디지털논리회로 5팀 miniCPU 설계

Verilog를 활용한 miniCPU 제작

20200628 경제학과 강석민
20202266 물리학과 박준우
20226073 교류학과 신윤규

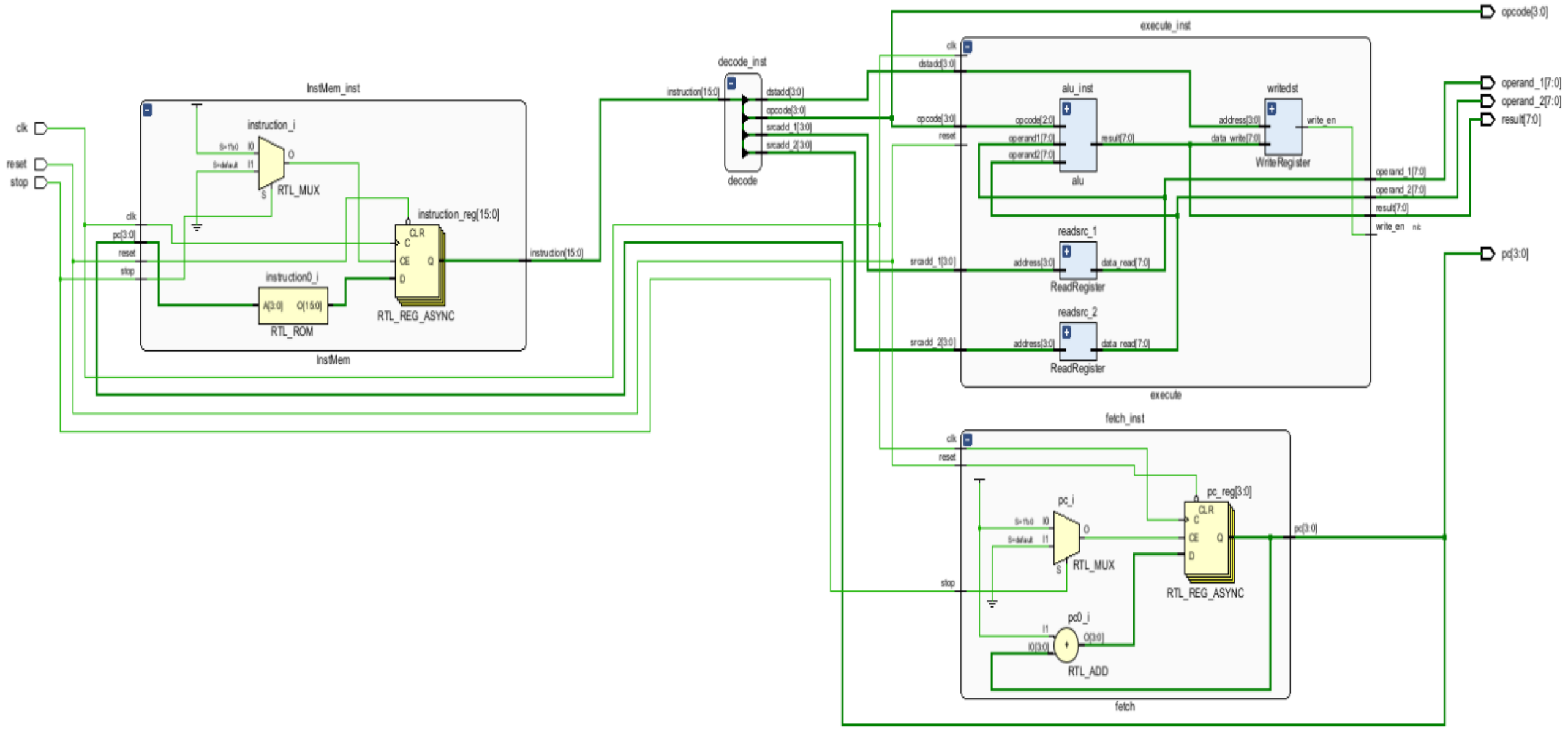
목차

개요

본 프로젝트의 목표는 간단한 16-bit CPU를 설계하고 구현하는 것입니다.

이 CPU는 Fetch, Decode, Execute의 동작을 수행하며, 각 단계는 독립적으로 동작합니다. 산술 및 논리 연산을 지원하며, 간단한 명령어 세트를 통해 기본적인 연산을 수행할 수 있도록 설계되었습니다.

회로도


clk
reset
stop
InstMem_inst
instruction_i
RTL_MUX
instruction_reg[15:0]
pc[3:0]
instruction0_i
RTL_ROM
RTL_REG_ASYNC
InstMem
instruction[15:0]
decode_inst
dstadd[3:0]
opcode[3:0]
srcadd_1[3:0]
srcadd_2[3:0]
decode
execute_inst
alu_inst
alu
writedst
WriteRegister
readsrc_1
ReadRegister
readsrc_2
operand_1[7:0]
operand_2[7:0]
result[7:0]
write_en
execute
fetch_inst
pc_i
pc_reg[3:0]
pc0_i
RTL_ADD
fetch

