Lab3 - Multi-Cycle CPU

team21: 20200001조은국, 20200431박현빈

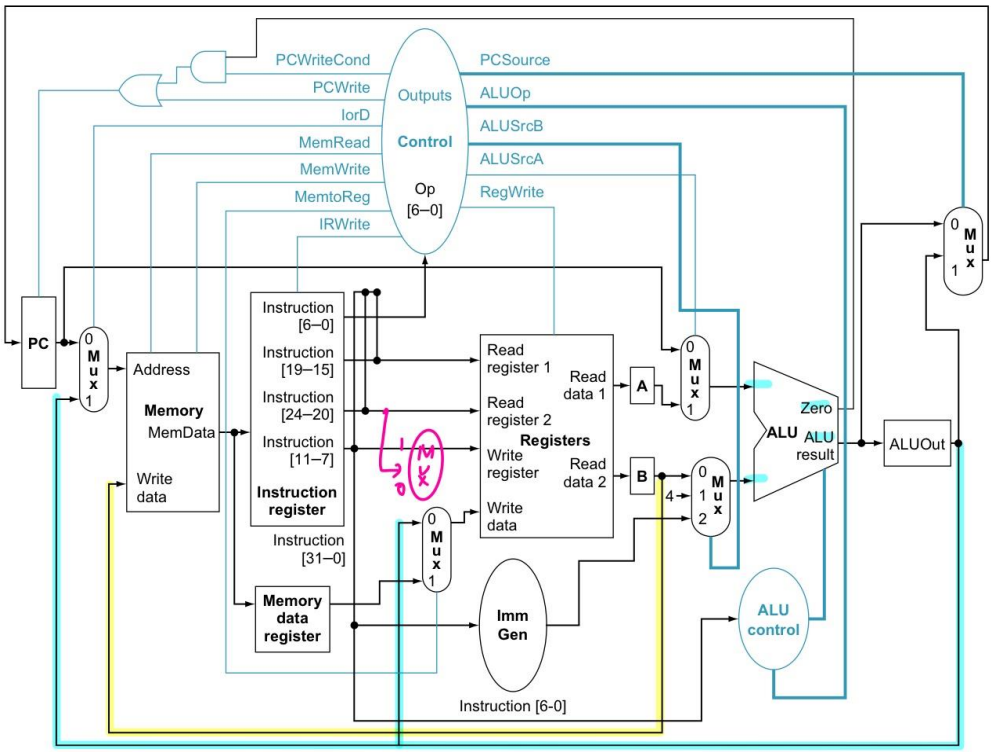
2022년 4월 19일

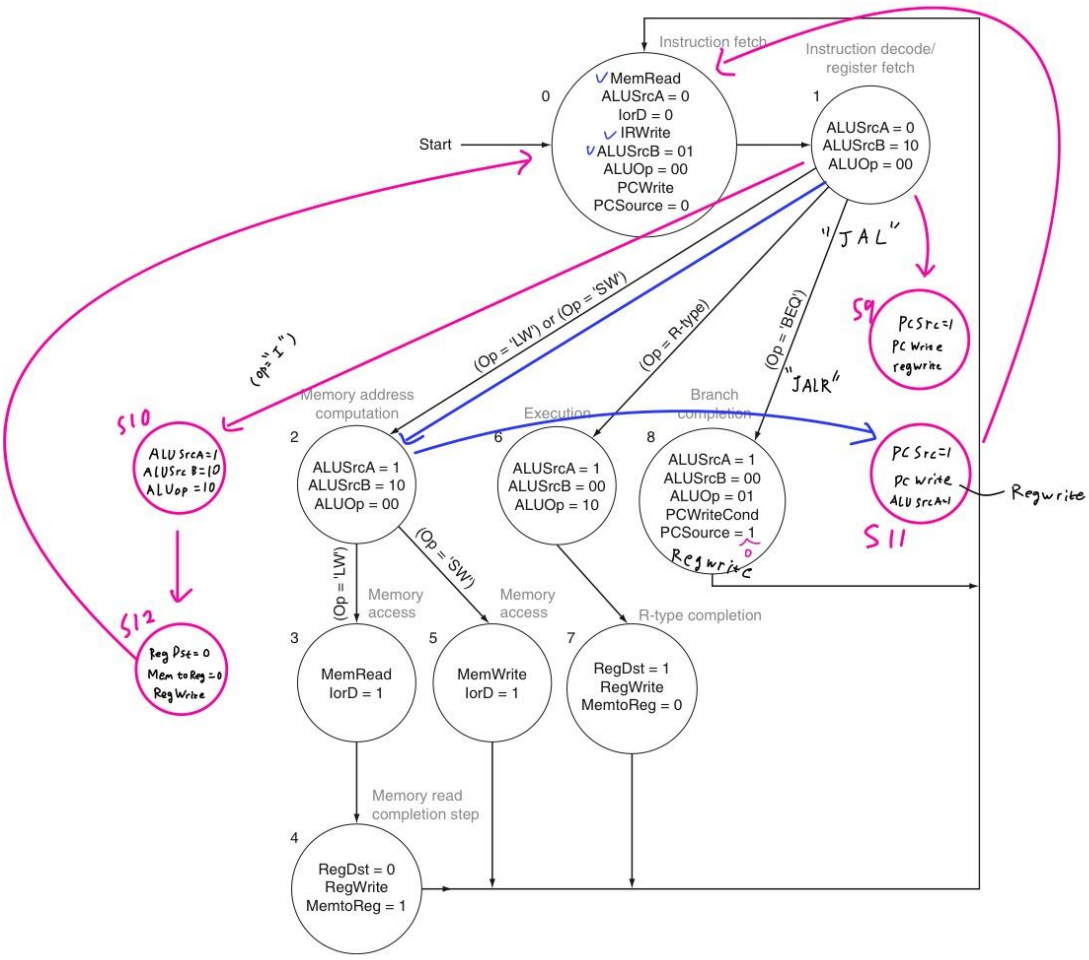
(CSED 311)

