디지털논리회로 5팀 최종 보고서

Verilog를 활용한 미니CPU 제작

20200628 경제학과 강석민

20202266 물리학과 박준우

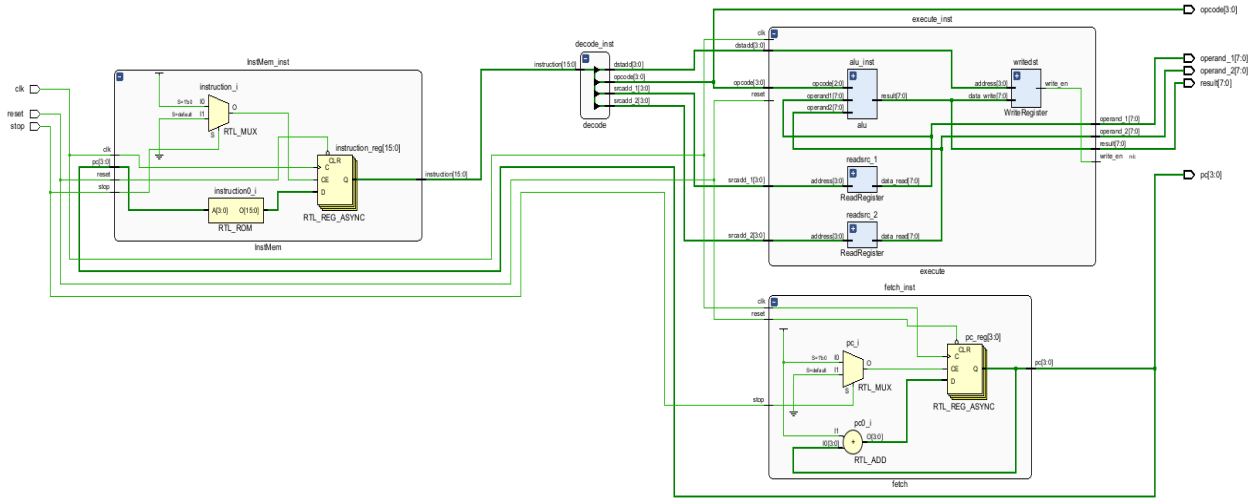
20226073 교류학과 신윤규

목차

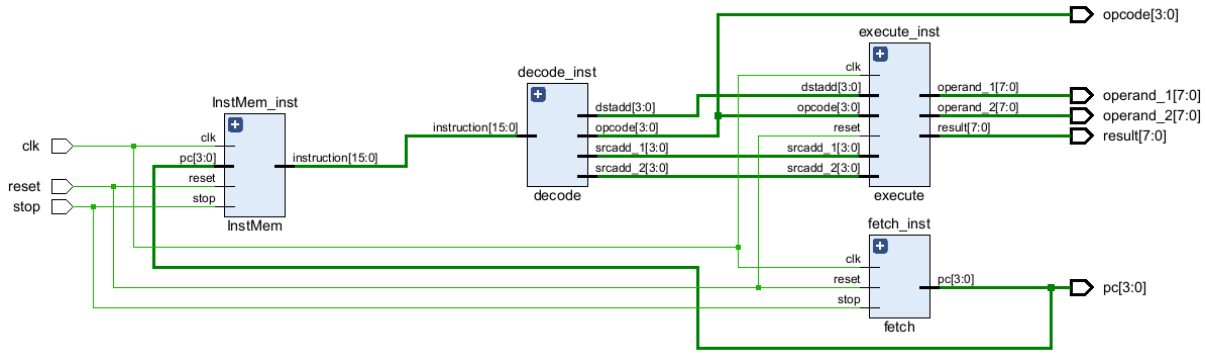
1. 프로젝트 개요 및 목표

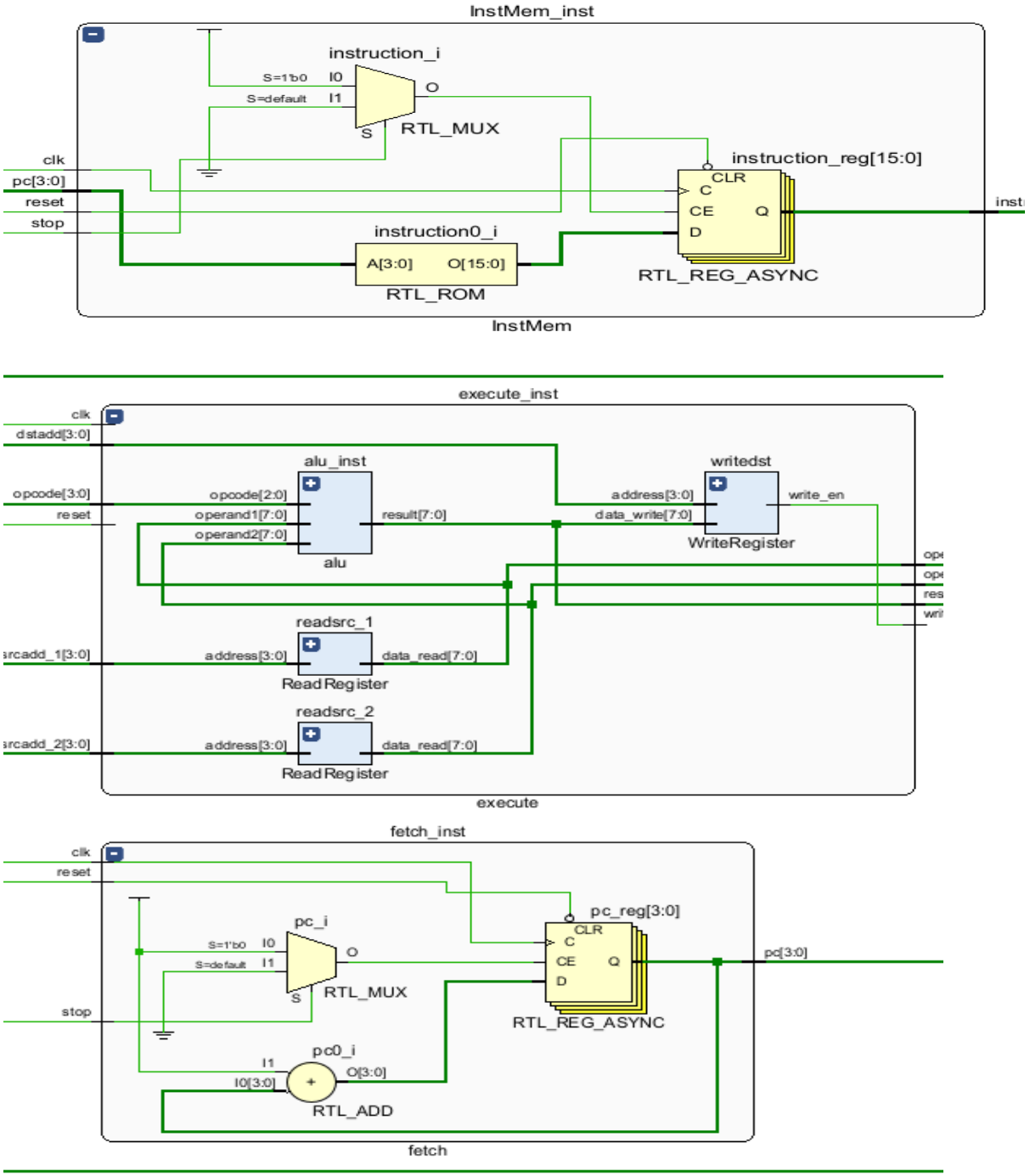
본 프로젝트의 목표는 간단한 16-bit CPU를 설계하고 구현하는 것입니다. 이 CPU는 Fetch, Decode, Execute의 동작을 수행하며, 각 단계는 독립적으로 동작합니다. 산술 및 논리 연산을 지원하며, 간단한 명령어 세트를 통해 기본적인 연산을 수행할 수 있도록 설계되었습니다.