CPU 구성 요소 및 코드 설명

Opcode

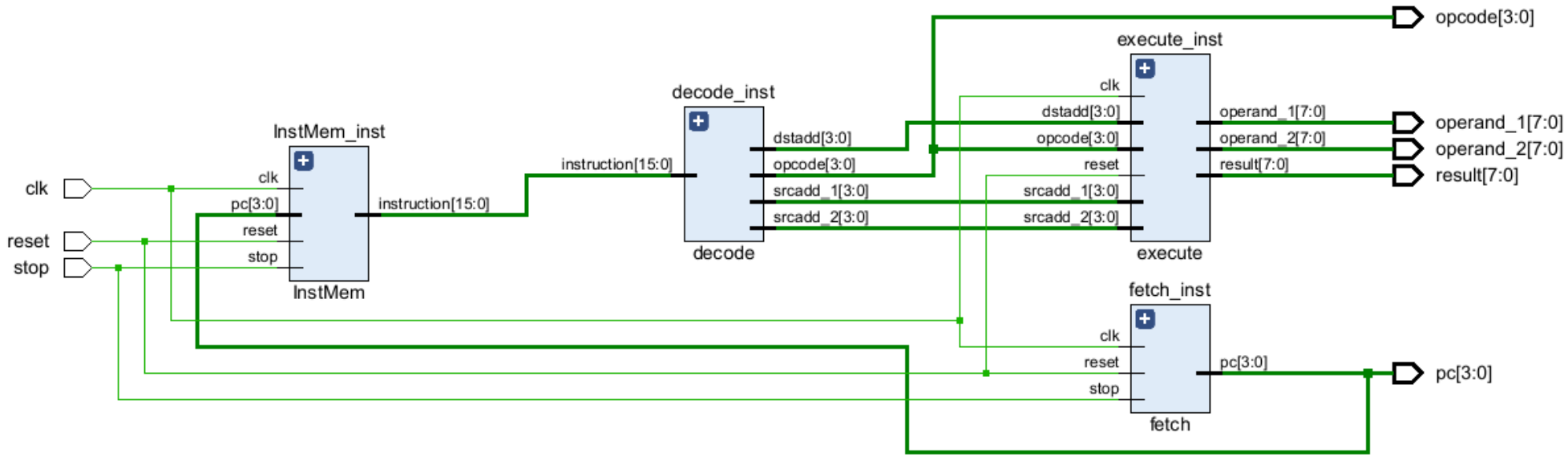
각 명령어의 Opcode를 다음과 같이 정의했습니다.

- 0000: ADD
- 0001: SUB
- 0010: NOT
- 0011: AND
- 0100: OR
- 0101: XOR
- 0110: Shift left
- 0111: Shift left

CPU 구성 요소 및 코드 설명

top.v: 전체 프로세서 시스템을 통합하는 최상위 모듈입니다. 다른 하위 모듈들을 인스턴스화하고 상호 연결하는 역할을 합니다.

fetch, InstMem, decode, execute 모듈을 연결하여 명령어의 전체 실행 사이클을 수행합니다.



CPU 구성 요소 및 코드 설명

top.v: 전체 프로세서 시스템을 통합하는 최상위 모듈입니다. 다른 하위 모듈들을 인스턴스화하고 상호 연결하는 역할을 합니다.

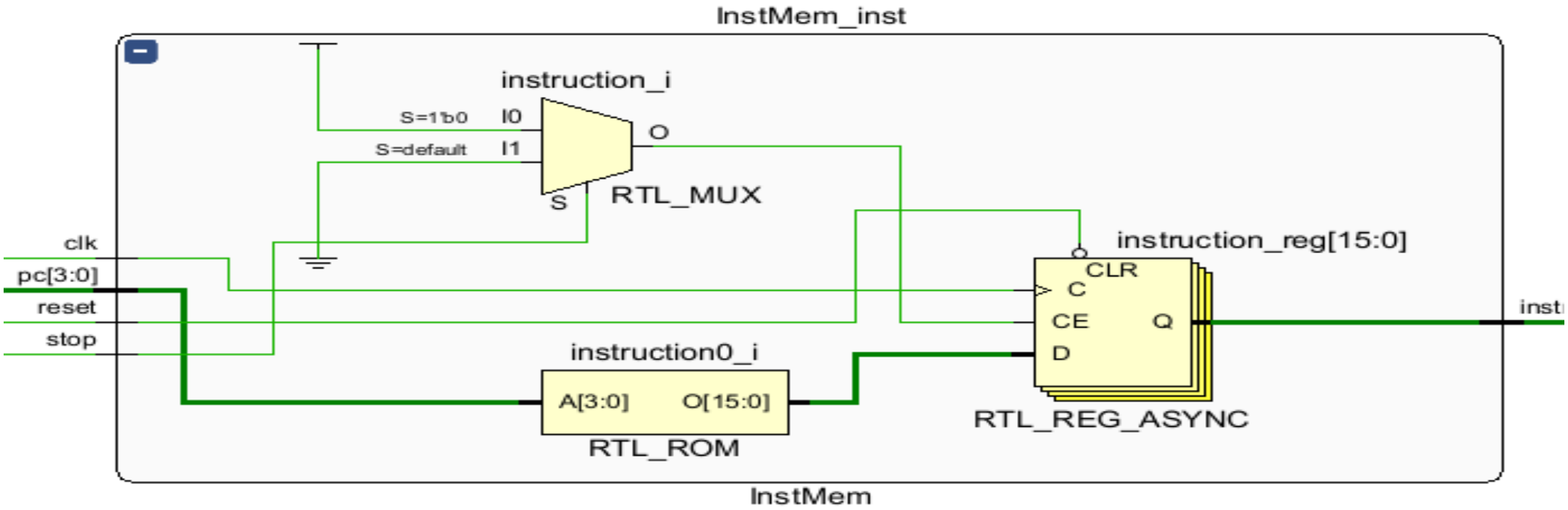
fetch, InstMem, decode, execute 모듈을 연결하여 명령어의 전체 실행 사이클을 수행합니다.

```
module Top(
    input   clk,              // 클럭 입력
    input   reset,            // 리셋 신호 입력
    input   stop,             // 중지 신호 입력

    output [3:0] opcode,
    output [7:0] operand_1,
    output [7:0] operand_2,
    output [7:0] result,
    output [3:0] pc
);

wire [15:0] instruction;
wire [3:0] srcadd_1, srcadd_2, dstadd;
wire write_en;

fetch fetch_inst(
    .clk(clk),              // 클럭 입력
    .reset(reset),            // 리셋 신호 입력
    .stop(stop),        // 멈춤 신호 입력
    .pc(pc)  // 프로그램 카운터 출력
);

InstMem InstMem_inst(
    //input
    .clk(clk),                // 클럭 입력
    .reset(reset),              // 리셋 신호 입력
    .stop(stop),               // 중지 신호 입력
    .pc(pc),           // 프로그램 카운터 입력 (4비트)

    //output
    .instruction(instruction) // 명령어 출력
);
```

```
decode decode_inst(
    //input
    .instruction(instruction),  // 입력 명령어

    //output
    .opcode(opcode),        // 출력 opcode
    .srcadd_1(srcadd_1),      // 출력 첫 번째 소스 주소
    .srcadd_2(srcadd_2),      // 출력 두 번째 소스 주소
    .dstadd(dstadd)         // 출력 대상 주소
);

execute execute_inst(
    //input
    .clk(clk),              // 클럭 입력
    .reset(reset),            // 리셋 신호 입력
    .opcode(opcode),    // 명령어 opcode 입력
    .srcadd_1(srcadd_1),  // 첫 번째 소스 데이터 입력
    .srcadd_2(srcadd_2),  // 두 번째 소스 데이터 입력
    .dstadd(dstadd),     // 대상 주소 입력 //FM decode

    //output
    .operand_1(operand_1),
    .operand_2(operand_2),
    .result(result),
    .write_en(write_en)
);

endmodule
```

CPU 구성 요소 및 코드 설명

InstMem.v: 프로세서의 명령어 메모리를 모델링한 모듈입니다. 프로그램 코드가 저장되어 있으며, 필요한 명령어를 fetch 모듈에 제공합니다.



