
                imm_gen_out = {20'hfffff, part_of_inst[31:25], part_of_inst[11:8], part_of_inst[7]};
            end
            else begin
                imm_gen_out = {20'h0, part_of_inst[31:25], part_of_inst[11:8], part_of_inst[7]};
            end
        end
        `BRANCH: begin
            if(part_of_inst[31] == 1) begin  // signed
                imm_gen_out = {19'h7ffff, part_of_inst[31], part_of_inst[7], part_of_inst[30:25], part_of_inst[11:8], 1'b0};
            end
            else begin
                imm_gen_out = {19'h0, part_of_inst[31], part_of_inst[7], part_of_inst[30:25], part_of_inst[11:8], 1'b0};
            end
        end
        `JAL: begin
            if(part_of_inst[31] == 1) begin  // signed
                imm_gen_out = {11'h7ff, part_of_inst[31], part_of_inst[19:12], part_of_inst[20], part_of_inst[30:21],1'b0};
            end
            else begin
                imm_gen_out = {11'h0, part_of_inst[31], part_of_inst[19:12], part_of_inst[20], part_of_inst[30:21],1'b0};
            end
        end
        default: begin
            imm_gen_out = 32'b0;
        end
    endcase
  end

endmodule
```

<그림 6, 7: immediate_generator.v 모듈의 구성>

immediate_generator.v 모듈은 32-bit 의 part_of_inst 를 input 으로 받아와 하위 7 bits 를 opcode 에 저장한다. Opcode 에 따라 적절한 immediate value 를 imm_gen_out 에 저장한다. Signed 연산 여부는 part_of_inst 의 상위 1 bit 로 판별한다. 전체 과정은 clock asynchronous 하게 이루어진다.

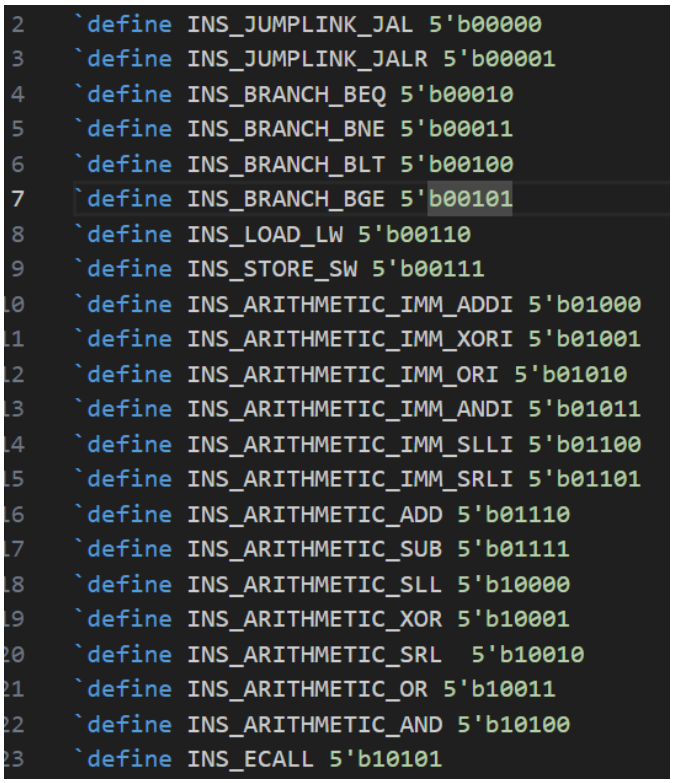
- **alu_control_unit.v**



```
`include "opcodes.v"
`include "instructions.v"

module alu_control_unit (
  input [6:0] Opcode,
  input [2:0] Funct3,
  input [6:0] Funct7,
  output reg [4:0] alu_op);

  always @(*) begin
    alu_op=5'b11111;

    if(Opcode == `JAL) begin
      alu_op = `INS_JUMPLINK_JAL;
    end

    else if(Opcode == `JALR) begin
      alu_op = `INS_JUMPLINK_JALR;
    end

    else if(Opcode == `BRANCH && Funct3 == `FUNCT3_BEQ) begin
      alu_op = `INS_BRANCH_BEQ;
    end

    else if(Opcode == `BRANCH && Funct3 == `FUNCT3_BNE) begin
      alu_op = `INS_BRANCH_BNE;
    end

    else if(Opcode == `BRANCH && Funct3 == `FUNCT3_BLT) begin
      alu_op = `INS_BRANCH_BLT;
    end

    else if(Opcode == `BRANCH && Funct3 == `FUNCT3_BGE) begin
      alu_op = `INS_BRANCH_BGE;
    end