Introduction

본 Lab은 Multi-Cycle RISC-V CPU를 구현하는 프로그램을 제작하는 과제이다. PC에서 명령어를 읽어와, 명령어에 따라 다른 clock을 소비하는 고유한datapath states를 거치도록 해결하도록 구현한다.

Design


PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[6–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[6–0]
Instruction
[19–15]
Instruction
[24–20]
Instruction
[11–7]
Instruction
register
Instruction
[31–0]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
ALU
Zero
ALU
result
ALUOut
Imm
Gen
ALU
control
Instruction [6-0]



Textbook에 나오는 Multi-cycle cpu에 대한 datapath와 control flow를 참조하고, 처리 가능한 명령어 states를 일부 추가하여 verilog로 구현하는 방식으로 design하였다.

PC (control.v)

input: next_pc

output: current_pc

다음 instruction의 address를 cpu의 계산 결과로부터 입력받아 출력한다.

Memory(memory.v)

input: address

output: Memory Data

pc를 address로 받아 그에 해당하는 instruction을 출력한다.

control에 따라 address에 해당하는 data memory의 값을 출력하거나, 해당 address의 memory에 특정 register의 값을 write한다.

* IF, WB 담당

Register(RegisterFile.v)

input: register 1,2, register direction, rd_din / write_enable

output: rs1_dout, rs2_dout

입력으로 받은 register에 해당하는 값을 출력하거나, control에 따라 register에 입력으로 받은 값을 write한다.

*instruction decode stage, write back stage 담당

ALU control(operator.v)

input: part_of_inst

output: alu_op

instruction의 일부를 입력으로 받아 alu에서 수행해야 하는 연산에 대한 control을 출력한다.

Clock Asynchronous

ALU(operator.v)

input: alu_op, alu_in_1, alu_in_2

output: alu_result, alu_bcond

control에 따라 입력받은 두 값에 대한 연산을 수행하고, 연산 결과를 출력한다.

*execute stage 담당

MUX(operator.v)

input: control, input1, input2

output: out

control에 따라 입력받은 두 값 중 하나만을 출력한다.

Immediate generator(control.v)

input: instruction[31:0]

output: immediate value

instruction에서 immediate부분을 도출하고, sign-extension하여 반환한다.

A, B, MDR, ALUOut (cpu.v)

clock synchrinous하게 update되도록 design하여 한 instruction의 여러 MicroStage가 정보를 저장해 놓게끔 한다.

```
always @(posedge clk) begin
    if (IRWrite || reset) begin
        IR <= IR_wire;
    end
    A <= A_wire;
    B <= B_wire;
    ALUOut <= alu_result;
    MDR <= MDR_wire;
end

assign opcode = IR[6:0];
assign IR_wire2 = IR;
assign A_Out = A;
assign B_Out = B;
assign ALUOut_Out = ALUOut;
assign MDR_Out = MDR;
```

Control Unit(control.v)

input: Opcode

output: 각각의 control signal bit (0 or 1)

opcode에 따라 여러 모듈(MUX, reg, mem 등)에 wire되는 control bit를 통제한다.

