Lab1 Report

이름: 권도현

학번: 2023065350

학과: 컴퓨터소프트웨어학부

Project Design

	High level simulation	Verilog test
Programming language	C++	Verilog
Environment	Ubuntu	Vivado
Objective	C++ ALU 구현 확인 / .mem 파일 생성	Verilog ALU 구현 확인
Testing	.ref 파일 이용	.mem 파일 이용

Overall structure

- 우리가 직접적으로 사용해야 하는 것은 ***alu_result: 연산 결과 저장, operand1: 피연산자1, operand2: 피연산자2, shamt: shift 연산** 양이다.

1. CPP

우선 operand1, operand2는 unsigned이다.

ADDU, AND, NOR, OR, SUB, XOR 연산들은 C++에서 제공하는 연산자로 각 연산의 과정을 동일하게 구현할 수 있었다.

SLL, SRL 은 shamt만큼 >> , << 를 이용해 구현하였다.

SLTU, EQ, NEQ는 (비교식) ? A:B 구조를 이용하여 (비교식)이 맞으면 1, 틀리면 0의 값을 갖도록 구현하였다.

SRA, SLT 는 signed로 type casting을 하고 SRL, SLTU 와 동일하게 구현하였다.

LUI는 operand2를 16 bit 만큼 Left shift 하였다.

```cpp
switch (static_cast<ALUOp>(aluop))
{
case ALU_ADDU:
        *alu_result = operand1 + operand2;
        break;
case ALU_AND:
        *alu_result = operand1 & operand2;
        break;
case ALU_NOR:
        *alu_result = ~(operand1 | operand2);
        break;
case ALU_OR:
        *alu_result = operand1 | operand2;
        break;
case ALU_SLL:
        *alu_result = operand2 << shamt;
        break;
case ALU_SRA:
        *alu_result = (int32_t)operand2 >> shamt;
        break;
case ALU_SRL:
        *alu_result = operand2 >> shamt;
        break;
case ALU_SUBU:
        *alu_result = operand1 - operand2;
        break;
case ALU_XOR:
        *alu_result = operand1 ^ operand2;
        break;
case ALU_SLT:
        *alu_result = (int32_t)operand1 < (int32_t)operand2 ? 1 : 0;
        break;
case ALU_SLTU:
        *alu_result = operand1 < operand2 ? 1 : 0;
        break;
case ALU_EQ:
        *alu_result = operand1 == operand2 ? 1 : 0;
        break;
case ALU_NEQ:
        *alu_result = operand1 != operand2 ? 1 : 0;
        break;
case ALU_LUI:
        *alu_result = operand2 << 16;
        break;
default:
        break;
}
```

2. Verilog

대부분의 ALU 연산을 C++에서 구현한 것과 비슷하게 구현했다.

달라진 점

1. Sequential Logic에서 동시에 업데이트 될 수 있도록 "=" 대신 "<="을 사용하였다.
2. Unsigned를 signed로 변환할 때, $signed() 를 사용하였다.
3. ALU_SRA에서 >>> 를 사용하였다.

```
always @(*) begin
    alu_result = 0;
    case (funct)
        `ALU_ADDU: alu_result <= operand1 + operand2;
        `ALU_AND : alu_result <= operand1 & operand2;
        `ALU_NOR : alu_result <= ~(operand1 | operand2);
        `ALU_OR  : alu_result <= operand1 | operand2;
        `ALU_SLL : alu_result <= operand2 << shamt;
        `ALU_SRA : alu_result <= $signed(operand2) >>> shamt;
        `ALU_SRL : alu_result <= operand2 >> shamt;
        `ALU_SUBU: alu_result <= operand1 - operand2;
        `ALU_XOR : alu_result <= operand1 ^ operand2;
        `ALU_SLT : alu_result <= ($signed(operand1) < $signed(operand2)) ? 1 : 0;
        `ALU_SLTU: alu_result <= (operand1 < operand2) ? 1 : 0;
        `ALU_EQ  : alu_result <= (operand1 == operand2) ? 1 : 0;
        `ALU_NEQ : alu_result <= (operand1 != operand2) ? 1 : 0;
        `ALU_LUI : alu_result <= operand2 << 16;
    endcase
```

Results

1. Cpp

```
kwondh3236@NOTKDH:~/computer_architecture/HW01/cpp$ ./alu
PASSED: 100, FAILED: 0
```

2. Verilog

> PASSED[31:0]	990
> FAILED[31:0]	0

Difficulties

1. *코드를 짜고 Verilog의 첫번째 실행에서 테스트 Te5에 대해서만 "FAILED"가 발생했다.*

```
TEST CTRL Sig: 5 :
FAILED
funct:  5, operand1 = 1895220943, operand2 = 2734752558, shamt = 28, result = 0000000a (Ans : fffffffa)
```

a. GLOBAL.V 에서 ALUop 5 = ALU_SRA 인 것을 확인

```
6  `define     ALU_SRA             4'b0101
```

b. 기존 코드 확인

```
`ALU_SRA : alu_result = $signed(operand2) >> shamt;
```

문제점: c++에서와 같이 shift 연산이 unsinged에 대해서는 logical right shift 연산을 수행하

고, signed에 대해서는 arithmetic right shift 연산 수행한다고 생각했다.