CPU 구성 요소 및 코드 설명

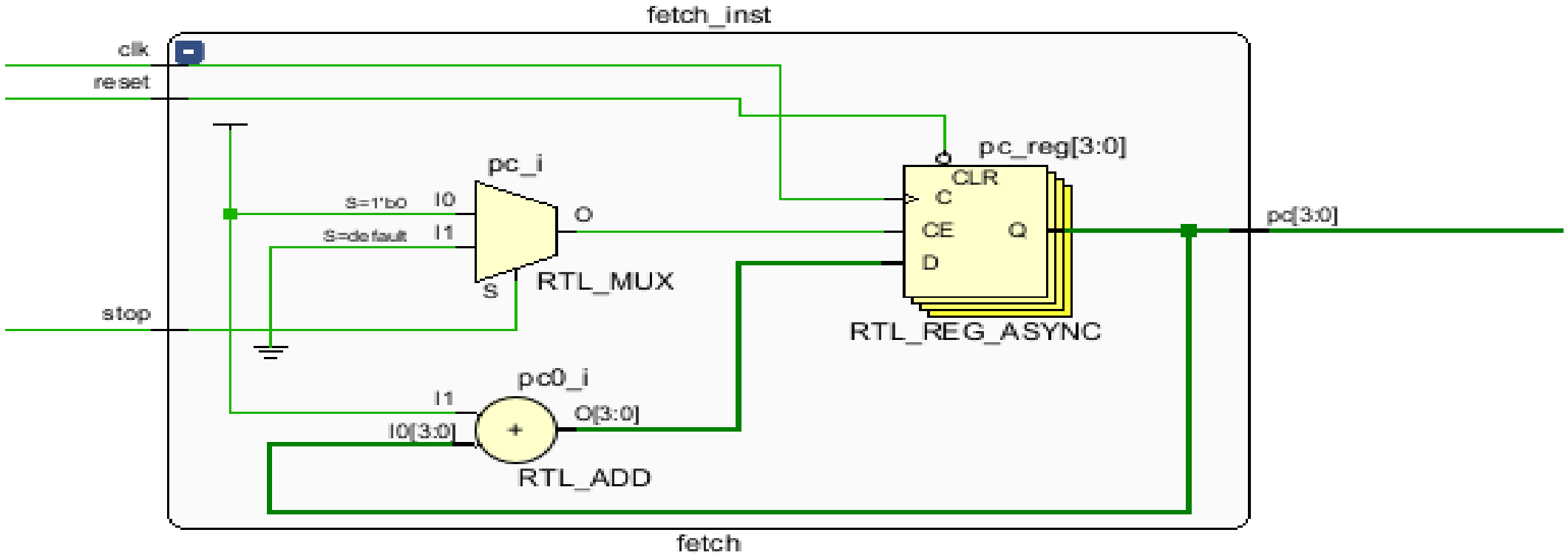
InstMem.v: 프로세서의 명령어 메모리를 모델링한 모듈입니다. 프로그램 코드가 저장되어 있으며, 필요한 명령어를 fetch 모듈에 제공합니다.

```
module InstMem(
    input clk,                  // 클럭 입력
    input reset,                // 리셋 신호 입력
    input stop,                 // 중지 신호 입력
    input [3:0] pc,             // 프로그램 카운터 입력 (4비트)
    output reg [15:0] instruction // 명령어 출력
);

    reg [15:0] mem[15:0];     // 명령어 메모리 16bit * 16 = 32byte

    always @(posedge clk or negedge reset) begin
        if (~reset)               // 리셋 시
            instruction <= 16'h0; // 현재 명령어 레지스터를 0으로 설정
        else if (~stop)           // 중지 신호가 아닌 경우
            instruction <= mem[pc]; // 현재 프로그램 카운터 위치의 명령어를 가져옴
    end
```

```
    // 초기 명령어 설정
    initial begin
        // Instruction 예제 16개 초기화
        mem[0]  = 16'b0000_0000_0000_0000; // No operation
        mem[1]  = 16'b0001_0010_0100_0001; // ADD reg[2] + reg[1] to reg[1]
        mem[2]  = 16'b0010_0100_0010_0010; // SUB reg[4] - reg[2] to reg[2]
        mem[3]  = 16'b0011_0011_0001_0100; // NOT reg[3] to reg[4]
        mem[4]  = 16'b0100_0110_0111_0001; // AND reg[6] & reg[7] to reg[1]
        mem[5]  = 16'b0101_0101_0011_0110; // OR reg[5] | reg[3] to reg[6]
        mem[6]  = 16'b0110_0011_0100_0111; // XOR reg[3] ^ reg[4] to reg[7]
        mem[7]  = 16'b0111_0100_0100_0001; // Shift left reg[4] << 1 to reg[1]
        mem[8]  = 16'b1000_0111_0010_0001; // Shift right reg[7] >> 2 to reg[1]
        mem[9]  = 16'b1001_0011_0010_0111; // ADD reg[3] + reg[3] to reg[7]
        mem[10] = 16'b1010_0110_1010_0101; // SUB reg[6] - reg[5] to reg[5]
        mem[11] = 16'b1011_0101_0110_1100; // NOT reg[5] to reg[12]
        mem[12] = 16'b1100_0010_1011_0010; // AND reg[2] & reg[11] to reg[2]
        mem[13] = 16'b1101_0001_0010_1101; // OR reg[1] | reg[2] to reg[13]
        mem[14] = 16'b1110_0011_0111_0001; // XOR reg[3] ^ reg[7] to reg[1]
        mem[15] = 16'b1111_0111_0101_0011; // Shift left reg[7] << 5 to reg[3]
    end

endmodule
```

CPU 구성 요소 및 코드 설명

fetch.v: 명령어 메모리에서 다음 실행할 명령어를 가져오는 모듈입니다. 프로그램 카운터를 관리하고, 명령어 데이터를 decode 모듈로 전달합니다.



