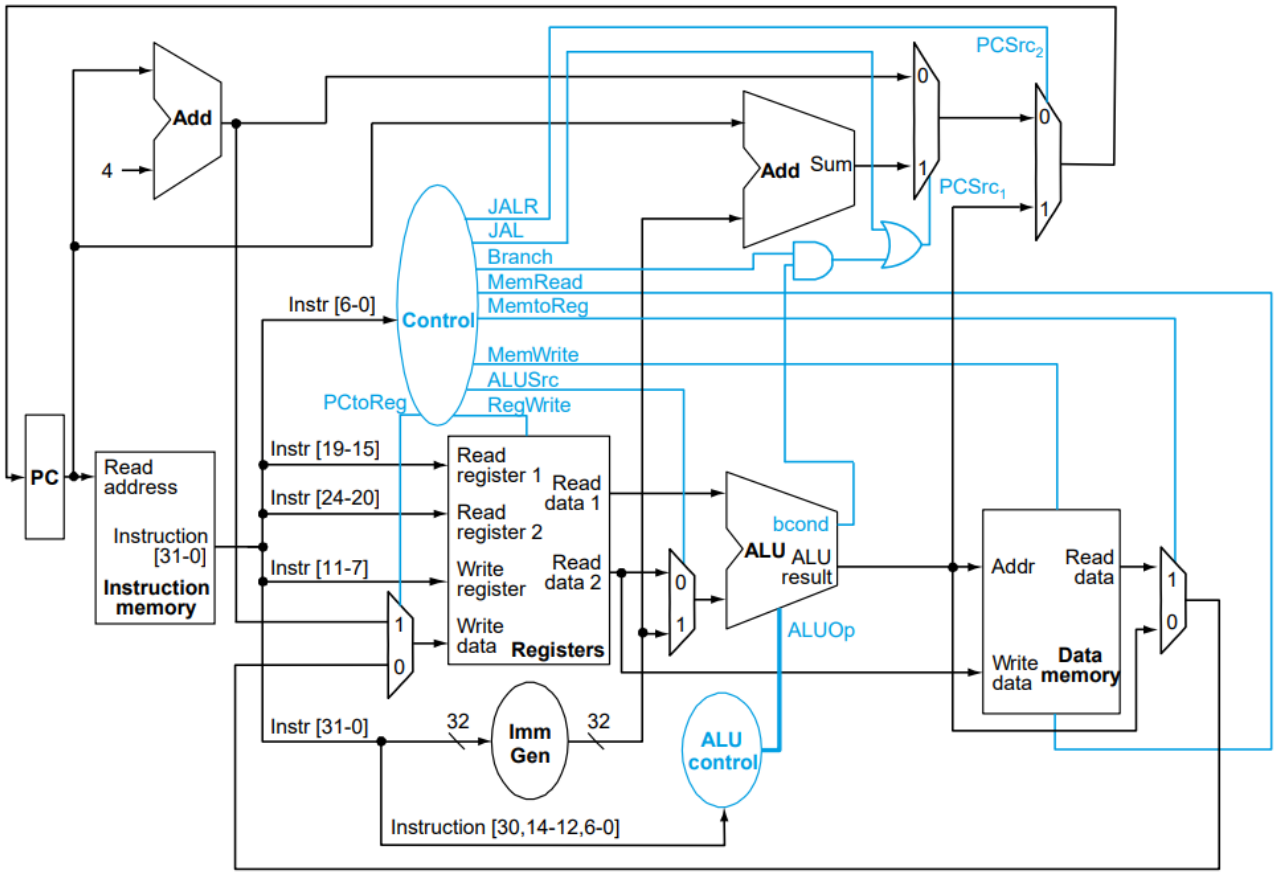
1. Introduction

   본 과제는 verilog를 이용해 간단한 single cycle cpu를 만들어 보는 과정에서 risc-v에 대한 이해와, 그것을 통해 간단한 instruction을 시뮬레이션 해 보는 것에 그 의의가 있다.

2. Design

   CPU는 Moore machine 기반의 FSM으로 구현하였다. 강의 자료에 있는 다음 CPU 설계도를 참고하였다.



cpu.v는 asynchronously하게, 다른 모듈들의 신호를 이어주는 역할만을 수행한다.

pc.v 는 clock synchronously하게, pc를 계산된 다음 pc로 바꾼다.

immediate_generator.v 또한 asynchronously하게, instruction 전체를 받아 각 instruction에 따라 risc-v 스펙 상으로 알맞은 32비트 상수를 출력한다.

instruction_memory.v는 asynchronously한 동작과, clock synchronously한 동작이 모두 포함되어 있다. Posedge clk마다 reset 신호를 확인하여 만약 reset 신호가 있다면 메모리를 초기화하는 clock synchronously한 로직과, address를 입력받아 그 주소의 메모리 값을 돌려주는 asynchronously한 로직으로 구성되어 있다.