2. 회로도



▲Vivado Schematic





▲Vivado Schematic

3. 프로젝트 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

모듈 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
```

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
```

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
```

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
```

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
```

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
```

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
```

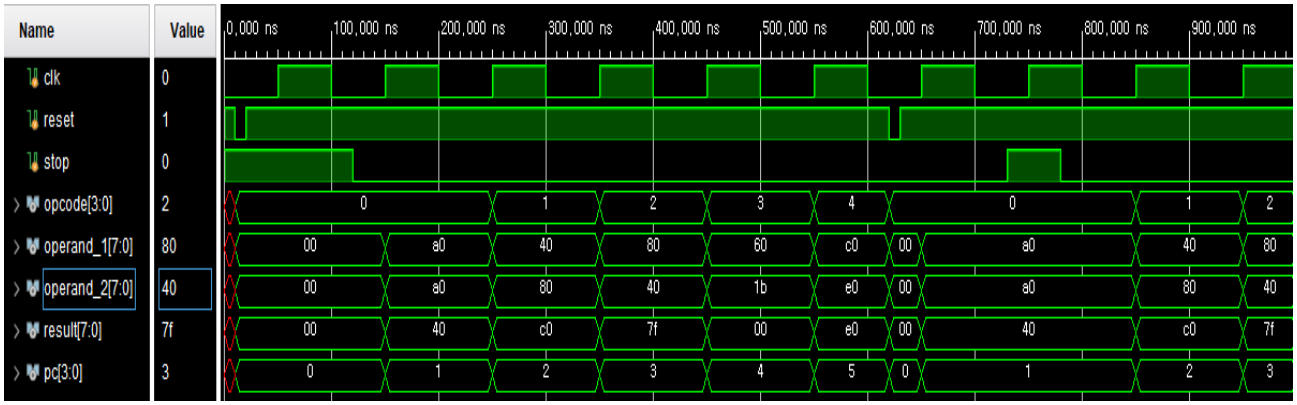
```
    // 읽기 동작 수행
    always @(address) begin
        data_read = regi[address];
    end

    // 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
```

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
```

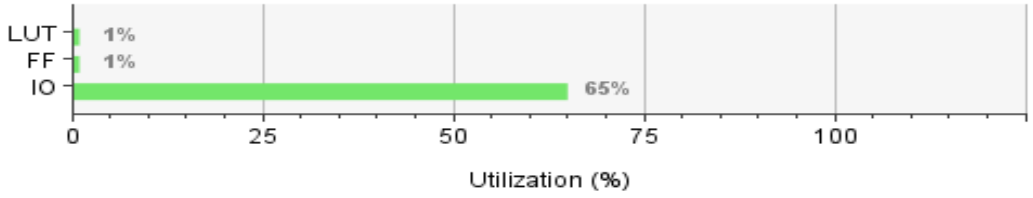
4. 수행 결과 및 시뮬레이션



▲Vivado Simulation

Summary

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

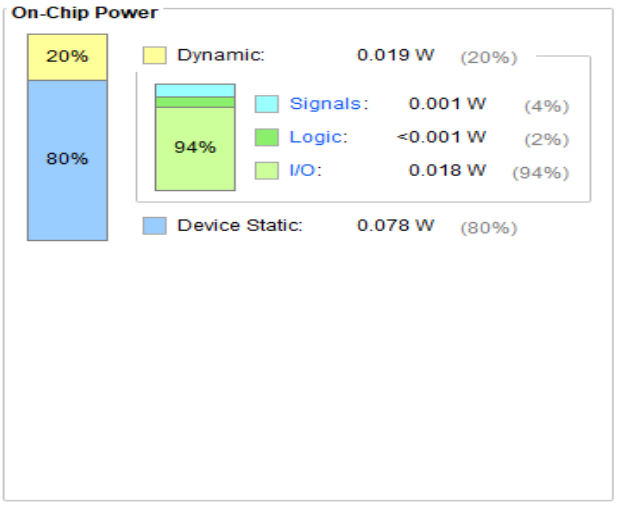


Summary

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

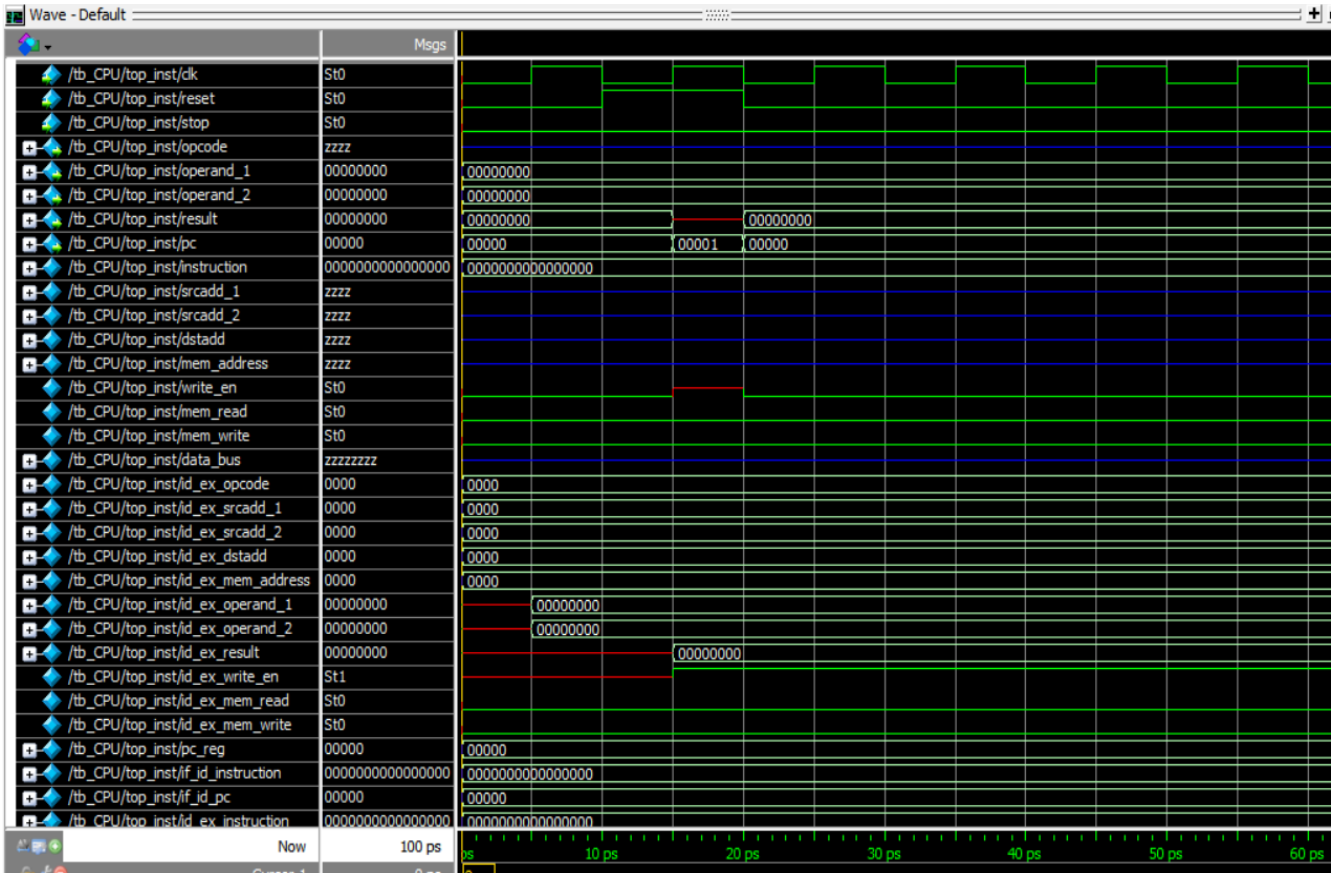


▲Synthesis 결과

5. 시행착오

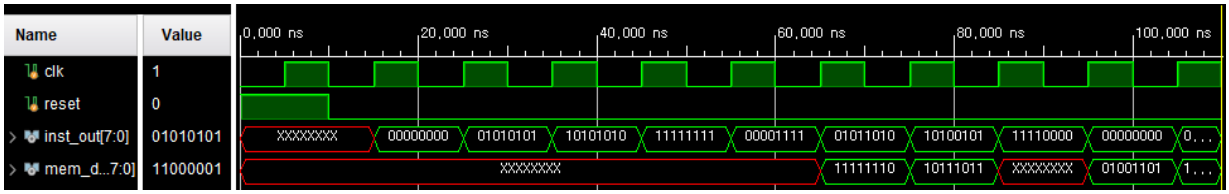
mini cpu를 만들기 위해 cpu구조에 맞게끔 ALU, Accumulator, Control Unit(FSM을 이용해서 fetch-decode-excute, pipelining기능), Instruction Register, Memory, Memory Address Register, Memory Buffer Register, Program Counter, RegisterFile(register들을 instance화하는 모듈), CPU(top모듈) 만들었습니다. 그런데 너무 많은 모듈들이 복잡하게 구성 되어 있고 그로 인해 인스턴스화 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

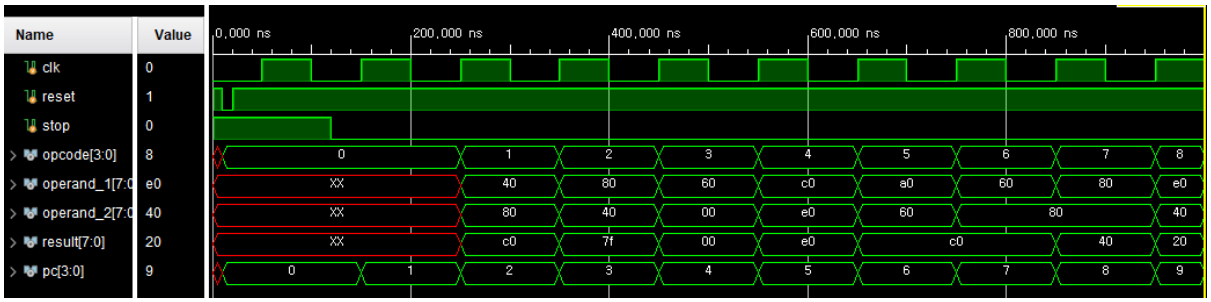


▲Quartus Prime Lite로 모듈 테스트 중 모듈의 테스트벤치가 원하는 대로 작동되지 않는 모습.