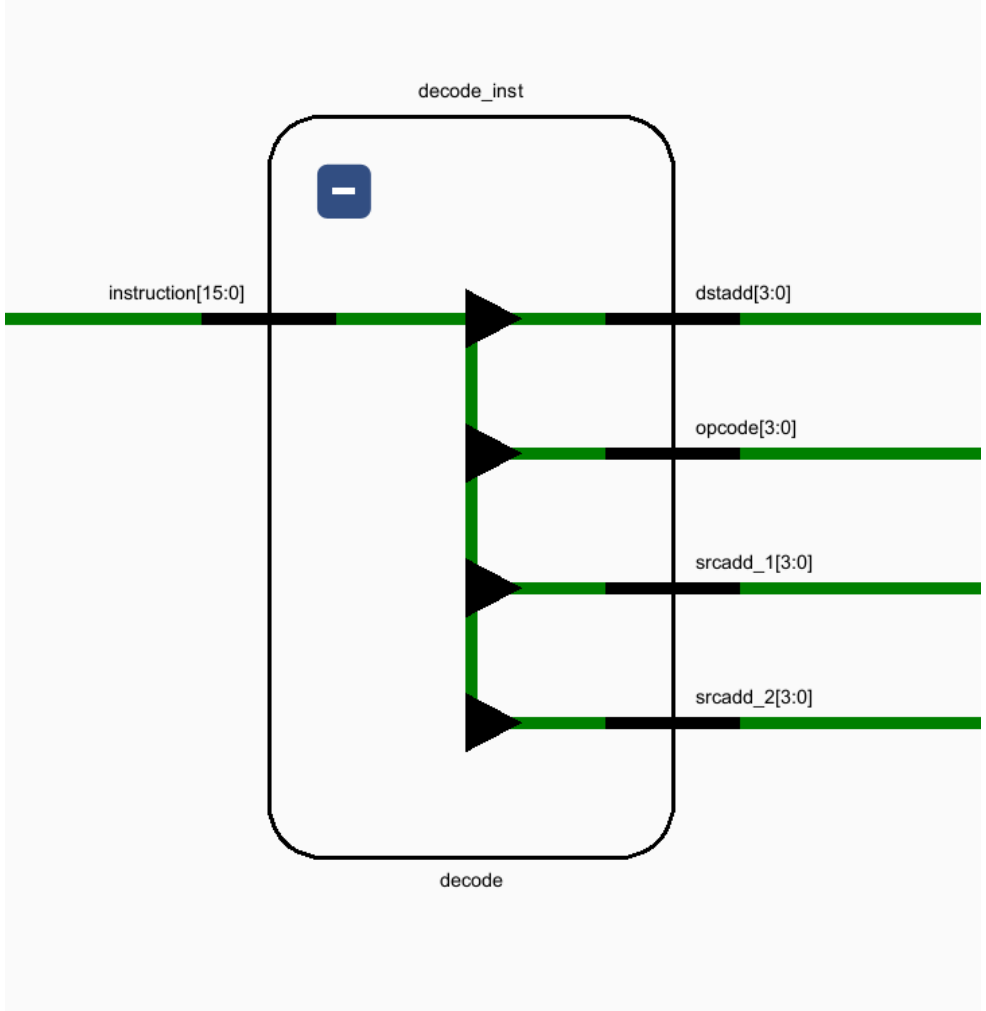
CPU 구성 요소 및 코드 설명

fetch.v: 명령어 메모리에서 다음 실행할 명령어를 가져오는 모듈입니다. 프로그램 카운터를 관리하고, 명령어 데이터를 decode 모듈로 전달합니다.

```
module fetch(
    input clk,          // 클럭 입력
    input reset,        // 리셋 신호 입력
    input stop,         // 멈춤 신호 입력
    output reg [3:0] pc  // 프로그램 카운터 출력
);

    // 항상 블록, 클럭의 상승 에지에서 동작
    always @(posedge clk or negedge reset) begin
        if (~reset) begin
            // reset 신호가 활성화되면 프로그램 카운터를 0으로 초기화
            pc <= 4'b0;
        end
        else if (~stop) begin
            // stop 신호가 비활성화된 경우 프로그램 카운터 증가
            pc <= pc + 4'b1;
        end
        // stop 신호가 활성화된 경우 프로그램 카운터 유지
    end

endmodule
```

CPU 구성 요소 및 코드 설명

decode.v: 가져온 명령어를 해석하여 실행에 필요한 정보를 추출하는 모듈입니다. 명령어 종류, 레지스터 주소, 연산 종류 등을 식별합니다.