    else if(Opcode == `LOAD) begin
      alu_op = `INS_LOAD_LW;
    end

    else if(Opcode == `STORE) begin
      alu_op = `INS_STORE_SW;
    end

    else if(Opcode == `ARITHMETIC_IMM && Funct3 == `FUNCT3_ADD) begin
      alu_op = `INS_ARITHMETIC_IMM_ADDI;
    end

    else if(Opcode == `ARITHMETIC_IMM && Funct3 == `FUNCT3_XOR) begin
      alu_op = `INS_ARITHMETIC_IMM_XORI;
    end

    else if(Opcode == `ARITHMETIC_IMM && Funct3 == `FUNCT3_OR) begin
      alu_op = `INS_ARITHMETIC_IMM_ORI;
    end

    else if(Opcode == `ARITHMETIC_IMM && Funct3 == `FUNCT3_AND) begin
      alu_op = `INS_ARITHMETIC_IMM_ANDI;
    end

    else if(Opcode == `ARITHMETIC_IMM && Funct3 == `FUNCT3_SLL) begin
      alu_op = `INS_ARITHMETIC_IMM_SLLI;
    end

    else if(Opcode == `ARITHMETIC_IMM && Funct3 == `FUNCT3_SRL) begin
      alu_op = `INS_ARITHMETIC_IMM_SRLI;
    end

    else if(Opcode == `ARITHMETIC && Funct3 == `FUNCT3_ADD) begin
      if(Funct7 == `FUNCT7_SUB) begin
        alu_op = `INS_ARITHMETIC_SUB;
      end
      else if(Funct7 == `FUNCT7_OTHERS) begin
        alu_op = `INS_ARITHMETIC_ADD;
      end
    end

    else if(Opcode == `ARITHMETIC && Funct3 == `FUNCT3_SLL) begin
      alu_op = `INS_ARITHMETIC_SLL;
    end

    else if(Opcode == `ARITHMETIC && Funct3 == `FUNCT3_SRL) begin
      alu_op = `INS_ARITHMETIC_SRL;
    end

    else if(Opcode == `ARITHMETIC && Funct3 == `FUNCT3_OR) begin
      alu_op = `INS_ARITHMETIC_OR;
    end

    else if(Opcode == `ARITHMETIC && Funct3 == `FUNCT3_AND) begin
      alu_op = `INS_ARITHMETIC_AND;
    end

    else if(Opcode == `ARITHMETIC && Funct3 == `FUNCT3_XOR) begin
      alu_op = `INS_ARITHMETIC_XOR;
    end

    else if(Opcode == `ECALL) begin
      alu_op = `INS_ECALL;
    end
  end

endmodule
```

<그림 8: alu_control_unit.v 모듈의 구성>

alu_control_unit.v 모듈은 opcode, funct3, funct7 에 따라 alu 가 수행해야 할 연산을 결정하는 모듈로, 경우에 따라 그림 9 에서 지정된 instruction 번호를 alu_op 에 할당한다. 이 과정은 clock asynchronous 하게 이루어진다.




<그림 9: instructions.v 파일의 구성>

- **alu.v**

```
`include "instructions.v"

module alu (
    input [4:0]  alu_op,      // input
    input [31:0] alu_in_1,    // input
  input [31:0] alu_in_2,    // input
  output reg [31:0] alu_result,  // output
  output reg alu_bcond);     // output

    always @(*) begin
    alu_result = 0;
    alu_bcond = 0;

    case(alu_op)
      `INS_JUMPLINK_JALR, `INS_LOAD_LW, `INS_STORE_SW, `INS_ARITHMETIC_IMM_ADDI, `INS_ARITHMETIC_ADD: begin
            alu_result = alu_in_1 + alu_in_2;
        end

        `INS_BRANCH_BEQ: begin
            alu_bcond = (alu_in_1 == alu_in_2) ? 1 : 0;
        end

        `INS_BRANCH_BNE: begin
            alu_bcond = (alu_in_1 != alu_in_2) ? 1 : 0;
        end

        `INS_BRANCH_BLT: begin
            alu_bcond = (alu_in_1 < alu_in_2) ? 1 : 0;
        end

        `INS_BRANCH_BGE: begin
            alu_bcond = (alu_in_1 >= alu_in_2) ? 1 : 0;
        end

        `INS_ARITHMETIC_IMM_XORI, `INS_ARITHMETIC_XOR: begin
            alu_result = alu_in_1 ^ alu_in_2;
        end

        `INS_ARITHMETIC_IMM_ORI, `INS_ARITHMETIC_OR: begin
            alu_result = alu_in_1 | alu_in_2;
        end

        `INS_ARITHMETIC_IMM_ANDI, `INS_ARITHMETIC_AND: begin
            alu_result = alu_in_1 & alu_in_2;
        end
```

