20180188 서승연 20180373 김현지

# Lab1. ALU

## Introduction

이번 과제의 목표는 ModelSim과 Verilog 사용법을 익히고 ALU를 디자인 및 구현하는 것이다. 모델심(Model Simulation)은 전자회로, 시스템에 사용되는 HW Description Language인 Verilog를 이용하여 gate를 모델링하여 결과를 얻는 IDE이다.

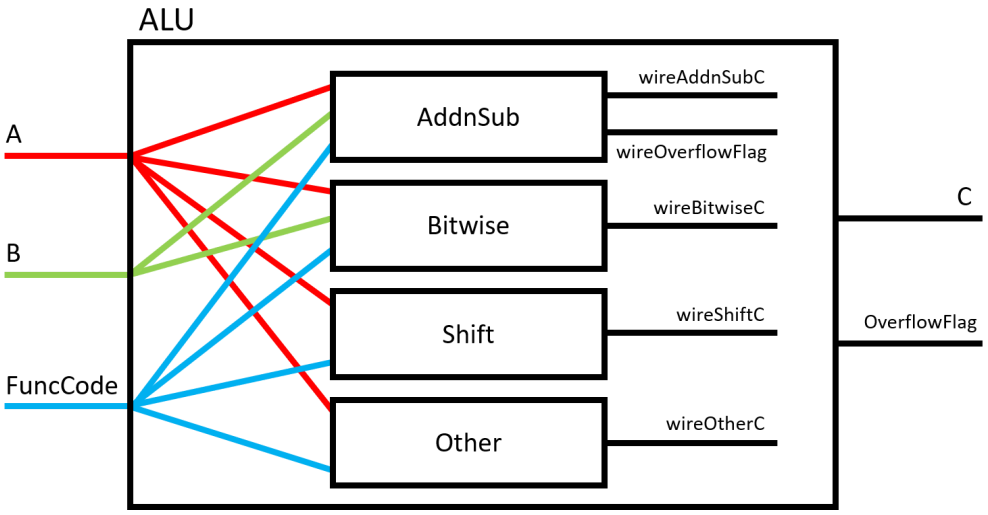
ALU란 Arithmetic Logic Unit의 준말로 두 숫자를 (A, B) input으로 받아 덧셈, 뺄셈 등의 arithmetic operation과 논리합, 논리곱 등의 logical operation을 계산하는 디지털 회로이다. 이번 과제에서는 2개의 숫자와 연산 정보에 대한 정보를 (FuncCode) input으로 받아 계산을 수행하는 ALU를 만드는 것을 목표로 한다. 아래 표는 FuncCode별 기능을 나타낸 표이다.

| FuncCode | Operation | Comment |
|---|---|---|
| 0000 | A + B | Signed Addition |
| 0001 | A – B | Signed Subtraction |
| 0010 | A | Identity |
| 0011 | ~A | Bitwise NOT |
| 0100 | A & B | Bitwise AND |
| 0101 | A \| B | Bitwise OR |
| 0110 | ~(A & B) | Bitwise NAND |
| 0111 | ~(A \| B) | Bitwise NOR |

| FuncCode | Operation | Comment |
|---|---|---|
| 1000 | A ⊕ B | Bitwise XOR |
| 1001 | ~(A ⊕ B) | Bitwise XNOR |
| 1010 | A << 1 | Logical Left Shift |
| 1011 | A >> 1 | Logical Right Shift |
| 1100 | A <<< 1 | Arithmetic Left Shift |
| 1101 | A >>> 1 | Arithmetic Right Shift |
| 1110 | ~A + 1 | Two's Complement |
| 1111 | 0 | Zero |

## Design

1. Top-Level Module(ALU) Division



ALU의 기능에 따라 아래와 같이 4가지 세부 Module로 디자인하였다.

AddnSub : ADD(0000), SUB(0001)
Bitwise : NOT(0011), AND(0100), OR(0101), NAND(0110), NOR(0111), XOR(1000), XNOR(1001)
Shift : LLS(1010), LRS(1011), ALS(1100), ARS(1101)
Other : ID(0010), TCP(1110), ZERO(1111)

(괄호의 4-bit binary는 FuncCode를 의미한다)

## 2. Submodules Operation

### a. AddnSub

Signed Addition & Subtraction 기능을 수행하는 Submodule이다. (0 ≤ FuncCode < 1)
먼저 A, B에 대한 연산을 각 Output C에 assign한다. 이때, OverflowFlag에 주의해야한다. Addition의 경우, Input A와 B의 MSB가 같지만 Output C의 MSB가 다를 경우 OverflowFlag가 1이 된다. (+와 +의 합이 - 인 경우 / -와 -의 합이 +인 경우) Subtraction의 경우, A와 B의 MSB가 다르지만, C의 MSB가 A와 다를 경우 OverflowFlab가 1이 된다. (+에서 -의 차가 - 인 경우/ -에서 +의 차가 +인 경우) 그 외에는 OverflowFlag가 0이 된다.