개별적인 모듈의 수가 많고, 메모리 및 레지스터 접근 제어 신호가 제대로 정의되어 있지 않아서 발생한 문제.



▲Vivado로 모듈 테스트 중 모듈의 테스트벤치가 원하는 대로 작동되지 않는 모습



▲pc=0 일때 덧셈 연산을 수행하여야 하는데 수행되지 않는 문제

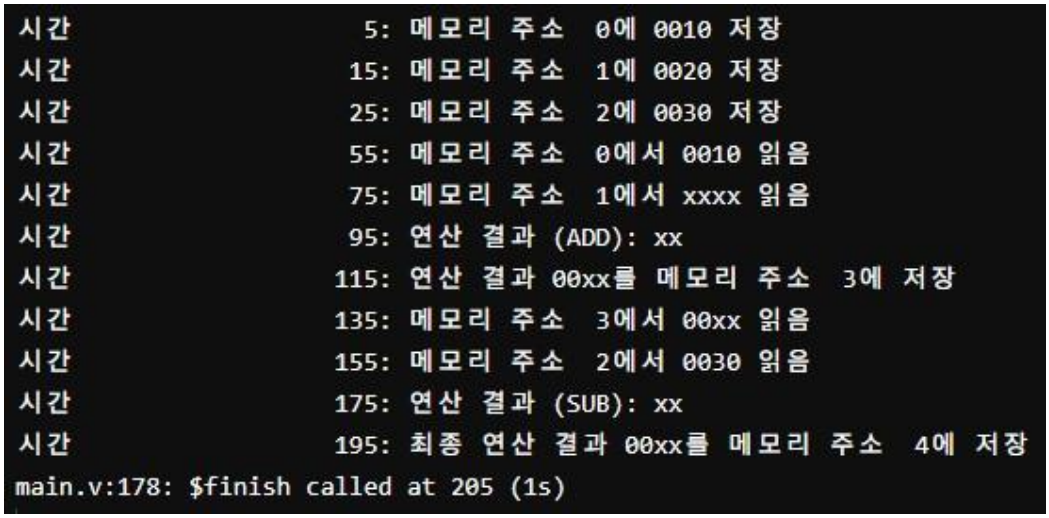
Transistor.v 모듈에서 regi[0]이 정의되지 않아 발생한 문제로, regi[0]의 초기값을 0으로 하여 이를 수정하였습니다.

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 (
```



▲AI에게 테스트벤치 제작을 요청했지만 제대로 작동하지 않는 모습

6. 별첨: 회의록

1차 회의록

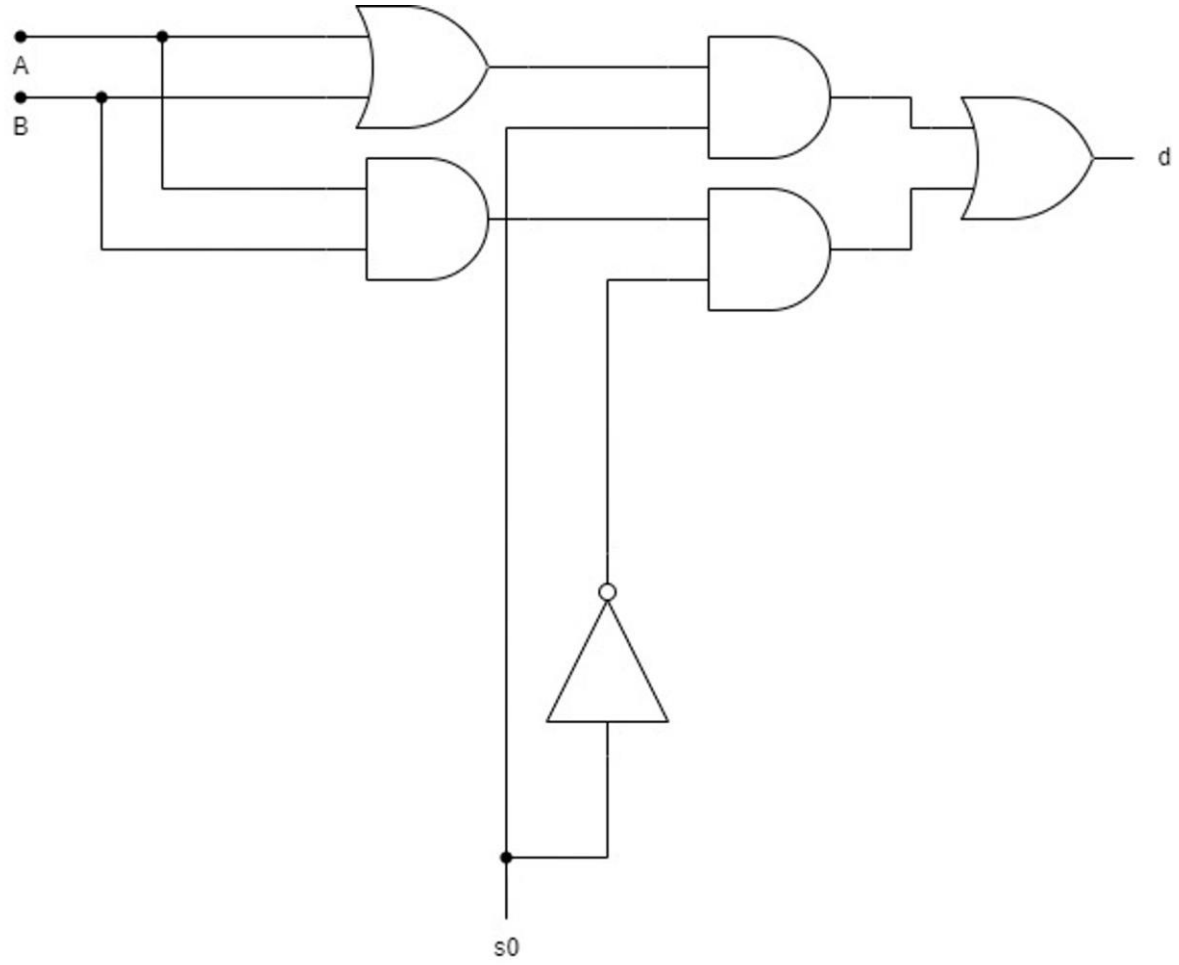
CPU를 만들어야 하므로 CPU의 구조, 구성 요소, 작동 원리에 대한 논의를 하였습니다.

CPU는 ALU(Arithmetic Logic Unit), Control Unit, Register Set로 구성되며 이를 Bus가 연결해주는 것, 그리고 Decoder와 Mux(Multiplexer)가 Bus를 구성함을 확인하였습니다.

Mux를 이용하여 피연산자 A, B 에 대해서 s0에 의해 AND 연산을 할지 OR 연산을 할지 결정되는 회로를 구성해 보았습니다.

다음 회로에서 A와 B는 서로 다른 연산 AND 와 OR 연산을 하게 됩니다. 각각의 연산 결과중 어떤것을 출력할지는 s0에 따라 결정됩니다. s0가 0일때 d=A||B 가 되고, s0가 1일때 d=A&B 가 됩니다.

이처럼 Mux는 둘 이상의 여러 신호 중 하나를 선택하여 내보내는 역할을 하게 됩니다.



2차 회의록

mux 활용에 대한 논의: 덧셈기, 뺄셈기, 곱셈기, 논리 연산기 등의 연산기를 mux를 통해 제어할 수 있도록 설계

회의 시간에 대한 논의: 중간고사 이후 매주 미리 회의 시간을 정하여 원활하게 회의를 할 수 있도록 하고자 함

ALU의 기능과 구성요소에 대한 공부

ALU의 기능:

산술 연산: 덧셈, 뺄셈, 곱셈, 나눗셈 등의 산술 연산을 수행

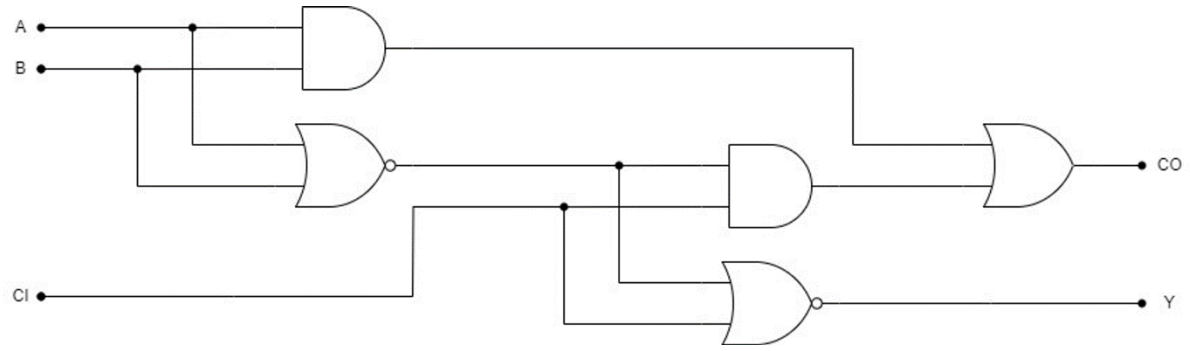
논리 연산: AND, OR, XOR, NOT 등의 논리 연산을 수행

시프트 연산: 비트를 왼쪽 또는 오른쪽으로 이동하는 시프트 연산을 수행

비교 연산: 두 값이 같은지, 큰지, 작은지 등을 비교하는 연산을 수행

조건부 분기: 비교 결과에 따라 프로그램의 실행 흐름을 제어하기 위해 조건부 분기를 수행

▼Adder회로도



3차 회의록

-덧셈기 및 뺄셈기

-4비트 덧셈 뺄셈 연산

-MUX 활용 덧셈 뺄셈 선택적 연산

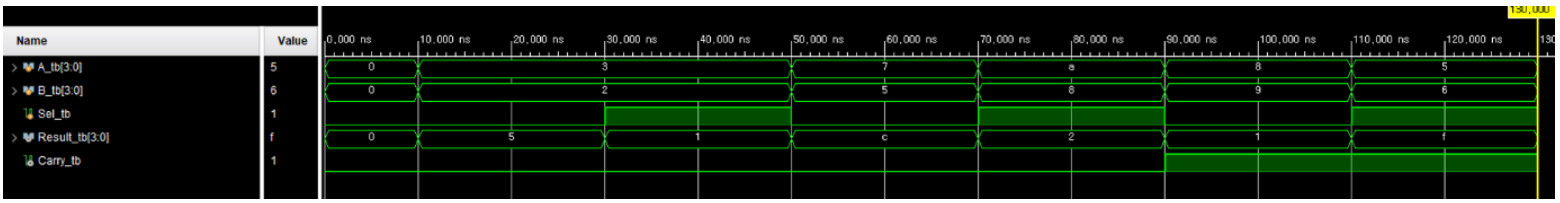
▼Verilog Code