해결: 인터넷 검색과 교수님께서 올려주신 2번째 PPT (Lecture 2)를 확인하고 Verilog는 Logical shift의 연산자와 Arithmetic shift의 연산자가 구분되어져 있다는 것을 확인했다.

c. 수정한 코드 `ALU_SRA : alu_result = $signed(operand2) >>> shamt;`
   - Arithmetic shift in Verilog: <<< , >>>

2. *C++로 High-level simulation 하고, Verilog로 low-level test 하는 구조를 정확히 이해하지 못했다.*

a. C++에서 .ref 코드를 통해 C++ ALU 연산 구현이 정확한지 검사한다.

```
ifstream fin_op1_ref;
ifstream fin_op2_ref;
ifstream fin_shamt_ref;
ifstream fin_funct_ref;
ifstream fin_result_ref;
fin_op1_ref.open("operand1.ref");
fin_op2_ref.open("operand2.ref");
fin_shamt_ref.open("shamt.ref");
fin_funct_ref.open("funct.ref");
fin_result_ref.open("alu_result.ref");
```

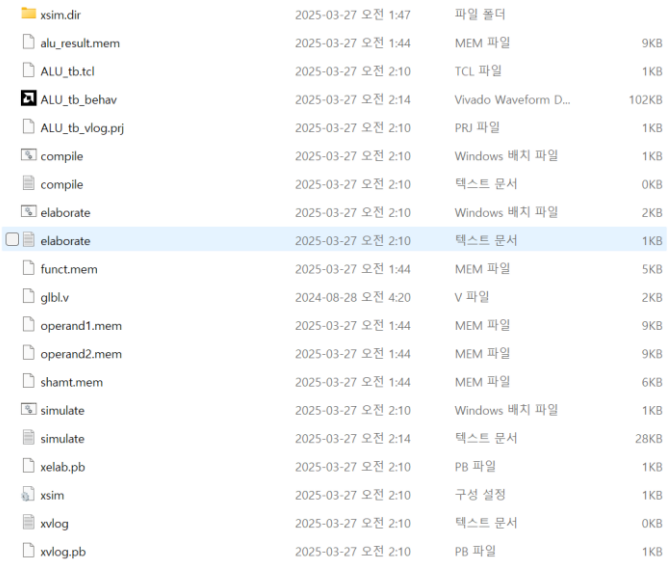
main.cpp

b. C++ 코드에 대해 시뮬레이션을 수행함과 동시에, Verilog testbench용 .mem 파일을 생성한다.

```
// Make a custom testbench using our cpp simulator!
ofstream fout_op1;
ofstream fout_op2;
ofstream fout_shamt;
ofstream fout_funct;
ofstream fout_result;
fout_op1.open("operand1.mem");
fout_op2.open("operand2.mem");
fout_shamt.open("shamt.mem");
fout_funct.open("funct.mem");
fout_result.open("alu_result.mem");
```

main.cpp

c. Vivado simulation에서 testbench가 .mem 파일들을 인식할 수 있도록, 해당 파일들을 Vivado 프로젝트 폴더 안으로 복사한다.



xsim.dir	2025-03-27 오전 1:47	파일 폴더	
alu_result.mem	2025-03-27 오전 1:44	MEM 파일	9KB
ALU_tb.tcl	2025-03-27 오전 2:10	TCL 파일	1KB
ALU_tb_behav	2025-03-27 오전 2:14	Vivado Waveform D...	102KB
ALU_tb_vlog.prj	2025-03-27 오전 2:10	PRJ 파일	1KB
compile	2025-03-27 오전 2:10	Windows 배치 파일	1KB
compile	2025-03-27 오전 2:10	텍스트 문서	0KB
elaborate	2025-03-27 오전 2:10	Windows 배치 파일	2KB
elaborate	2025-03-27 오전 2:10	텍스트 문서	1KB
funct.mem	2025-03-27 오전 1:44	MEM 파일	5KB
glbl.v	2024-08-28 오전 4:20	V 파일	2KB
operand1.mem	2025-03-27 오전 1:44	MEM 파일	9KB
operand2.mem	2025-03-27 오전 1:44	MEM 파일	9KB
shamt.mem	2025-03-27 오전 1:44	MEM 파일	6KB
simulate	2025-03-27 오전 2:10	Windows 배치 파일	1KB
simulate	2025-03-27 오전 2:14	텍스트 문서	28KB
xelab.pb	2025-03-27 오전 2:10	PB 파일	1KB
xsim	2025-03-27 오전 2:10	구성 설정	1KB
xvlog	2025-03-27 오전 2:10	텍스트 문서	0KB
xvlog.pb	2025-03-27 오전 2:10	PB 파일	1KB

₩sim_1₩behav₩xsim

d. Verilog의 ALU_tb.v testbench에서 $readmemh, $readmemb로 로딩되어 Verilog ALU 설계의 정확성을 테스트하는 데 사용된다.