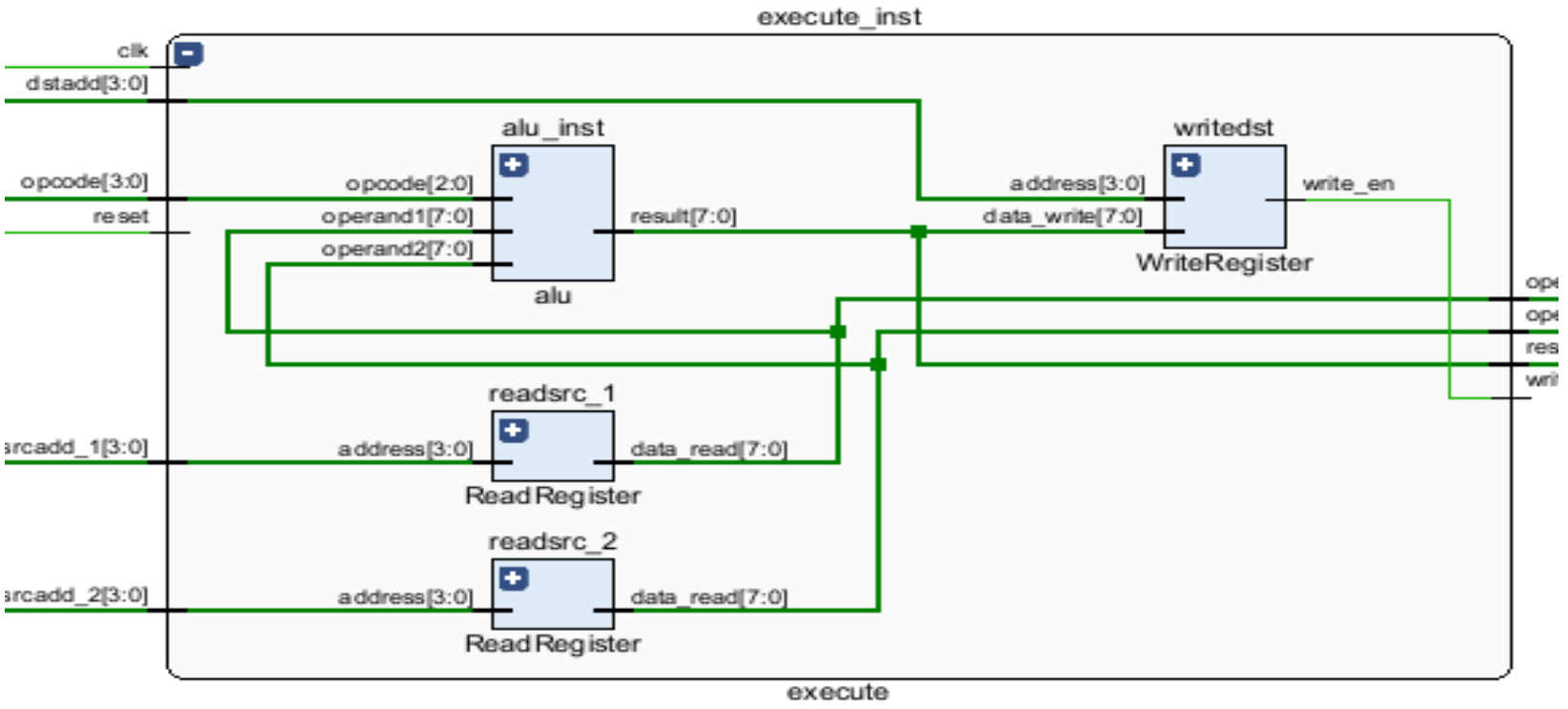
CPU 구성 요소 및 코드 설명

decode.v: 가져온 명령어를 해석하여 실행에 필요한 정보를 추출하는 모듈입니다. 명령어 종류, 레지스터 주소, 연산 종류 등을 식별합니다.

```
module decode(
    input [15:0] instruction,  // 입력 명령어
    output [3:0] opcode,       // 출력 opcode
    output [3:0] srcadd_1,     // 출력 첫 번째 소스 주소
    output [3:0] srcadd_2,     // 출력 두 번째 소스 주소
    output [3:0] dstadd        // 출력 대상 주소
);

    // 각 필드를 명령어에서 추출
    assign opcode = instruction[15:12];
    assign srcadd_1 = instruction[11:8];
    assign srcadd_2 = instruction[7:4];
    assign dstadd = instruction[3:0];

endmodule
```

CPU 구성 요소 및 코드 설명

execute.v: decode 모듈에서 전달받은 정보를 바탕으로 실제 연산을 수행하는 모듈입니다. ALU와 레지스터 파일을 사용하여 명령어를 실행합니다.



CPU 구성 요소 및 코드 설명

execute.v: decode 모듈에서 전달받은 정보를 바탕으로 실제 연산을 수행하는 모듈입니다. ALU와 레지스터 파일을 사용하여 명령어를 실행합니다.

```
module execute(
    input clk,              // 클럭 입력
    input reset,            // 리셋 신호 입력
    input [3:0] opcode,     // 명령어 opcode 입력
    input [3:0] srcadd_1,   // 첫 번째 소스 데이터 입력
    input [3:0] srcadd_2,   // 두 번째 소스 데이터 입력
    input [3:0] dstadd,     // 대상 주소 입력 //FM decode

    output [7:0] operand_1, // 초기화를 제거합니다.
    output [7:0] operand_2, // 초기화를 제거합니다.
    output [7:0] result,    // 초기화를 제거합니다.
    output write_en         // 초기화를 제거합니다.
);

    // 레지스터 모듈 인스턴스 생성 (읽기용)
    ReadRegister readsrc_1(
        .address(srcadd_1),
        .data_read(operand_1) // srcadd_1 주소에서 읽은 데이터
    );
    ReadRegister readsrc_2(
        .address(srcadd_2),
        .data_read(operand_2) // srcadd_2 주소에서 읽은 데이터
    );

    // ALU 모듈 인스턴스 생성
    alu alu_inst (
        .opcode(opcode[2:0]),   // opcode의 하위 3비트를 전달
        .operand1(operand_1),
        .operand2(operand_2),
        .result(result)
    );

    // 레지스터 모듈 인스턴스 생성 (쓰기용)
    WriteRegister writedst(
        .address(dstadd),
        .data_write(result),    // 결과 값을 쓰기 데이터로 설정
        .write_en(write_en)    // 쓰기 동작 활성화 신호
    );

endmodule
```

CPU 구성 요소 및 코드 설명

alu.v: 산술 논리 연산 장치(Arithmetic Logic Unit)를 모델링한 모듈입니다. 다양한 연산을 수행하여 결과를 execute 모듈에 전달합니다.

```
module alu (
  input  [2:0] opcode,
  input  [7:0] operand1,
  input  [7:0] operand2,
  output reg [7:0] result
);

  wire [7:0] sum;
  wire [7:0] sub;
  wire co, bo;

  // Adder and Subtractor instances
  adder_8bit adder (
    .a(operand1),
    .b(operand2),
    .ci(1'b0),
    .sum(sum),
    .co(co)
  );
```

```
  subtractor_8bit subtractor (
    .a(operand1),
    .b(operand2),
    .bi(1'b0),
    .sub(sub),
    .bo(bo)
  );

  // ALU operation selection and execution
  always @(*) begin
    case (opcode)
      3'b000: result = sum;                    // add
      3'b001: result = sub;                    // sub
      3'b010: result = ~operand1;              // not
      3'b011: result = operand1 & operand2;    // and
      3'b100: result = operand1 | operand2;    // or
      3'b101: result = operand1 ^ operand2;    // xor
      3'b110: result = operand1 << 1;          // shift left
      3'b111: result = operand1 >> 1;          // shift right
      default: result = 8'b00000000;
    endcase
  end

endmodule
```

CPU 구성 요소 및 코드 설명

register.v: 프로세서의 레지스터 파일을 모델링한 모듈입니다. 데이터 읽기/쓰기 기능을 제공하며, execute 모듈과 상호작용합니다.