data_memory.v, register_file.v 또한 마찬가지로 clock synchronously 한 로직과 asynchronously 한 로직으로 구성되어 있다. 두 모듈 모두 data를 read하는 부분은 address(rd)를 받아 asynchronously하게 그 address(rd)의 값을 반환하고, data를 write하는 부분은 clock synchronously하게 동작한다.

alu_control_unit.v은 alu에게 어떤 연산을 수행해야 하는 지 알려주는 모듈로, asynchronously하게 동작한다. 간단히 instruction의 일부를 받아 aluop를 out한다.

alu.v 또한 op, input1, input2를 받아 asynchrnously하게 연산을 수행한다. 이 때 만약 bcond를 설정해야 할 상황이라면 설정한다.

마지막으로 control_unit.v는 instruction의 일부를 받아 각 모듈에게 asynchronously 하게 신호를 준다.

이 CPU의 동작은 크게 5가지로 나눌 수 있다. IF -> ID -> EX -> MEM -> WB 이다.

IF는 Instruction fetch의 약자로, pc의 address의 instruction을 가져온다

ID는 Instruction 을 decode하고 operand를 fetch한다.

EX는 alu가 실제 연산을 수행하는 단계이다.

MEM은 memory에 access를 하는 단계이다.

WB는 레지스터/메모리에 다시 값을 쓰는 단계이다.

EX 단계부터는 각 instruction 마다 수행 될 수도, 수행되지 않을 수도 있다. 또한 WB의 경우는 data를 다시 써 주어야 하기에 다름 클럭에 일어나는, clock synchronously 하게 동작하며, PC 모듈에 의해 pc 또한 clock synchronously하게 올라간다. 그 외에 ID, EX, MEM 부분은 magic memory, single cycle cpu의 가정에 따라 asynchronously하게 동작한다.

3. Implementation

본 섹션에서는 각 주요 모듈의 상세 동작 원리와 구현 방식을 소개한다.

```
module pc(
    input reset,                    // input (Use reset to initialize PC. Ini-
tial value must be 0)
    input clk,                      // input
    input [31:0] next_pc,           // input
    output reg [31:0] current_pc); // output

    always @(posedge clk) begin
        if (reset) begin
            current_pc <= 0;
        end
        else begin
            current_pc <= next_pc;
        end
    end

endmodule
```

pc 모듈은 clock synchronously하게, current pc를 0으로, 그렇지 않다면 current pc를 next pc로 설정하는 동작을 수행한다. 이를 통하여 다음 clock에는 다음 pc의 동작을 수행하게 될 것이다.

```
`include "opcodes.v"

module immediate_generator (
    input [31:0] part_of_inst,
    output reg [31:0] imm_gen_out);
    always @(*) begin
        case (part_of_inst[6:0])
            `STORE : imm_gen_out = {{20{part_of_inst[31]}},
part_of_inst[31:25], part_of_inst[11:7]};
            `JAL : imm_gen_out = {{11{part_of_inst[31]}}, part_of_inst[31],
part_of_inst[19:12], part_of_inst[20], part_of_inst[30:21], 1'b0};
            `BRANCH : imm_gen_out = {{19{part_of_inst[31]}}, part_of_inst[31],
part_of_inst[7], part_of_inst[30:25], part_of_inst[11:8], 1'b0};
            default: imm_gen_out = {{20{part_of_inst[31]}},
part_of_inst[31:20]};
        endcase
    end
endmodule
```

immediate genetator는 case 문을 이용하여 asynchronously 하게 동작하도록 구현하였다. 변수명은 part_of_inst지만 실제로는 instruction 전체를 받아서, instruction의 종류에 따라서 opcode에 맞는 상수값을 만든다.

```
  // Asynchrnously read data from the memory
  assign dout = mem_read ? mem[dmem_addr] : 0;

  // Synchronously write data to the memory
  // (use dmem_addr to access memory)
  always @(posedge clk) begin
    if (mem_write) begin
      mem[dmem_addr] <= din;
    end
  end
```