```
module Adder(
    input [3:0] A,
    input [3:0] B,
    output [3:0] Sum,
    output CarryOut
    );

    assign {CarryOut, Sum} = A + B;
endmodule

module Subtractor(
    input [3:0] A,
    input [3:0] B,
    output [3:0] Difference,
    output Borrow
    );

    assign {Borrow, Difference} = A - B;
endmodule

module ArithmeticUnit(
    input [3:0] A,
    input [3:0] B,
    input Sel,  // 0일때 덧셈, 1일때 뺄셈
    output [3:0] Result,
    output CarryOut,
    output Overflow
    );

    wire [3:0] Sum, Difference;
    wire CarryOutAdd, BorrowSub;

    // 모듈 인스턴스
    Adder add(
        .A(A),
        .B(B),
        .Sum(Sum),
        .CarryOut(CarryOutAdd)
    );
    Subtractor sub(
        .A(A),
        .B(B),
        .Difference(Difference),
        .Borrow(BorrowSub)
    );
```

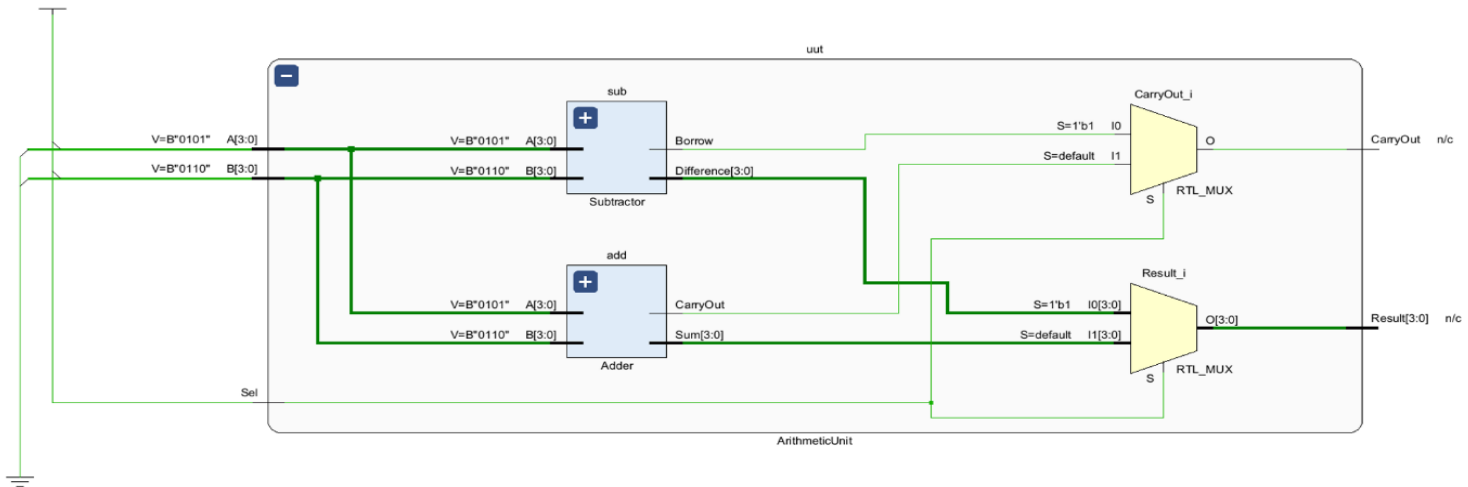
```
    //먹스
    assign Result = Sel ? Difference : Sum;
    assign CarryOut = Sel ? BorrowSub : CarryOutAdd;

endmodule


////////////////////////////////////////////////////////////////////////////////
//////////////////////////////////테스트벤치//////////////////////////////////
////////////////////////////////////////////////////////////////////////////////

`timescale 1ns / 1ps

module ArithmeticUnit_tb;

reg [3:0] A_tb, B_tb;
reg Sel_tb;
wire [3:0] Result_tb;
wire Carry_tb;

ArithmeticUnit uut (
    .A(A_tb),
    .B(B_tb),
    .Sel(Sel_tb),
    .Result(Result_tb),
    .CarryOut(Carry_tb)
);

initial begin
    A_tb = 0; B_tb = 0; Sel_tb = 0;
    #10 $display("Sel=0 for add, Sel=1 for substract");

    // Test Case 1: Addition 3 + 2
    A_tb = 4'd3; B_tb = 4'd2; Sel_tb = 1'b0;  // Sel 0 for addition
    #10 $display("Time: %t, A=%b(%d), B=%b(%d), Sel=%b, Result=%b(%d),
Carry(orBorrow)=%b", $time, A_tb, A_tb, B_tb, B_tb, Sel_tb, Result_tb,
Result_tb, Carry_tb);
    #10;  // Wait 10 ns

    // Test Case 2: Subtraction 3 - 2
    A_tb = 4'd3; B_tb = 4'd2; Sel_tb = 1'b1;  // Sel 1 for subtraction
    #10 $display("Time: %t, A=%b(%d), B=%b(%d), Sel=%b, Result=%b(%d),
Carry(orBorrow)=%b", $time, A_tb, A_tb, B_tb, B_tb, Sel_tb, Result_tb,
Result_tb, Carry_tb);
    #10;  // Wait 10 ns

    // Test Case 3: Addition 7 + 5
    A_tb = 4'd7; B_tb = 4'd5; Sel_tb = 1'b0;
    #10 $display("Time: %t, A=%b(%d), B=%b(%d), Sel=%b, Result=%b(%d),
Carry(orBorrow)=%b", $time, A_tb, A_tb, B_tb, B_tb, Sel_tb, Result_tb,
Result_tb, Carry_tb);
    #10;