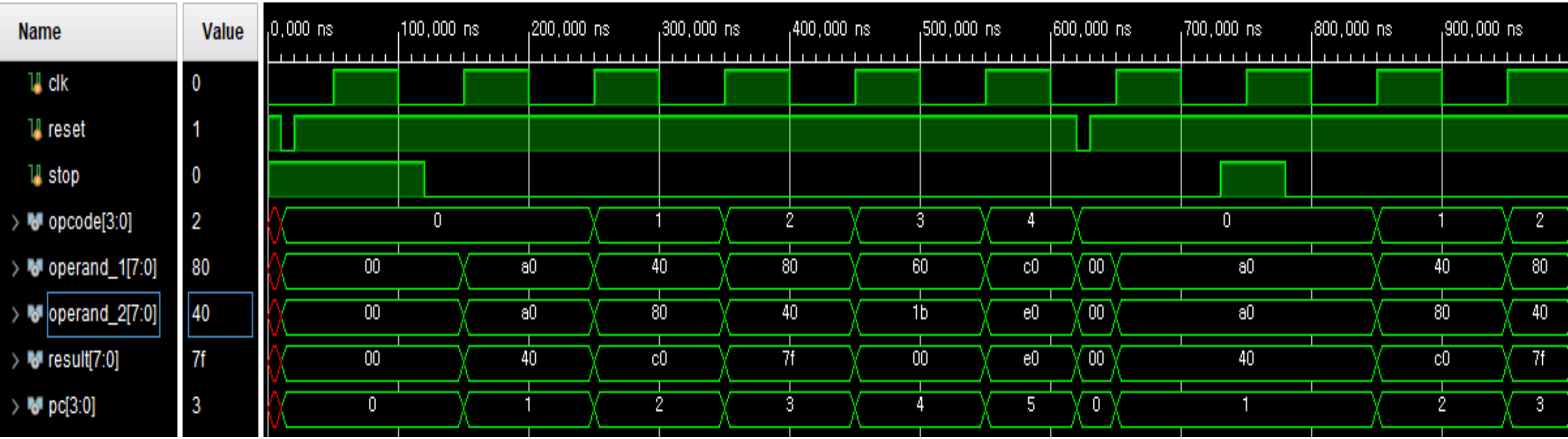
```
module Register(
    input [3:0] address,
    input [7:0] data_write,
    input write_en,
    output reg [7:0] data_read
);
    reg [7:0] regi[15:0];  // 값 메모리 1byte * 16

    // 메모리 초기화 블록

initial begin
    regi[1]  = 8'd0;
    regi[2]  = 8'd64;
    regi[3]  = 8'd96;
    regi[4]  = 8'd128;
    regi[5]  = 8'd160;
    regi[6]  = 8'd192;
    regi[7]  = 8'd224;
    regi[8]  = 8'd32;
    regi[9]  = 8'd64;
    regi[10] = 8'd96;
    regi[11] = 8'd128;
    regi[12] = 8'd160;
    regi[13] = 8'd192;
    regi[14] = 8'd224;
    regi[15] = 8'd32;
end

    // 읽기 동작 수행
    always @(address) begin
        data_read = regi[address];
    end

    // 쓰기 동작 수행
    always @(posedge write_en) begin
        if (write_en) begin
            regi[address] = data_write;
        end
    end
endmodule

module ReadRegister(
    input [3:0] address,
    output [7:0] data_read
);
Register regi_inst (
    .address(address),
    .data_write(8'b0),   // 쓰기 동작에 사용되지 않으므로 0으로 초기화
    .write_en(1'b0),     // 쓰기 동작 비활성화
    .data_read(data_read)
);
endmodule

module WriteRegister(
    input [3:0] address,
    input [7:0] data_write,
    output reg write_en
);
Register regi_inst (
    .address(address),
    .data_write(data_write),
    .write_en(write_en),
    .data_read()
);
endmodule
```

CPU 구성 요소 및 코드 설명

FDE_tb.v: 전체 프로세서 시스템의 동작을 테스트하기 위한 테스트벤치 모듈입니다. 각 모듈의 입출력을 연결하고, 시뮬레이션을 통해 프로세서의 기능을 검증합니다.

```
`timescale 1ns/1ps

module FDE_tb();

    // 입력 신호
    reg clk;
    reg reset;
    reg stop;

    // 출력 신호
    wire [3:0] opcode;
    wire [7:0] operand_1;
    wire [7:0] operand_2;
    wire [7:0] result;
    wire [3:0] pc;

    // 테스트할 CPU 모듈
    Top Top_inst(
        //input
        .clk(clk),          // 클럭 입력
        .reset(reset),      // 리셋 신호 입력
        .stop(stop),        // 중지 신호 입력

        //output
        .opcode(opcode),
        .operand_1(operand_1),
        .operand_2(operand_2),
        .result(result),
        .pc(pc)
    );

    // 초기화 및 동작 시나리오
    initial begin
        // 초기값 설정
        clk = 0; reset = 1; stop = 1;

        #10
        reset = 0;          // 리셋 비활성화

        #10
        reset = 1;          // 리셋 활성화

        #100
        stop = 0;           // 멈춤 비활성화

        #1000
        $finish;            // 테스트 종료
    end

    // 클럭 생성
    always begin
        #50
        clk = ~clk;         // 클럭 반전
    end

endmodule
```