*instruction decode stage 담당

Control Unit은 2가지 아래의 모듈로 나뉜다.

1. State Control

   Microcode 단위로 instruction을 처리하기 위해 instruction input과 current stage에

따라 앞에서 첨부한 FSM대로 동작하도록 stage를 나누고, stage transition을 결정한다.

2. Cotrol_bit Setting

   앞에서 결정한 FSM state에 따른 Microcode Outputs(control bit)를 결정한다.

1. State Control 코드 첨부

```
module ControlUnit(
    input [6:0] part_of_inst,
    input clk,
    input reset,
    input [4:0] Curr_State_in,
    input bcond,
    output [4:0] Next_State_out
    );

    reg [4:0] Curr_State;
    reg [4:0] Next_State;
    reg [6:0] opcode;
    assign Next_State_out = Next_State;
    // bit-setting for each state

    //next-state, transition

   always @(*) begin
    Curr_State <= Curr_State_in;
    opcode <= part_of_inst;

            if(reset) begin
                    // 0 to 32'b0
                    Next_State <= 5'b01111;
            end
            else if(Curr_State == 5'b01111) begin
                    Next_State <= 5'b00001;
              end
            else if(Curr_State == 5'b00000 || Next_State == 5'b10000 || Next_State == 5'b10001) begin
                    Next_State <= 5'b01111;
              end

            else if(Curr_State == 5'b00001) begin
                    if((opcode == `LOAD) || (opcode == 7'b0100011)) begin
                            Next_State <= 5'b00010;
                    end
                    else if(opcode == `ARITHMETIC) begin
                            Next_State <= 5'b00110;
                    end
                    else if(opcode == `BRANCH) begin
                            Next_State <= 5'b10010;
                    end
                    else if(opcode == `JAL || opcode == `JALR) begin
                            Next_State <= 5'b01001;
                    end
                    else if(opcode == `ARITHMETIC_IMM) begin
                            Next_State <= 5'b01010;
                    end
```

```
            end

            else if(Curr_State == 5'b00010) begin
                        if(opcode == `LOAD) begin
                                    Next_State <= 5'b00011;
                        end
                        else if(opcode == `STORE) begin
                                    Next_State <= 5'b00101;
                        end
            end

            else if(Curr_State == 5'b00011) begin
                        Next_State <= 5'b00100;
            end

            else if(Curr_State == 5'b00110) begin
                        Next_State <= 5'b00111;
            end

            else if(Curr_State == 5'b01010) begin
                        Next_State <= 5'b01100;
            end
            else if( Curr_State == 5'b01001) begin
                   if(opcode == `JAL) begin
                       Next_State <= 5'b01011;
                    end
                    else if(opcode == `JALR) begin
                       Next_State <= 5'b01101;
                    end
            end
            else if(Curr_State == 5'b01000) begin
                  if(bcond == 1) begin
                      Next_State = 5'b01110;
                   end
                   else if(bcond == 0) begin
                      Next_State = 5'b00000;
                   end
            end
            else if(Curr_State == 5'b01011 || Curr_State == 4'b01110) begin
              Next_State = 5'b10000;
            end
            else if(Curr_State == 5'b01101) begin
              Next_State = 5'b10001;
            end
            else if(Curr_State ==5'b10010) begin
              Next_State = 5'b01000;
            end
            else if(Curr_State == 5'b00100 || Curr_State == 5'b00101 || Curr_State == 5'b00111 || Curr_State == 5'b01100 ) begin
                        Next_State <= 4'b00000;
            end

  end

endmodule
```