### b. Bitwise

Bitwise와 관련된 기능을 수행하는 Submodule이다. (3 ≤ FuncCode < 10)
연산에 있어 복잡한 구현없이, 해당하는 A, B의 bitwise 연산을 기호로 표현하여 Output C에 assign 하면된다.

### c. Shift

Shift와 관련된 기능을 수행하는 Submodule이다. (10 ≤ FuncCode < 14)
대부분의 연산에 있어 복잡한 구현없이, 해당하는 A의 Shift 연산을 기호로 표현하여 Output C에 assign 하면된다. 다만, Arithmetic Right Shift의 경우 A가 기본적으로 Unsigned이기 때문에, A의 MSB에 따라 Shift된 C의 MSB값을 변경하도록 한다.

### d. Other

위에 기술된 기능을 제외한 나머지를 수행하는 Submodule이다. (3 ≤ FuncCode < 10)
연산에 있어 복잡한 구현없이, Output C에 해당하는 연산을 기호로 표현하여 assign 하면된다.

## 3. How to Interconnect

### a. Input

각 Submodule의 Input에 ALU(Top-level module)의 Input을 똑같이 연결하였다. Shift와 Other의 연산의 경우 Input B가 필요없기 때문에 이에 대해서는 연결하지 않았다.

### b. Output

각 Subodule의 Output은 module마다 다른 wire로 먼저 연결시켰다. 이후, FuncCode값의 범위에 따라 어떤 Submodule의 Output을 ALU의 Output으로 연결할 지를 결정하도록 하였다. OverflowFlag를 다루는 연산을 수행하는 모듈은 AddnSub module뿐이므로, 이 모듈에만 OverflowFlag값을 output할 수 있는 wire를 연결하였으며, 다른 모듈이 사용될때는 OverflowFlag가 항상 0이 되도록 하였다.

# Implementation

## 1. ALU module

```
wire wireOverflowFlag;
wire [data_width - 1: 0] wireAddnSubC;
wire [data_width - 1: 0] wireBitwiseC;
wire [data_width - 1: 0] wireShiftC;
wire [data_width - 1: 0] wireOtherC;

ALU_AddnSub #(16) AddnSub(A, B, FuncCode, wireAddnSubC, wireOverflowFlag);
ALU_Bitwise #(16) Bitwise(A, B, FuncCode, wireBitwiseC);
ALU_Shift   #(16) Shift(A, FuncCode, wireShiftC);
ALU_Other   #(16) Other(A, FuncCode, wireOtherC);

initial begin
   C = 0;
   OverflowFlag = 0;
end

always @(*) begin
   OverflowFlag <= 0;
   if (FuncCode >= 0 && FuncCode < 2) begin
      C <= wireAddnSubC;
      OverflowFlag <= wireOverflowFlag;
   end
   else if (FuncCode >= 3 && FuncCode < 10) begin
      C <= wireBitwiseC;
   end
   else if (FuncCode >= 10 && FuncCode < 14) begin
      C <= wireShiftC;
   end
   else begin
      C <= wireOtherC;
   end
end
```

먼저, 각 submodule을 모두 만들고, Output을 Initialize하였다. Top-level module ALU에서는 input들을 submodule로 넘겨 실행시킨다. submodules의 output은 wire에 저장된다.

always 구문은 @()안의 port가 변경될 때마다 항상 실행된다. parameter가 *이므로 inputs의 변화가 있으면 실행된다. `FuncCode`에 따라 적절한 submodule의 output wire 값이 output reg C에 할당된다.

## 2. Submodules

### a. ALU_AddnSub

```
module ALU_AddnSub #(parameter data_width = 16) (
    input [data_width - 1 : 0] A,
    input [data_width - 1 : 0] B,
    input [3 : 0] FuncCode,
     output reg [data_width - 1: 0] C,
     output reg OverflowFlag
);
   always @(*) begin
      case(FuncCode)
          `FUNC_ADD : begin
            C <= A + B;
               if((A[data_width - 1] == B[data_width - 1]) && (A[data_width - 1] != C[data_width - 1]))
                        OverflowFlag <= 1;
               else  OverflowFlag <= 0;
         end
         `FUNC_SUB : begin
            C <= A - B;
            if ((A[data_width-1] == 0 && B[data_width-1] == 1 && C[data_width-1] == 1)
                || (A[data_width-1] == 1 && B[data_width-1] == 0 && C[data_width-1] == 0)) OverflowFlag <= 1;
            else OverflowFlag <= 0;
         end
         default   : C <= 0;             //do nothing
      endcase

   end

endmodule
```

always문과 case문을 이용해 `FuncCode`에 따라 addition 또는 subtraction을 할 수 있도록 한다. 표현식 data[data_width - 1]는 data의 MSB를 참조한다. input들과 계산값의 MSB를 비교하여 overflow를 detection 할 수 있다.

b. ALU_bitwise

```
module ALU_Bitwise #(parameter data_width = 16) (
   input [data_width - 1 : 0] A,
   input [data_width - 1 : 0] B,
   input [3 : 0] FuncCode,
    output reg [data_width - 1: 0] C
);

   always @(*) begin
      case(FuncCode)
         `FUNC_NOT  : C <= ~ A;
         `FUNC_AND  : C <= A & B;
         `FUNC_OR   : C <= A | B;
         `FUNC_NAND : C <= ~(A & B);
         `FUNC_NOR  : C <= ~(A | B);
         `FUNC_XOR  : C <= A ^ B;
         `FUNC_XNOR : C <= ~(A ^ B);
         default    : C <= 0;             //do nothing
      endcase
   end

endmodule
```

마찬가지로 always문과 case문을 이용해 적절한 비트연산을 수행한다.