```
        `INS_ARITHMETIC_IMM_SLLI, `INS_ARITHMETIC_SLL: begin
            alu_result = alu_in_1 << alu_in_2;
        end

        `INS_ARITHMETIC_IMM_SRLI, `INS_ARITHMETIC_SRL: begin
            alu_result = alu_in_1 >> alu_in_2;
        end

        `INS_ARITHMETIC_SUB: begin
            alu_result = alu_in_1 - alu_in_2;
        end


        default: begin

        end
    endcase
  end
endmodule
```

<그림 10, 11: alu.v 모듈의 구성>

alu.v 모듈은 앞서 alu_control_unit.v 모듈에서 지정한 alu_op 에 따라 알맞은 연산을 수행하여 연산 결과를 alu_result 와 alu_bcond 에 저장한다. 이 과정은 clock asynchronous 하게 이루어진다.

- **data_memory.v**

```
module data_memory #(parameter MEM_DEPTH = 16384) (input reset,
                                                   input clk,
                                                   input [31:0] addr,    // addr
                                                   input [31:0] din,     // data
                                                   input mem_read,       // is r
                                                   input mem_write,      // is w
                                                   output reg [31:0] dout);  //
  integer i;
  // Data memory
  reg [31:0] mem[0: MEM_DEPTH - 1];
  // Do not touch dmem_addr
  wire [13:0] dmem_addr;
  assign dmem_addr = addr[15:2];
  // Do not touch or use _unused_ok
  wire _unused_ok = &{1'b0,
                      addr[31:16],
                      addr[1:0],
                      1'b0};

  // TODO
  // Asynchrnously read data from the memory
  always @(*) begin
    dout = 0;
    if (mem_read) begin
      dout = mem[dmem_addr]; // Read operation
    end
  end
  // Synchronously write data to the memory
  // (use dmem_addr to access memory)
  always @(posedge clk) begin
    if (mem_write) begin
      mem[dmem_addr] <= din; // Write operation
    end
  end

  // Initialize data memory (do not touch)
  always @(posedge clk) begin
    if (reset) begin
      for (i = 0; i < MEM_DEPTH; i = i + 1)
        // DO NOT TOUCH COMMENT BELOW
        /* verilator lint_off BLKSEQ */
        mem[i] = 32'b0;
        /* verilator lint_on BLKSEQ */
        // DO NOT TOUCH COMMENT ABOVE
    end
  end
endmodule
```

<그림 12: data_memory.v 모듈의 구성>

data_memory.v 모듈은 reset 이 1 인 경우 clock asynchronous 하게 data memory 를 초기화한다. Reset 이 1 이 아닌 경우, mem_read 가 1 일 때는 asynchronous 하게 dmem_addr 위치에 있는 데이터 값을 읽어 dout 에 저장한다. mem_write 이 1 일 때는 synchronous 하게 din 을 dmem_addr 위치에 저장한다.

- **mux.v**

```
module mux2to1 (
    input [31:0] in1,
    input [31:0] in2,
    input select,
    output reg [31:0] out);

    always @(*) begin
        if(!select) begin
            out = in1;
        end
        else begin
            out = in2;
        end
    end

endmodule
```

<그림 13: mux.v 모듈의 구성>

```
mux2to1 mux_imm (
  .in1(rs2_dout),
  .in2(imm_gen_out),
  .select(alu_src),
  .out(alu_in_2)
);
```

```
mux2to1 PCSrc1 (
  .in1(pc_4),
  .in2(pc_imm),
  .select((branch & alu_bcond) | is_jal),
  .out(reg_PCSrc1)
);

mux2to1 PCSrc2 (
  .in1(reg_PCSrc1),
  .in2(alu_result),
  .select(is_jalr),
  .out(next_pc)
);
```

```
mux2to1 mux_data(
  .in1(alu_result),
  .in2(data_dout),
  .select(mem_to_reg),
  .out(mem_to_reg_result)
);