2. Cotrol_bit Setting 코드 첨부

```
module ControlUnit2(
    input [4:0] Next_State,
    input [6:0] part_of_inst,
    input clk,
    input reset,
    output reg PCWriteCond,
    output reg PCWrite,
    output reg IorD,
    output reg MemRead,
    output reg MemWrite,
    output reg MemtoReg,
    output reg IRWrite,
    output reg PCSource,
    output reg [1:0] ALUOp,
    output reg AluSrcA,
    output reg [1:0] AluSrcB,
    output reg RegWrite,
    output reg RegDst,
    output reg is_ecall,
    output [4:0] Curr_State
    );


    reg [4:0] Curr_State_reg;


     always @(negedge clk) begin
        Curr_State_reg <= Next_State;

            if(part_of_inst == `ECALL) begin
                    is_ecall <= 1;
            end
            else begin
                    is_ecall <= 0;
            end

            // Complement of 'PCWriteCond' form FSM, state8
            if(Next_State == 5'b01000) begin
                    PCWriteCond <= 1;
            end
            else begin
                    PCWriteCond <= 0;
            end


            if(Next_State == 5'b00000 || Next_State == 5'b10000 || Next_State == 5'b10001) begin
                    PCWrite <= 1;
            end
            else begin
                    PCWrite <= 0;
            end


            if(Next_State == 5'b00011 || Next_State == 5'b00101) begin
                    IorD <= 1;
            end
            else begin
                    IorD <= 0;
            end


            if(Next_State == 5'b00000 || Next_State == 5'b00011 || Next_State == 5'b01111 || Next_State == 5'b00001 || Next_State == 5'b10000 || Next_State == 5'b10001 || Next_State == 5'b10010 ) begin
                    MemRead <= 1;
            end
            else begin
                    MemRead <= 0;
            end


            if(Next_State == 5'b00101) begin
                    MemWrite <= 1;
            end
            else begin
                    MemWrite <= 0;
            end
```

```
        if(Next_State == 5'b00100) begin
                MemtoReg <= 1;
        end
        else begin
                MemtoReg <= 0;
        end


        if(Next_State == 5'b00000 || Next_State == 5'b01111 || Next_State == 5'b00001 || Next_State == 5'b10000 || Next_State == 5'b10001 || Next_State == 5'b10010 ) begin
                IRWrite <= 1;
        end
        else begin
                IRWrite <= 0;
        end


        if(Next_State == 5'b01000) begin
                PCSource <= 1;
        end
        else begin
                PCSource <= 0;
        end


        if(Next_State == 5'b01000) begin
                ALUOp <= 2'b01;
        end
        else if(Next_State == 5'b00110 || Next_State == 5'b01010) begin
                ALUOp <= 2'b10;
        end
        else begin
                ALUOp <= 2'b00;
        end


        if(Next_State == 5'b00000 || Next_State == 5'b01100 || Next_State == 5'b01111 || Next_State == 5'b01001) begin
                AluSrcB <= 2'b01;
        end
        else if(Next_State == 5'b00001 || Next_State == 5'b00010 || Next_State == 5'b01010 || Next_State == 5'b01011 || Next_State == 5'b01101 || Next_State == 5'b01110 || Next_State == 5'b10000 || Next_State == 5'b10001) begin
                AluSrcB <= 2'b10;
        end
        else if(Next_State == 5'b10010) begin
            AluSrcB <= 2'b11;
        end
        else begin
                AluSrcB <= 2'b00;
        end


        if(Next_State == 5'b00010 || Next_State == 5'b00110 || Next_State == 5'b01000 || Next_State == 5'b01010 || Next_State == 5'b10001) begin
                AluSrcA <= 1;
        end
        else begin
                AluSrcA <= 0;
        end


        if(Next_State == 5'b00100 || Next_State == 5'b00111 || Next_State == 5'b01100 || Next_State == 5'b01011 || Next_State == 5'b01101) begin
                RegWrite <= 1;
        end
        else begin
                RegWrite <= 0;
        end


        if(Next_State == 5'b00111 || Next_State == 5'b01011 || Next_State == 5'b01101) begin
                RegDst <= 1;
        end
        else begin
                RegDst <= 0;
        end
end

assign Curr_State = Curr_State_reg;

endmodule
```

Implementation

PC

Current PC를 next_PC로, posedge Clock synchronous하게 update한다.

reset == 1일 경우 0으로 초기화

Memory

mem_read나 mem_write의 값에 따라 input으로 받은 address의 값을 output으로 내보내거나, input으로 받은 address에 input으로 받은 값을 저장한다.

read: Clock asynchronous

write: posedge Clock synchronous

Register

input에 해당하는 register 값을 반환하거나, write_enable에 따라 rd에 저장한다.

Read는 Clock asynchronous, Write는 posedge에 따라 Clock synchronous

posedge clock synchronous하게 rf[0] = 0이 되도록 한다.

Control Unit

opcode와 Curr_State에 따라 Next_State를 설정하고, Curr_State에 따른 Control Signal을 제어한다.