NOT: 0은 1로 1은 0으로 바꿔주는 비트연산이다.

AND: 1-1인 경우 1, 그 외는 0으로 바꿔주는 비트연산

OR: 0-0인 경우 0, 그 외는 1로 바꿔주는 비트연산

NAND: 1-1인 경우 0, 그 외는 1로 바꿔주는 비트연산으로 and 연산에 not을 하여 구현한다.

NOR: 0-0인 경우 1, 그 외는 0으로 바꿔주는 비트연산. 마찬가지로 or + not으로 구현한다.

XOR: 1-0 또는 0-1인 경우 1로 계산하는 비트연산

XNOR: 1-0 또는 0-1인 경우 0으로 계산해주며 xor+not으로 구현한다.

c. ALU_shift

```
module ALU_Shift #(parameter data_width = 16) (
   input [data_width - 1 : 0] A,
   input [3 : 0] FuncCode,
    output reg [data_width - 1: 0] C
);

   always @(*) begin
      case(FuncCode)
         `FUNC_LLS  : C <= A << 1;
         `FUNC_LRS  : C <= A >> 1;
         `FUNC_ALS  : C <= A <<< 1;
         `FUNC_ARS  : begin
            C <= A >>> 1;
            if (A[data_width - 1] == 1) C[data_width - 1] <= 1;
         end
         default : C <= 0;             //do nothing
      endcase
   end

endmodule
```

위와 마찬가지로 function code에 따라 적절한 연산을 수행한다.

LLS: logical left shift. 왼쪽으로 1칸 이동하는 비트연산. 빈자리는 0을 채운다.
LRS: logical right shift. 오른쪽으로 1칸 이동하는 비트연산. 빈자리는 0을 채운다.
ALS: arithmetic left shift. 왼쪽으로 1칸 이동하는 비트연산. 빈자리는 0을 채운다.
ARS: arithmetic right shift. 오른쪽으로 1칸 이동하며 빈자리는 MSB를 복제하여 채운다.

d. ALU_Other

```
module ALU_Other #(parameter data_width = 16) (
   input [data_width - 1 : 0] A,
   input [3 : 0] FuncCode,
    output reg [data_width - 1: 0] C
);

   always @(*) begin
      case(FuncCode)
         `FUNC_ID   : C <= A;
         `FUNC_TCP  : C <= ~A + 1;
         `FUNC_ZERO : C <= 0;
         default : C <= 0;              //do nothing
      endcase
   end

endmodule
```

위와 마찬가지로 function code에 따라 적절한 연산을 수행한다.
TCP는 2의 보수를 의미하며 not 연산으로 1의 보수를 구하고 1을 더한다.

3. Full Scenario

처음 top level module이 실행되면 output 값을 초기화한다. 각 Submodule에서는 Input A, B, FuncCode 중 하나의 값이 변경되면 always 구문이 작동하여 submodule의 Output이 변경되며, Top level module에서는 FuncCode가 변경되면 always문에서 function code에 맞는 submodule의 output이 최종 output에 연결되게 된다.

# Discussion

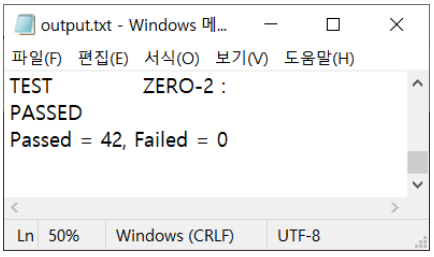
Module들 간에 연결하는 과정에서 어려움이 있었다.

처음에는 always문 내부에서 각 case에 따라 4개의 모듈을 실행하는 것이 가능할 것이라 생각했는데, simulation하는 과정에서 module이 완전한 function의 개념이 아니기 때문에 불가능함을 깨달았다.

이후 이를 해결하기 위해, case에 따라 module을 바꾸는 것이 아니라, 각 module의 output을 각각 다른 wire로 뽑아낸 뒤, 내부 module의 wire를 최종 top level module의 output에 case에 따라 다르게 연결하는 방식으로 구현하였더니 제대로 simulation이 돌아가는 것을 확인할 수 있었다.

# Conclusion

이번 Lab에서는 ALU를 Verilog에서 Implement하는 것을 목적으로 하였다. 이를 통해 ModelSim 및 Verilog의 사용법을 익힐 수 있었고, Modularization을 구현하는 과정에서 verilog에서 여러 module을 사용할 수 있게 되었다.



구현한 ALU에 대해 TestBench를 모두 Pass한 것으로 아래와 같이 output.txt에서 확인하였고, ALU도 성공적으로 Implement하였다.