instruction memory, data memory, register file은 동작이 어느정도 비슷하기에 묶어서 설명하고자 한다. 대표적으로 data memory의 코드를 가져왔다. 다들 asynchronously 하게 data를 출력하는 부분과, clock synchronously하게 data를 write하는 방법으로 크게 나눌 수 있다. register file의 경우 x0 레지스터에 읽기/쓰기를 시도할 경우 0을 반환하도록 따로 작성해 주었고, clock synchronously하게 reset 신호를 받아 초기화 하는 로직은 기존에 있는 템플릿(do not touch)를 그대로 이용하였다.

```
`include "opcodes.v"
`include "alu_func.v"

module alu_control_unit (
    input [16:0] part_of_inst,
    output reg [3:0] alu_op);

    wire [6:0] func7 = part_of_inst[16:10];
    wire [2:0] func3 = part_of_inst[9:7];
    wire [6:0] opcode = part_of_inst[6:0];

    always @(*) begin
        case (opcode)
            `ARITHMETIC: begin
                case (func3)
                    `FUNCT3_ADD: alu_op = ((func7 == `FUNCT7_OTHERS) ?
`FUNC_ADD : `FUNC_SUB);
                    `FUNCT3_SLL: alu_op = `FUNC_LLS;
                    `FUNCT3_XOR: alu_op = `FUNC_XOR;
                    `FUNCT3_OR: alu_op = `FUNC_OR;
                    `FUNCT3_AND: alu_op = `FUNC_AND;
```

위는 alu control unit의 일부이다. synchronously하게 동작하며, instruction의 opcode, func3, func7 부분을 받아 alu에서 실제로 수행할 연산을 지정한다.

```
`include "alu_func.v"

module alu (
    input [3:0] alu_op,
    input signed [31:0] alu_in_1,
    input signed [31:0] alu_in_2,
    output reg [31:0] alu_result,
    output reg alu_bcond);

    always @(*) begin
        case(alu_op)
            `FUNC_ADD: begin
                alu_result = alu_in_1 + alu_in_2;
                alu_bcond = 0;
            end
            `FUNC_SUB: begin
                alu_result = alu_in_1 - alu_in_2;
                alu_bcond = 0;
            end
            `FUNC_BEQ: begin
```

위는 alu의 일부이다. 이 또한 synchronously하게 작동하는데, 일반적으로는 연산을 수행해 alu_result 로 out하지만 bcond를 설정해야 하는 상황이라면(이는 alu_op에 의해 결정된다.) alu_result 가 아닌 bcond 를 설정한다. 각 경우 default는 편의상 0으로 지정하였다. (사용되지 않기에 상관없다.)

```
`include "opcodes.v"
module control_unit (
    input [6:0] part_of_inst,
    output is_jal,
    output is_jalr,
    output branch,
    output mem_read, // MemRead
    output mem_to_reg, // MemtoReg
    output mem_write, // MemWrite
    output alu_src, // ALUSrc
    output write_enable, // RegWrite
    output pc_to_reg,
    output is_ecall); // (ecall inst)
    assign write_enable = (part_of_inst != `STORE) && (part_of_inst !=
`BRANCH);
    assign alu_src = (part_of_inst != `ARITHMETIC) && (part_of_inst !=
`BRANCH);
    assign mem_read = (part_of_inst == `LOAD);
    assign mem_write = (part_of_inst == `STORE);
    assign mem_to_reg = (part_of_inst == `LOAD);
    assign pc_to_reg = (part_of_inst == `JAL) || (part_of_inst == `JALR);
    assign is_jal = (part_of_inst == `JAL);
    assign is_jalr = (part_of_inst == `JALR);
    assign branch = (part_of_inst == `BRANCH);
    assign is_ecall = (part_of_inst == `ECALL);
endmodule
```

마지막으로 control unit은 synchronously하게, inst의 부분을 받아서 각종 신호를 출력하는 역할을 수행한다. 이를 통해 각 모듈들의 상세 동작을 결정한다.

4. Discussion

x0 레지스터를 0으로 처리하는 과정에서, 구현상 먼저 떠오른 아이디어로써 값을 쓸 때에는 동일하게 쓰고, 단지 읽을 때 0으로 처리하도록 구현하였다. 그러나 추후 더 고민해 본 결과 x0레지스터는 그냥 0으로 두는 것이 레지스터 하나라는 소중한 자원을 아끼는 방법임을 알게 되었다. 과제 제출시에는 PPT상에서 x1~x31 레지스터만 검사한다고 적혀 있어 이를 반영하지 않은 상태로 제출하였는데, 생각해보면 구현이 차이가 거의 나지는 않는데 채점 받을 때에는 일반적인 구현을 하는 것이 훨씬 편할 것으로 보여 다음 번에 비슷한 상황이 생긴다면 조금 더 일반적이게 코드를 작성하는 것이 좋을 것 같다.

5. Conclusion

본 과제에서는 Verilog를 사용하여 직접 single cycle cpu를 구현하며, 그 과정에서 single cycle cpu, 그리고 risc-v에 대한 더욱 자세한 이해를 할 수 있었다. 이를 바탕으로 다음번에는 더 나아가 multi-cycle, pipelined cpu를 구현해 보며 더욱 컴퓨터 아키텍처에 대한 이해를 높일 수 있을 것이라 예상한다.