수행 결과 및 시뮬레이션

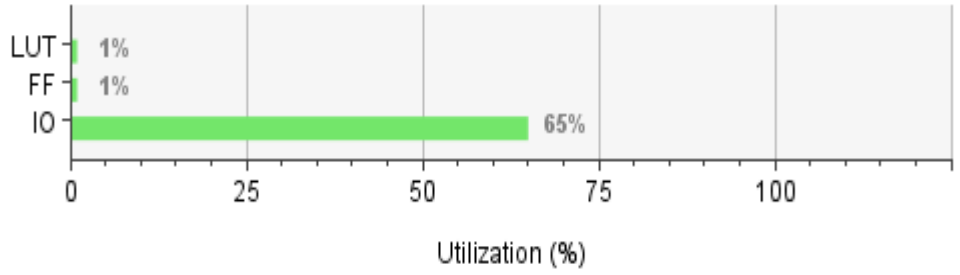


▲Vivado Simulation

수행 결과 및 시뮬레이션

Summary

Resource	Utilization	Available	Utilization %
LUT	24	17600	0.14
FF	15	35200	0.04
IO	35	54	64.81

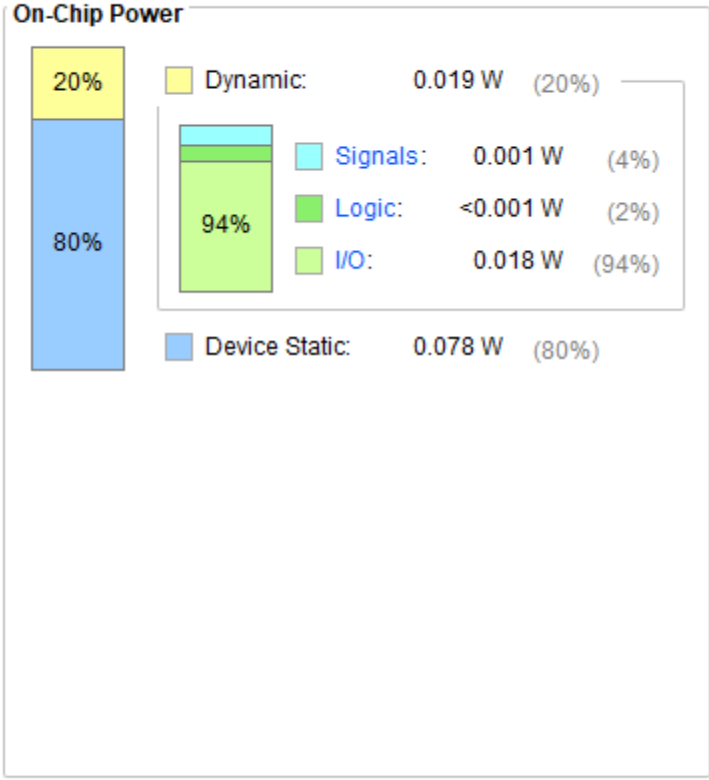


Summary

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**0.097 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**26.1°C**
Thermal Margin:	73.9°C (6.3 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	11.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

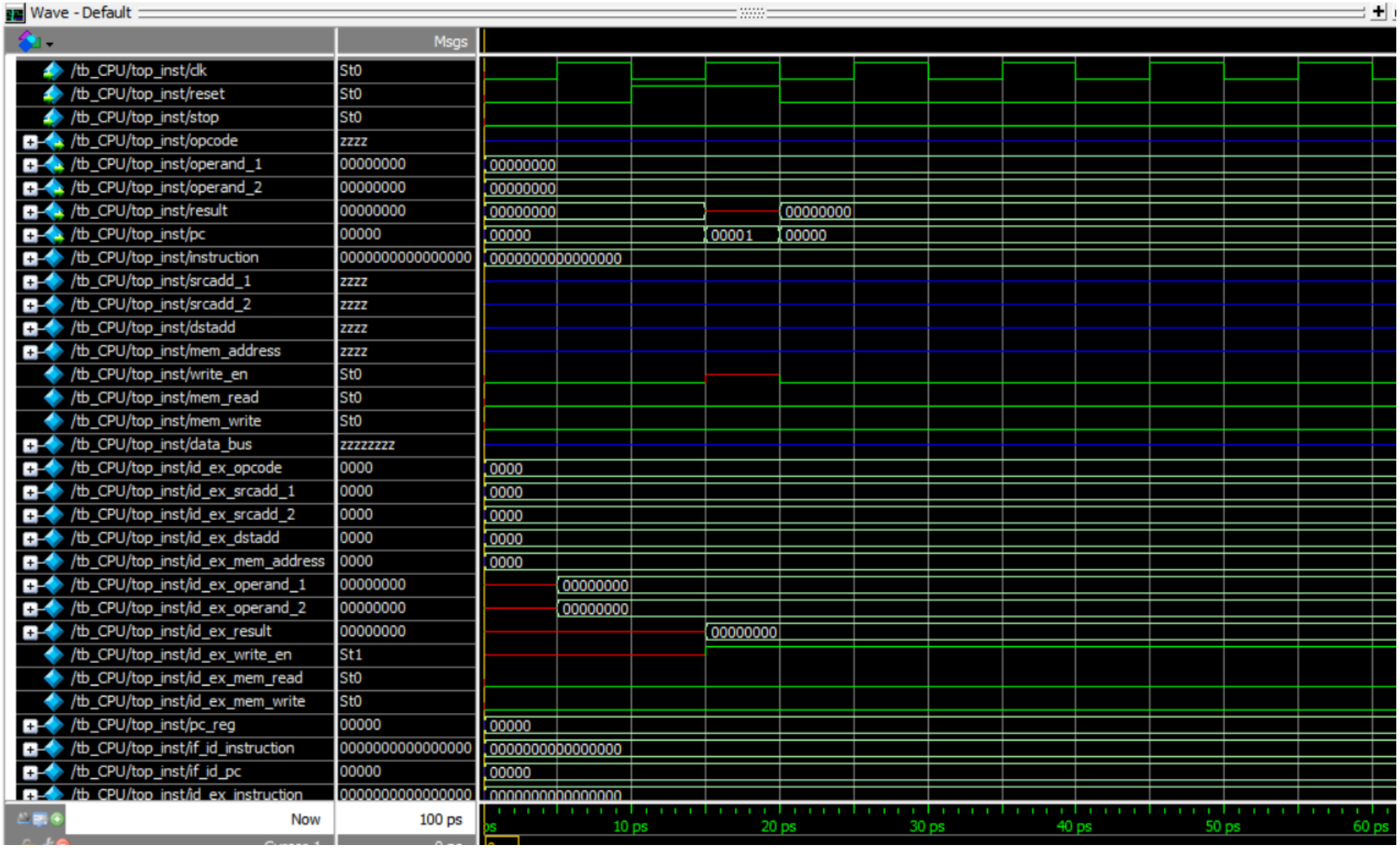
Launch Power Constraint Advisor to find and fix invalid switching activity