mux2to1 mux_write_data(
  .in1(mem_to_reg_result),
  .in2(pc_4),
  .select(pc_to_reg),
  .out(rd_din)
);
```

<그림 13: cpu.v 에서 사용된 mux2to1 모듈>

mux2to1.v 모듈은 select 의 값에 따라 in1 과 in2 중 하나의 값을 반환하는 역할이다. 그림 13 에서처럼 mux_imm 에서는 alu_src 가 0 일 때, rs2_dout 을 반환하고, alu_src 가 1 일 때

imm_gen_out 을 반환한다. PCSrc1 에서는 (branch & alu_bcond) | is_jal 이 0 일 때 pc_4 를 반환하고, (branch & alu_bcond) | is_jal 이 1 일 때 pc_imm 을 반환한다. PCSrc2 에서는 is_jalr 이 0 일 때 reg_PCSrc1 을 반환하고, is_jalr 이 1 일 때 alu_result 를 반환한다. mux_data 에서는 mem_to_reg 가 0 일 때 alu_result 를 반환하고, mem_to_reg 가 1 일 때 data_dout 을 반환한다. mux_write_data 에서는 pc_to_reg 가 0 일 때 mem_to_reg_result 를 반환하고, pc_to_reg 가 1 일 때 pc_4 를 반환한다.

## Discussion

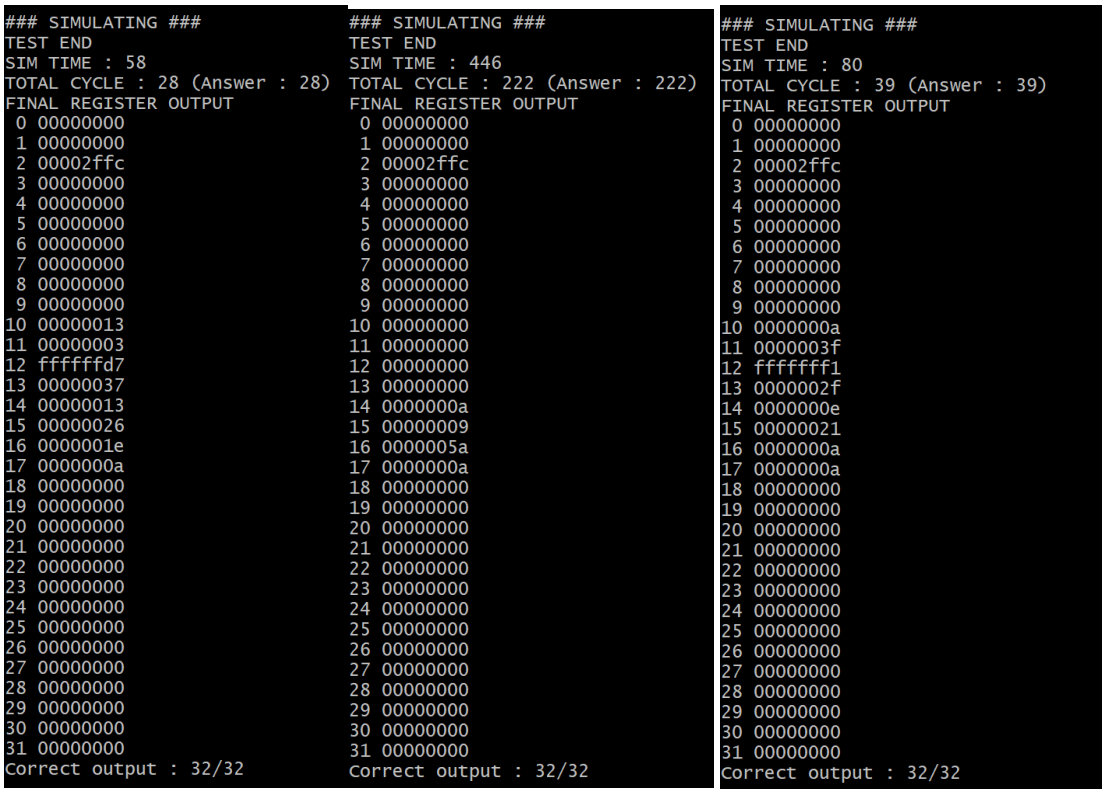
single cycle CPU 를 구현하기에 앞서 동기식과 비동기식 로직을 이해하는 것을 중점적으로 공부하였으며, 이를 기반으로 회로를 디자인하였다. 동기식(Synchronous) 로직은 모든 저장 요소가 동일한 clock 신호에 의해 트리거되는 회로이며, 순차 로직(Sequential Logic)에 주로 사용된다.

반대로, 비동기식(Asynchronous) 로직은 클록 신호와 독립적으로 작동하는 회로이다. clock 신호의 상태와 무관하게 신호의 변화가 발생할 수 있으며, 신호의 변화가 즉시 반영된다. 이런 비동기식 로직은 조합 로직(Combinational Logic)에 주로 사용된다.

각 모듈을 설계함에 있어서 동기식, 비동기식 로직을 구분하여 코드를 작성하고자 노력하였다.

## Conclusion

주어진 testbench 를 활용하여 basic_mem, loop_mem, non-controlflow_mem 에 해당하는 3 개의 테스트를 수행하였다. 그 결과, 아래와 같이 올바른 cycle 수와 register 값을 출력하는 것을 확인하였다.




<그림 14: Simulation 결과>