    // Test Case 4: Subtraction 10 - 8
    A_tb = 4'd10; B_tb = 4'd8; Sel_tb = 1'b1;
```

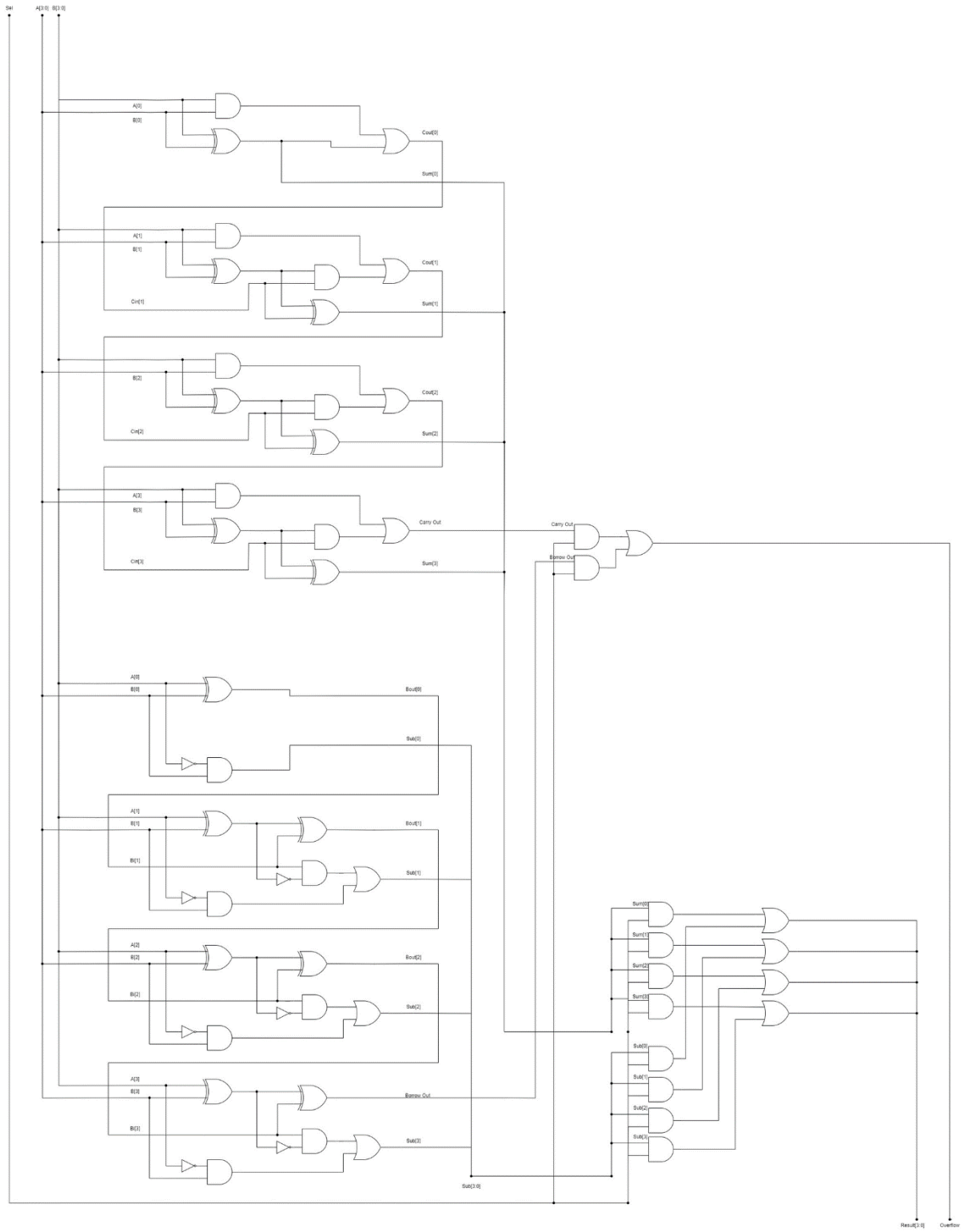
```
    #10 $display("Time: %t, A=%b(%d), B=%b(%d), Sel=%b, Result=%b(%d), Carry(orBorrow)=%b", $time, A_tb, A_tb, B_tb, B_tb, Sel_tb, Result_tb, Result_tb, Carry_tb);
    #10;

    // Test Case 5
    A_tb = 4'd8; B_tb = 4'd9; Sel_tb = 1'b0;
    #10 $display("Time: %t, A=%b(%d), B=%b(%d), Sel=%b, Result=%b(%d), Carry(orBorrow)=%b", $time, A_tb, A_tb, B_tb, B_tb, Sel_tb, Result_tb, Result_tb, Carry_tb);
    #10;

    // Test Case 6
    A_tb = 4'd5; B_tb = 4'd6; Sel_tb = 1'b1;
    #10 $display("Time: %t, A=%b(%d), B=%b(%d), Sel=%b, Result=%b(%d), Carry(orBorrow)=%b", $time, A_tb, A_tb, B_tb, B_tb, Sel_tb, Result_tb, Result_tb, Carry_tb);
    #10;