▲Synthesis Summary

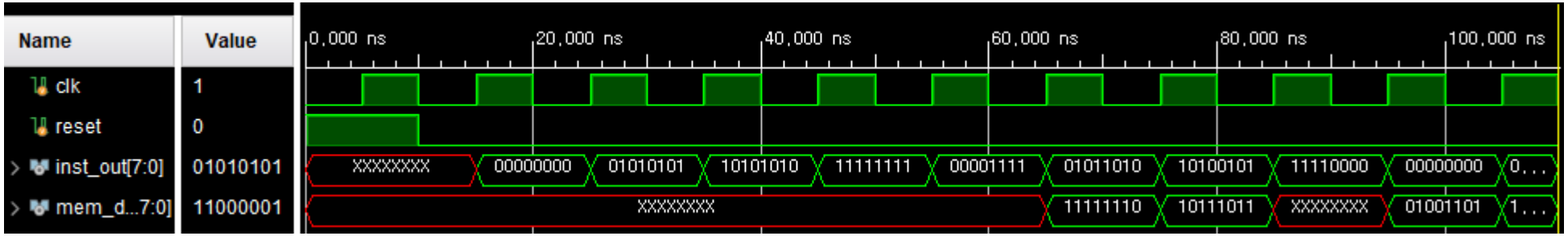
시행착오

너무 많은 모듈들이 복잡하게 구성 되어있고 그로 인해 인스턴스화 오류 및 동작 오류가 발생하였습니다.
CPU 구조 및 동작 원리에 대한 정확한 이해와 verilogHDL 사용이 미숙하여 복잡하고 다양한 기능과 동작을 수행하는 CPU를 구현하는데는 큰 어려움이 있을 것이라고 판단하였습니다.
이를 개선하기 위해서 모듈들을 간소화하고 동작 확인 후 기능 추가 및 보완하는 방식으로 프로젝트 진행하였습니다.
https://github.com/potatogyu99/cpu_2.git



◀Quartus Prime Lite로 모듈 테스트 중 모듈의 테스트벤치가 원하는 대로 작동되지 않는 모습.
개별적인 모듈의 수가 많고, 메모리 및 레지스터 접근 제어 신호가 제대로 정의되어 있지 않아서 발생한 문제.

시행착오



뤼튼 AI 과제와 업무

테스트 벤치에 $display 문을 추가하여 시뮬레이션 중에 발생하는 주요 사건들을 확인할 수 있게 하겠습니다. 이는 디버깅 과정에서 매우 유용하며, 시뮬레이션의 진행 상황을 쉽게 파악할 수 있게 해줍니다.

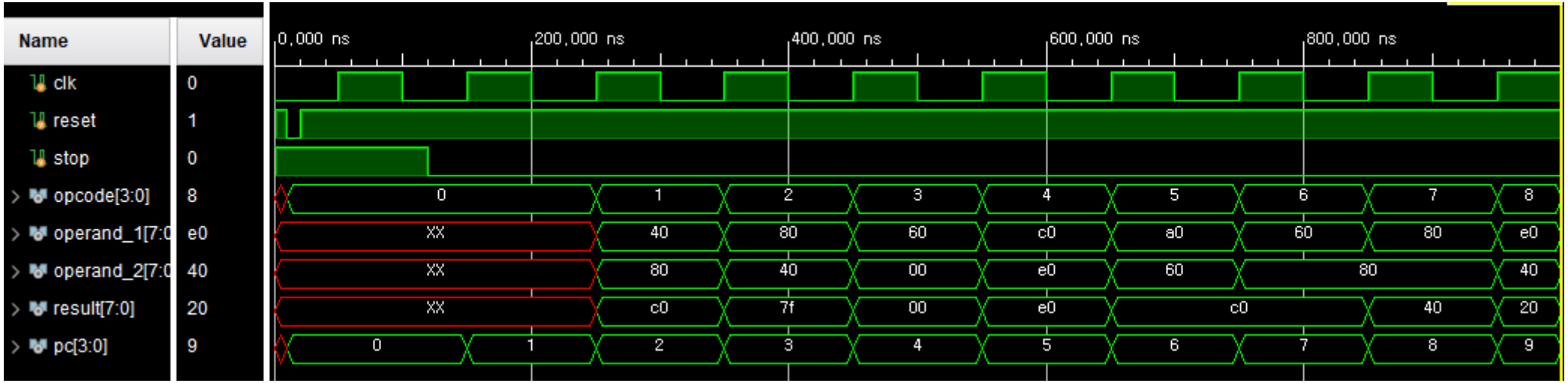
verilog

```
module testbench;
  reg clk;
  reg [2:0] opcode;
  reg [7:0] operand1;
  reg [7:0] operand2;
  wire [7:0] alu_result;
  reg [3:0] addr;
  reg [15:0] data_in;
  reg write_en;
  wire [15:0] data_out;

  // ALU instance
  alu my_alu (
```

```
시간                   5: 메모리 주소  0에 0010 저장
시간                  15: 메모리 주소  1에 0020 저장
시간                  25: 메모리 주소  2에 0030 저장
시간                  55: 메모리 주소  0에서 0010 읽음
시간                  75: 메모리 주소  1에서 xxxx 읽음
시간                  95: 연산 결과 (ADD): xx
시간                 115: 연산 결과 00xx를 메모리 주소  3에 저장
시간                 135: 메모리 주소  3에서 00xx 읽음
시간                 155: 메모리 주소  2에서 0030 읽음
시간                 175: 연산 결과 (SUB): xx
시간                 195: 최종 연산 결과 00xx를 메모리 주소  4에 저장
main.v:178: $finish called at 205 (1s)
```

▲GPT 사용, 코드 실행 중에 빈번히 발생했던 자잘한 오류들. Verilog 언어의 기본 지식이 부족하다 보니 디버깅에 시간이 많이 소요되었습니다.

시행착오



▲pc=0 일때 덧셈 연산을 수행하여야 하는데 수행되지 않는 문제.
Transistor.v 모듈에서 regi[0]이 정의되지 않아 발생한 문제로, regi[0]의 초기값을 0으로 하여 이를 수정하였습니다.