1. control unit1

   Curr_State가 바뀔 때마다 Curr_State와 opcode에 따라서 Next_State를 update한다.

2. control unit2

   negedge clock synchronous 하게 Curr_State를 Next_State로 update한다.

ALU Control Unit

input으로 들어오는 ALUOP과 opcode, funct7, fuct3 에 따라 알맞은 alu_op을 output으로 출력한다. (clock asynchronous)

Immediate generator

if문으로 Opcode에 따라 immediate value part를 찾는다.

sign extension을 위해, 최고차항의 비트를 반복하여 32 bit를 채운다. 가령 12 bit imm_value의 최고차항이 1인 경우, 앞의 20 bit를 1로 채운다.

Clock asynchronous

ALU

alu_op에 따라 and, sub, or, xor, sll, srl을 계산하여 alu_result를 반환한다.

또는 alu_op에 따라 eq, ne, lt, ge를 계산하여 alu_bcond로 반환한다.

eq, ne, lt, ge 계산에는 SF와 OF라는 condition code를 활용한다.

ALU는 각 state마다 발생하는 opcode와 mux를 통해 들어오는 input에 따라서

1. pc+4, pc+ immediate, register value + immediate등 next_pc 혹은 store, load에 관한 계산 수행
2. R-Type, I-Type execution 수행
3. branch condition 출력

등 resoursce reuse가 되도록 하였다.

Clock asynchronous

MUX

다른 모듈 구현 보조를 위해 구현한다.

A, B, ALUOUT 등 register

posedge clock synchronous하게 wire 값을 받아 저장한다.

Discussion

posedge clock synchronous 하게 register, pc, microcode를 전부 통일하다 보니 순서가 엉키는 상황이 발생했다. 그래서 microcode는 negedge clock synchronous 하게 변화하도록 하였다. 또한 이런 타이밍 문제 때문에 FSM에서 state 0 다음에 state f를 거쳐 state 1로 가도록 하였다.

register.v 파일을 수정하면 안된다고 하였지만, rf[0]이 0이 되도록 할 방법을 생각해내지 못해

register.v 파일을 수정하였다.

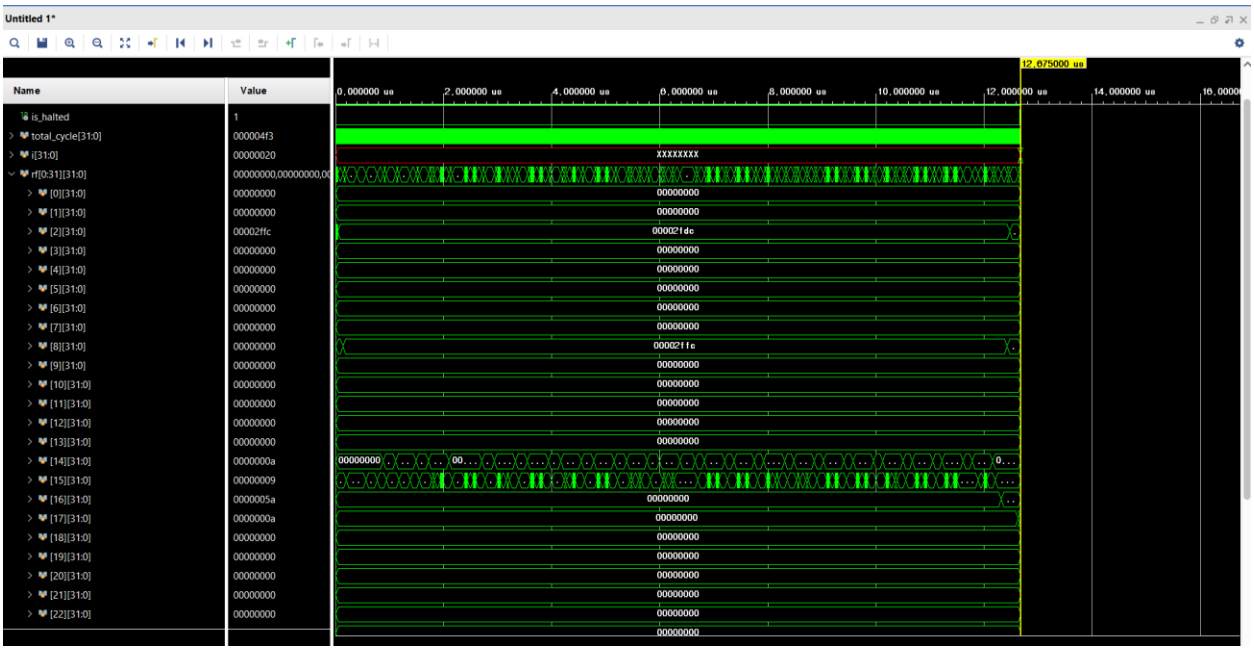
Why is a multi-cycle CPU better?