    $finish;
end

endmodule
```

▼Vivado Simulation



▼Vivado Schemetic




Sel
A[3:0]
B[3:0]
A[0]
B[0]
Cout[0]
Sum[0]
A[1]
B[1]
Cout[1]
Cin[1]
Sum[1]
B[2]
Cout[2]
Cin[2]
Sum[2]
A[3]
B[3]
Carry Out
Cin[3]
Sum[3]
Carry Out
Borrow Out
A[0]
B[0]
Bout[0]
Sub[0]
A[1]
B[1]
Bout[1]
B[1]
Sub[1]
A[2]
B[2]
Bout[2]
B[2]
Sub[2]
A[3]
B[3]
Borrow Out
B[3]
Sub[3]
Sub[3:0]
Sum[0]
Sum[1]
Sum[2]
Sum[3]
Sub[0]
Sub[1]
Sub[2]
Sub[3]
Result[3:0]
Overflow

4차 회의록

instruction에 대한 논의: 4비트의 opcode + 12비트의 operand (피연산자 및 결과값의 레지스터 주소, 각 4비트)로 하여 총 16비트의 instruction 구성 논의 및 메모리 크기에 대한 논의

(예: instruction= 0001 0110 0011 1000 일 때, 6번 레지스터의 값과 3번 레지스터의 값을 opcode 0001에 대응되는 연산대로 연산하여 결과값을 8번 레지스터에 저장)

ALU에서 한번에 연산할 수 있는 값의 크기를 8비트로 지정.

(unsigned 값은 0~255, signed값은 (-128 ~ 127) 까지 값을 한번에 계산 가능.

값의 최상위 비트(가장 왼쪽 비트)를 부호를 위한 비트로 할당.

또한 음수인 값은 최저값 1111 1111 에서 최고값 1000 0000 으로 하여 0일때 절대값이 커지는 것으로 함.

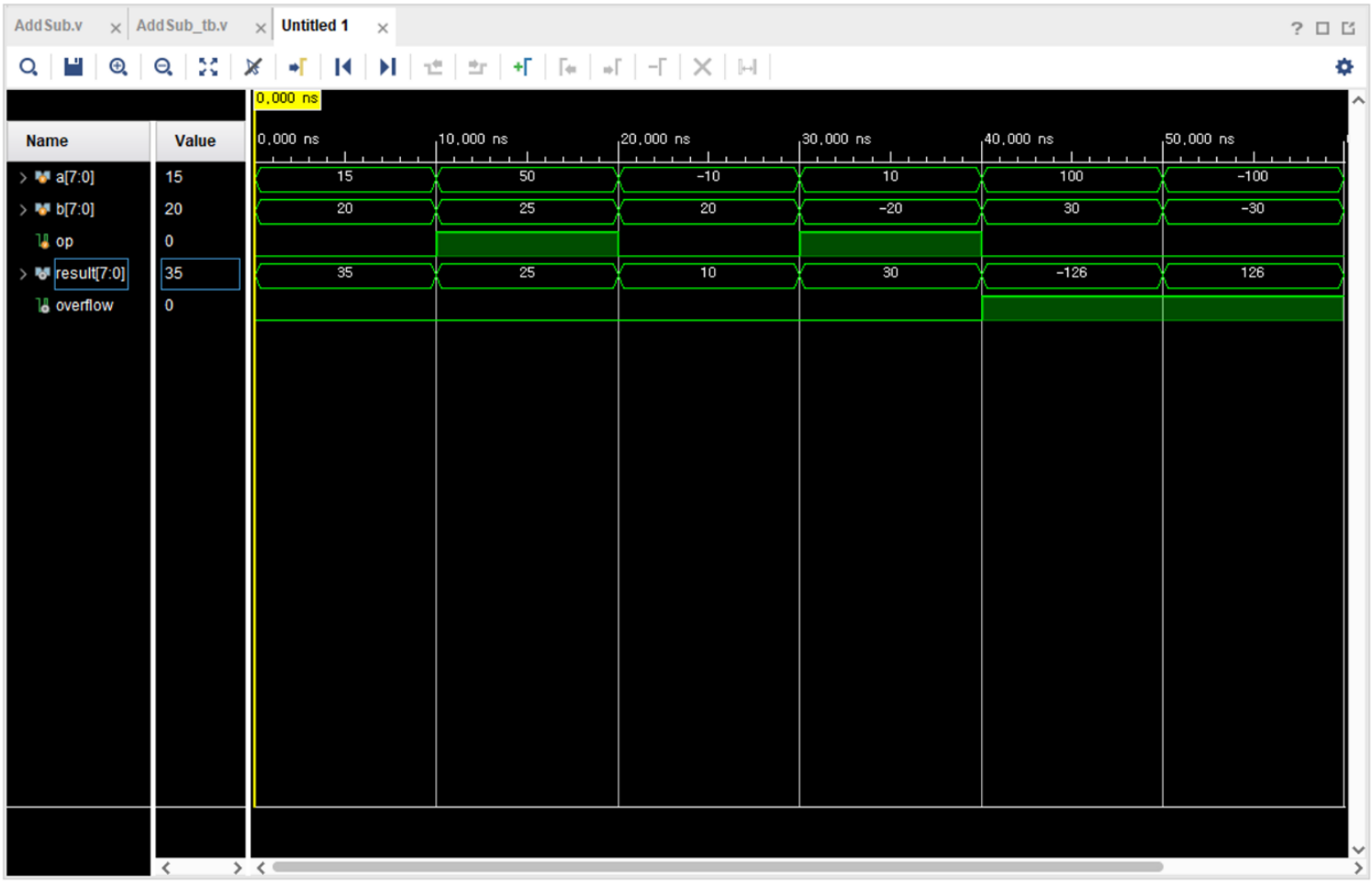
(ex. -1은 1111 1111, 127은 0111 1111, -62는 1100 0010)

▼Verilog Code

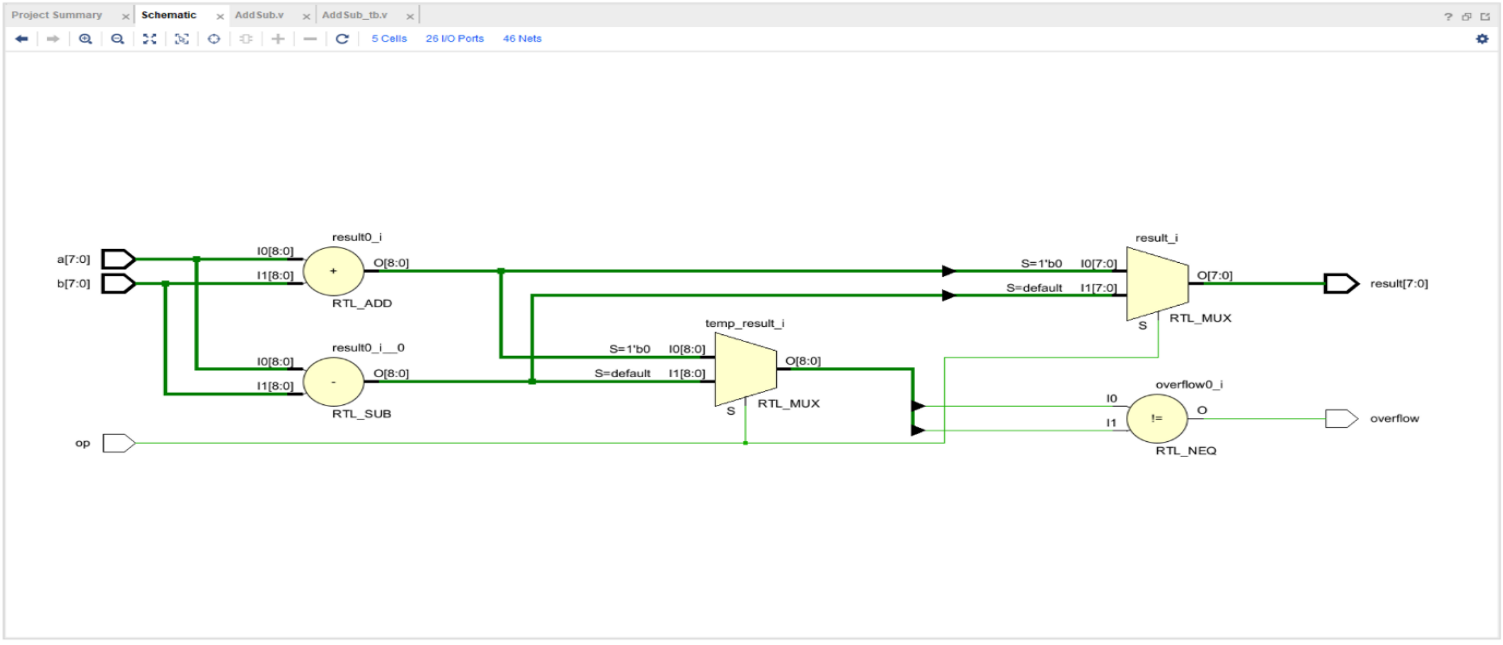
```
module AddSub(
    input signed [7:0] a,
    input signed [7:0] b,
    input op, // 0: addition, 1: subtraction
    output reg signed [7:0] result,
    output reg overflow
);
    // 임시 결과를 저장할 9비트 변수 (오버플로 체크를 위해)
    reg signed [8:0] temp_result;

    always @(a, b, op) begin
        if (op == 0) begin
            // 덧셈
            temp_result = {a[7], a} + {b[7], b};
            result = temp_result[7:0];
        end else begin
            // 뺄셈
            temp_result = {a[7], a} - {b[7], b};
            result = temp_result[7:0];
        end
        // 오버플로 체크
        overflow = (temp_result[8] != temp_result[7]);
    end
endmodule




`timescale 1ns / 1ps

module testbench;
```

```
  reg [7:0] a;
  reg [7:0] b;
  reg op;
  wire [7:0] result;

  AddSub uut (
    .a(a),
    .b(b),
    .op(op),
    .result(result),
    .overflow(overflow)
  );

  initial begin
    $monitor("Time = %d, a = %d, b = %d, op = %b, result = %d, overflow = %b",
$time, a, b, op, result, overflow);

    // 테스트 케이스 1: 덧셈
    a = 8'd15; b = 8'd20; op = 1'b0; // 15 + 20
    #10; // 10ns 대기

    // 테스트 케이스 2: 뺄셈
    a = 8'd50; b = 8'd25; op = 1'b1; // 50 - 25
    #10;

    // 테스트 케이스 3: 음수와의 덧셈
    a = -8'd10; b = 8'd20; op = 1'b0; // -10 + 20
    #10;

    // 테스트 케이스 4: 음수와의 뺄셈
    a = 8'd10; b = -8'd20; op = 1'b1; // 10 - (-20)
    #10;

    a = 8'd100; b = 8'd30; op = 1'b0;
    #10;

    a = -8'd100; b = -8'd30; op = 1'b0;
    #10;

    $finish;
  end
endmodule
```

▼Vivado Simulation



▼Vivado Schemetic



5차 회의록

지난주에 ALU의 구성요소로 사칙연산중 덧셈기와 뺄셈기를 중점적으로 완성하였고, 이번주는 논리연산자를 위주로 베릴로그 코드로 구현해보고자 한다.

우선 각 연산자 (and, or, nand, nor, not, xor, xnor) 에 대하여 opcode를 지정한다.

이때 논리연산자를 다른연산자와 따로 구분하여 가독성등 여러 이점을 누리기 위해 최상위 비트를 1로 하는 7개의 opcode를 완성하였다.

1000 AND

1001 OR

1010 NAND

1011 NOR

1100 NOT A

1101 NOT B

1110 XOR

1111 XNOR

이때, NOT 연산은 피연산자를 하나만 필요로 한다. 따라서 instruction이 두개의 피연산자가 있는 다음과 같은 형태라면 (opcode src1 src2 dst) 피연산자 하나는 무시하게 된다.

또한, CPU 설계에 대한 내용도 논의가 완료되었다.

CPU가 원활히 작동하기 위해서 값을 저장하는 레지스터 뿐만 아니라, 명령을 저장하는 instruction 레지스터 또한 필요할 것으로 예상하고 있다.

마지막으로, 위 논의한 내용을 토대로 논리연산유닛을 베릴로그로 구현하였다.

▼Verilog Code

```
module logic(a,b,opcode,outlu); //outlu : OutLogicUnit
input [15:0] a;
input [15:0] b;
input [2:0] opcode;
output [15:0] outlu;

reg [15:0] outlu;
```

```
always@(a,b,opcode)
begin

case(opcode)
3'b000: outlu = (a & b);
3'b001: outlu = (a | b);
3'b010: outlu = (~(a & b));
3'b011: outlu = (~(a | b));
3'b100: outlu = (~ a);
3'b101: outlu = (~ b);
3'b110: outlu = (a ^ b);
3'b111: outlu = (~(a ^ b));
default outlu = 16'h0000;
endcase

end

endmodule




module tb_logic();
reg [15:0] a;
reg [15:0] b;
reg [2:0] opcode;
wire [15:0] outlu;

// Instantiation of the module

logic l1 (
.a(a),
.b(b),
.opcode(opcode),
.outlu(outlu)
);

// Initialization

initial
begin
        a = 16'h0000;
        b = 16'h0000;
        opcode = 3'b000;
end

// Stimulus must be writte in such a way that we test all the cases for input
conditions.
initial
begin
        // When A>B test all the conditions
        #10
        a = 16'h0009;
        b <= 16'h0005;
        # 5 opcode = 3'b000;
```

```
        # 5 opcode = 3'b001;
        # 5 opcode = 3'b010;
        # 5 opcode = 3'b011;
        # 5 opcode = 3'b100;
        # 5 opcode = 3'b101;
        # 5 opcode = 3'b110;
        # 5 opcode = 3'b111;

        // When A<B test all the cases
        #10
        a = 16'h0003;
        b = 16'h000F;
        # 5 opcode = 3'b000;
        # 5 opcode = 3'b001;
        # 5 opcode = 3'b010;
        # 5 opcode = 3'b011;
        # 5 opcode = 3'b100;
        # 5 opcode = 3'b101;
        # 5 opcode = 3'b110;
        # 5 opcode = 3'b111;

        // When A=B test all the conditions
        #10
        a = 16'h00E9;
        b = 16'h00E9;
        # 5 opcode = 3'b000;
        # 5 opcode = 3'b001;
        # 5 opcode = 3'b010;
        # 5 opcode = 3'b011;
        # 5 opcode = 3'b100;
        # 5 opcode = 3'b101;
        # 5 opcode = 3'b110;
        # 5 opcode = 3'b111;

end
endmodule
```

6차 회의록

이번주는 CPU가 값을 읽고 쓸 수 있는 메모리를 구현하고, 이를 ALU 연산에 활용해보고자 하였다.

이를 위해, 16비트크기의 값을 16개 저장할 수 있는 간단한 메모리를 베릴로그 코드로 구현하였다.

```
module data_memory(
    input clk,                // 클럭 신호
    input [3:0] addr,         // 4 비트 주소 입력
    input [15:0] data_in,     // 16 비트 데이터 입력
    input write_en,           // 쓰기 신호
    output reg [15:0] data_out // 16 비트 데이터 출력
);

    reg [15:0] memory_array [15:0];   // 16 개의 16 비트 메모리 공간 선언

    // 메모리 쓰기 동작
    always @(posedge clk) begin
        if (write_en)
            memory_array[addr] <= data_in;
    end

    // 메모리 읽기 동작
    always @(posedge clk) begin
        data_out <= memory_array[addr];
    end

endmodule
```

위 코드는 write_en 이 1일때 지정된 주소 adder에 해당하는 메모리에 data_in의 값을 쓰고, 이를 읽어온다. write_en이 0일때는 메모리에 값을 쓰는 동작을 하지 않는다.

그리고 이전에 연산을 해주는 ALU또한 베릴로그 코드로 구현한바 있다. 이번엔 이를 발전시켜, opcode를 처리할때 decoder를 활용하는 ALU를 구현하였다.

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

  // 3-to-8 decoder
  reg [7:0] decoder_out;
  always @(*) begin
    case (opcode)
      3'b000: decoder_out = 8'b00000001; // add
      3'b001: decoder_out = 8'b00000010; // sub
      3'b010: decoder_out = 8'b00000100; // not
      3'b011: decoder_out = 8'b00001000; // and
      3'b100: decoder_out = 8'b00010000; // or
      3'b101: decoder_out = 8'b00100000; // xor
```

```
      3'b110: decoder_out = 8'b01000000; // shift left
      3'b111: decoder_out = 8'b10000000; // shift right
      default: decoder_out = 8'b00000000;
    endcase
  end

  // ALU operation selection and execution
  always @(*) begin
    case (1'b1)
      decoder_out[0]: result = operand1 + operand2;    // add
      decoder_out[1]: result = operand1 - operand2;    // sub
      decoder_out[2]: result = ~operand1;            // not
      decoder_out[3]: result = operand1 & operand2;    // and
      decoder_out[4]: result = operand1 | operand2;    // or
      decoder_out[5]: result = operand1 ^ operand2;    // xor
      decoder_out[6]: result = operand1 << 1;        // shift left
      decoder_out[7]: result = operand1 >> 1;        // shift right
      default: result = 8'b00000000;
    endcase
  end

endmodule
```

이 둘을 활용하여, 다중 연산을 처리할 수 있는 형태로 응용해보았다.

예를들어, A+B=C 일때, C-D 의 값을 구하려면, C값을 어딘가 저장하는 동작을 반드시 해야한다. 아래 코드는 그러한 동작을 수행할 수 있게 한다.

두 모듈을 인스턴스해서 테스트하는 테스트벤치를 만들었다.

```
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 (
    .opcode(opcode),
    .operand1(operand1),
    .operand2(operand2),
    .result(alu_result)
  );

  // data_memory instance
  data_memory my_memory (
    .clk(clk),
```

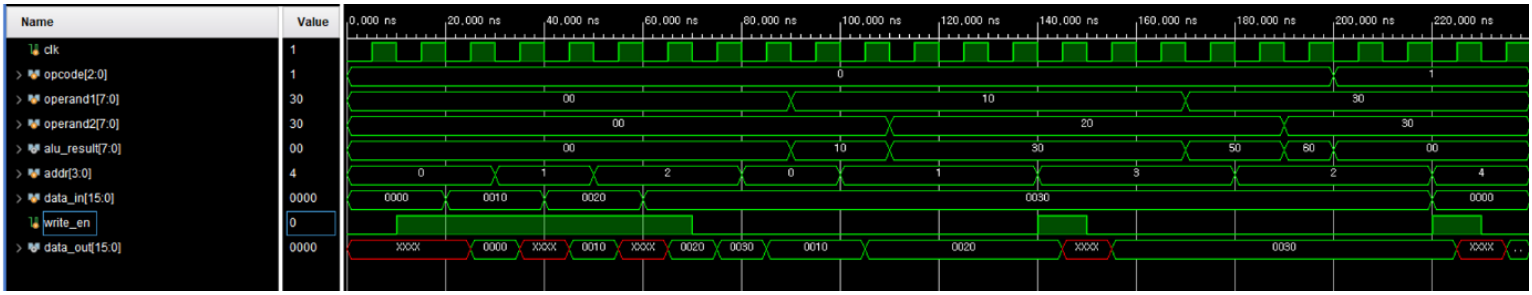
```
    .addr(addr),
    .data_in(data_in),
    .write_en(write_en),
    .data_out(data_out)
);

// Clock generation
initial begin
    clk = 0;
    forever #5 clk = ~clk; // 10ns clock period
end

initial begin
    // Initialization
    write_en = 0;
    addr = 0;
    data_in = 0;
    opcode = 0;
    operand1 = 0;
    operand2 = 0;

    // 1. Set initial memory values
    #10;
    write_en = 1;
    addr = 4'b0000;
    #10;
    data_in = 16'h0010; // Store 16'h0010 at memory address 0
    $display("시간 %t: 메모리 주소 %h에 %h 저장", $time, addr, data_in);

    #10;
    addr = 4'b0001;
    #10;
    data_in = 16'h0020; // Store 16'h0020 at memory address 1
    $display("시간 %t: 메모리 주소 %h에 %h 저장", $time, addr, data_in);

    #10;
    addr = 4'b0010;
    #10;
    data_in = 16'h0030; // Store 16'h0030 at memory address 2
    $display("시간 %t: 메모리 주소 %h에 %h 저장", $time, addr, data_in);
    #10;
    write_en = 0;

    // 2. Read and operate
    #10;
    addr = 4'b0000;
    #10;
    operand1 = data_out[7:0];
    $display("시간 %t: 메모리 주소 %h에서 %h 읽음", $time, addr, data_out);

    #10;
    addr = 4'b0001;
    #10;
    operand2 = data_out[7:0];
    $display("시간 %t: 메모리 주소 %h에서 %h 읽음", $time, addr, data_out);
```

```
    // 3. Perform operation (e.g., ADD)
    #10;
    opcode = 3'b000; // ADD operation
    #10;
    $display("시간 %t: 연산 결과 (ADD): %h", $time, alu_result);

    // 4. Store the operation result in memory
    #10;
    write_en = 1;
    addr = 4'b0011; data_in = {8'b0, alu_result}; // Store the operation result
at memory address 3
    #10;
    $display("시간 %t: 연산 결과 %h 를 메모리 주소 %d 에 저장", $time, data_in,
addr);
    write_en = 0;

    // 5. Read the stored value for another operation
    #10;
    addr = 4'b0011;
    #10;
    operand1 = data_out[7:0];
    $display("시간 %t: 메모리 주소 %d 에서 %h 읽음", $time, addr, data_out);

    #10;
    addr = 4'b0010;
    #10;
    operand2 = data_out[7:0];
    $display("시간 %t: 메모리 주소 %d 에서 %h 읽음", $time, addr, data_out);

    // 6. Perform another operation (e.g., SUB)
    #10;
    opcode = 3'b001; // SUB operation
    #10;
    $display("시간 %t: 연산 결과 (SUB): %h", $time, alu_result);

    // 7. Store the final operation result in memory
    #10;
    write_en = 1;
    addr = 4'b0100; data_in = {8'b0, alu_result}; // Store the final operation
result at memory address 4
    #10;
    write_en = 0;
    $display("시간 %t: 최종 연산 결과 %h 를 메모리 주소 %d 에 저장", $time,
data_in, addr);

    // End of test
    #10;
    $finish;
  end
endmodule
```