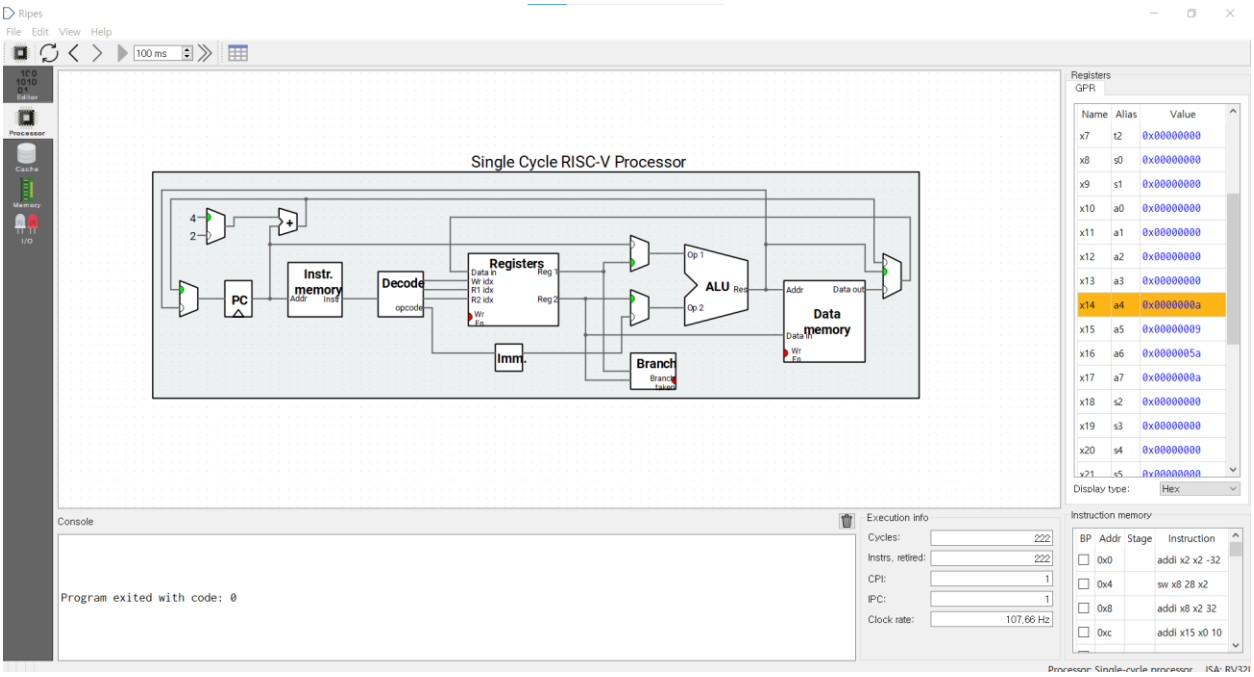
single-cycle cpu는 일괄적인 하나의 clock에 모든 명령어를 처리하는 반면, multi-cycle cpu는 instruction의 연산 시간에 따라 다른 수의 clock을 소비한다. 그렇기에 single-cycle cpu의 단위 clock은 longest instruction이기에 항상 instruction당 longest instruction latency의 시간을 소비하는 반면, Multi-cycle cpu는 작은 단위 clock을 정의하여 각 instruction의 latency와 유사한 시간을 소비하도록 디자인 할 수 있다. 즉, 빠르다.

Conclusion

1. 실험 결과

   여러 testbench code로 ripes의 결과와 비교해 보았을 때, 구현한 CPU가 정상적으로 작동함을 확인할 수 있다.





2. 결론

이번 Lab-Session에서는 instruction에 따라 다른 clock에 처리하는 Multi-Cycle CPU를 구현하였다. Multi-Cycle이 동작하는 Microcode control 방식과 이에 따른 memory의 저장, data reading, ALU calculation, PC change 등을 구현하며 배울 수 있었다.또한 각 module을 구현하며 clock synchronous와 clock asynchronous의 개념을 이해하고, verilog 문법으로 표현하는 방법도 익힐 수 있었다.