▼실행결과

```
시간                   20: 메모리 주소 0 에 0010 저장
시간                   40: 메모리 주소 1 에 0020 저장
시간                   60: 메모리 주소 2 에 0030 저장
시간                   90: 메모리 주소 0 에서 0010 읽음
시간                  110: 메모리 주소 1 에서 0020 읽음
시간                  130: 연산 결과 (ADD): 30
시간                  150: 연산 결과 0030 를 메모리 주소  3 에 저장
시간                  170: 메모리 주소  3 에서 0030 읽음
시간                  190: 메모리 주소  2 에서 0030 읽음
시간                  210: 연산 결과 (SUB): 00
시간                  230: 최종 연산 결과 0000 를 메모리 주소  4 에 저장
main.v:195: $finish called at 240 (1s)
```

▼Vivado Simulation



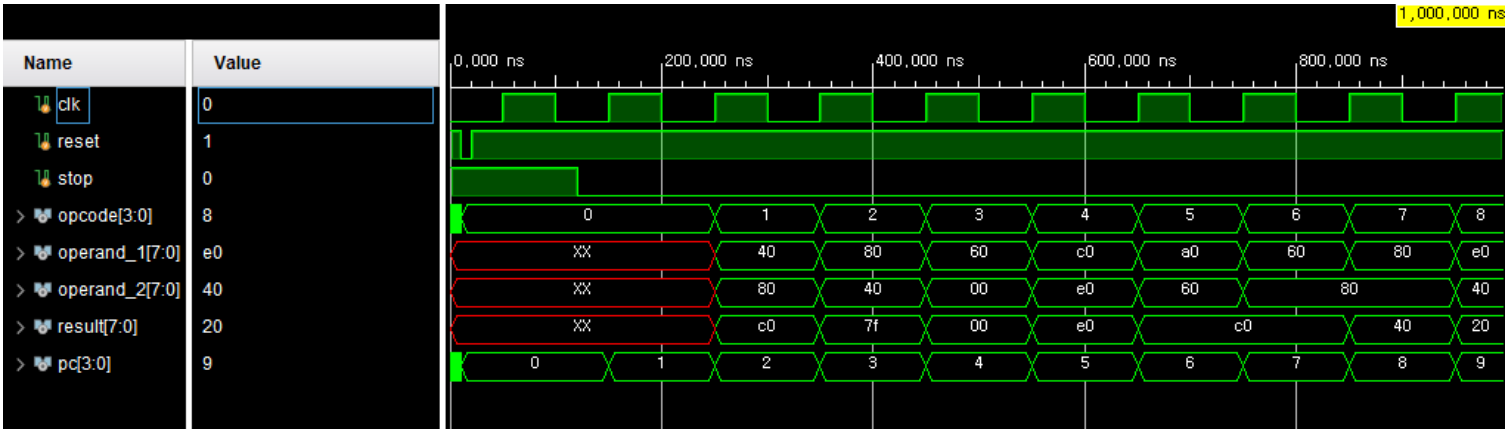
7차 회의록

디지털 논리회로 과제에서 나온 FDE 설계를 팀원들끼리 비교해보고 동작방식, 최적화 방안, 보완할 점 등에 대해 논의 하였습니다.

신윤규 팀원은 mini cpu만들기 위해 cpu구조에 맞게끔 ALU, Accumulator, ControlUnit(FSM을 이용해서 fetch-decode-excute pipelining 기능), InstructionRegister, Memory, MemoryAddressRegister, MemoryBufferRegister, ProgramCounter, RegisterFile(register들을 instance화 하는 모듈), CPU(top 모듈) 만들었습니다. 그런데 너무 많은 모듈들이 복잡하게 구성되어있고 그로 인해 인스턴스화 오류 및 동작 오류가 발생하는 듯 합니다. 이를 개선하기 위해서 모듈들을 간소화하고 동작 확인 후 기능 추가 및 보완하고자 합니다.

https://github.com/potatogyu99/cpu_2.git

강석민 팀원이 설계한 FDE가 향후 CPU설계에 좋은 참고자료가 될 것이라는 의견이 모아졌습니다.



https://github.com/lota09/Digital-Logic-Circuit/tree/main/week9/FDE

▼Vivado Simulation

▼설계된 FDE의 동작 원리설명

1. 테스트 벤치는 Top이라는 모듈을 인스턴스화 하고, 모듈 Top은 각 하위 모듈을 서로 연결시키는 역할을 합니다.

2. Top에서 처음으로 인스턴스화 하는 모듈은 fetch 모듈입니다. 이 모듈에서 clk과 reset 신호에

의해 프로그램 카운터 pc 가 제어됩니다.

3. fetch에서 처리한 pc는 명령어메모리가 저장되어있는 InstMem 모듈의 입력이 됩니다. 이 모듈에서 pc의 값에 따라 미리 초기화 되어있는 각 명령어를 불러오고, 이를 출력으로 내보냅니다.

4. InstMem에서 내보낸 instruction은 decode 모듈에 의해 해석됩니다. 16비트로 구성된 instrution은 최상위 비트부터 4비트씩 4조각으로 나누어지며, 각각 opcode, 피연산자레지스터주소1 ,피연산자레지스터주소2, 결과레지스터주소 로 재구성하여 출력합니다.

5. decode모듈에서 내보낸 opcode ,srcadd1 ,scradd2 ,dstadd를 execute모듈에서 처리합니다. 이때 register모듈을 통해 입력된 주소를 입력하여 피연산자를 불러오고, opcode의 값에따라 alu에서 불러온 피연산자 값을 다르게 처리합니다. 이때 alu에서 최상위 비트를 제외한 나머지 3비트만 처리합니다. 예를들어 000은 덧셈연산을, 001을 뺄셈연산을 처리합니다.

6. execute모듈에서 연산이 진행된 결과값은 register모듈을 통해 지정된 레지스터 주소 dstadd에 저장됩니다. 그리고 시뮬레이션 결과를 보기위해, top모듈에서 피연산자 값 operand1 및 operand2 그리고 연산결과 result를 execute모듈 출력에서 가져와서 top모듈이 출력합니다.

보완할 점:

1. 위 FDE 시스템에서 초기화된 명령어를 수정할 수는 없습니다. 하지만 실제 CPU는 지정된 명령만 수행하는 것이 아닌, CPU에 입력된 명령을 수행하여야 합니다.

2. 실제 CPU에서는 alu 모듈이 구체적인 동작을 할 수 있도록 각 비트마다 구체적인 경로가 있습니다. 하지만 설계된 FDE 시스템은 베릴로그 연산자를 통해 연산하였으므로, 이는 하드웨어적 설계보다는 소프트웨어적 설계에 가깝습니다. - 이를 보완하기 위해 이전 회의를 통해 완성한 가산기 감산기 등을 사용할 것입니다. 그 외의 모듈들은 향후 설계할 계획입니다.

3. CPU는 alu 처리외에도 다양한 처리를 합니다. 하지만 설계된 fde시스템은 opcode 최상위 비트가 0일때 동작만 설계되어있습니다. opcode 최상위 비트가 1일때 기존과 다른 동작을 할 수 있게 만들 계획입니다.