Lab #1. ALU

2021 Computer Architecture Lab Report

20190065 강두경, 20190316 유병호

Introduction

이번 lab의 목표는 Verilog를 이용해 ALU Module를 디자인하는 것이다. ALU는 Arithmetic Logic Unit의 약자로, 덧셈 뺄셈 등의 산술 연산과 and, or 같은 논리 연산을 모두 지원한다.

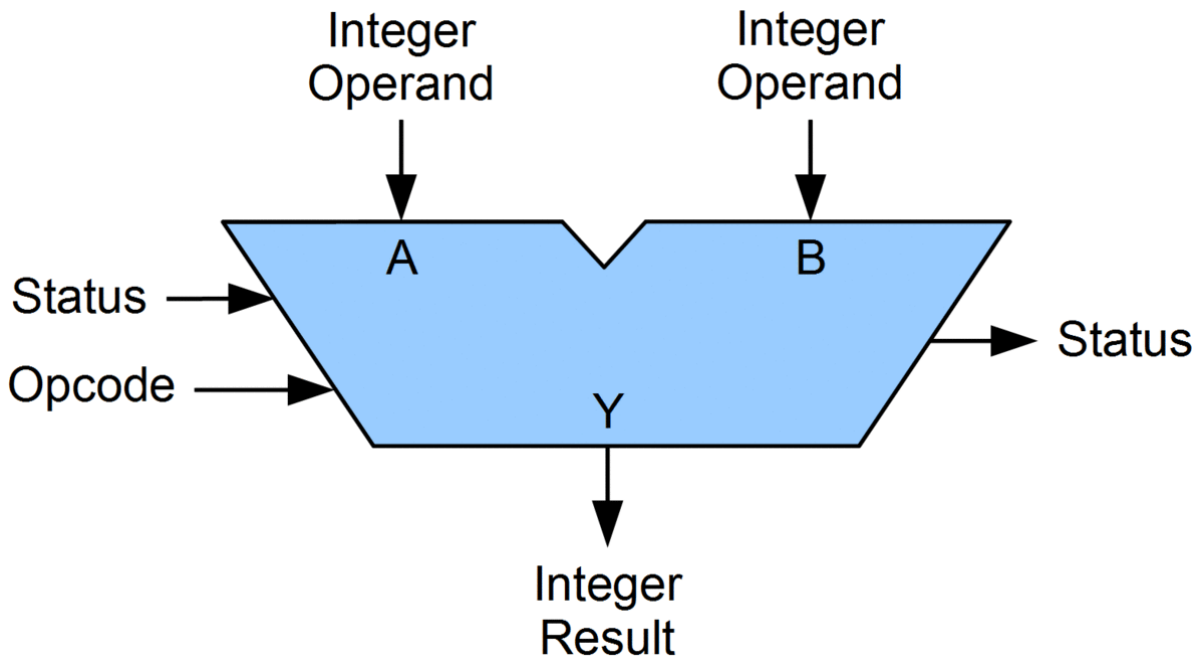

Figure 1. Arithmetic Logic Unit

ALU는 2개의 Operand와 Opcode 역할을 하는 FuncCode를 받아, Result를 연산한다. 본 Lab에서는 add, sub 연산에 대해 overflow 여부를 확인하는 OF 또한 출력값으로 갖는다. FuncCode는 4비트의 값을 갖는 정수이며, 대응되는 연산은 다음과 같다.

Aa FuncCode	Operation	Comment
0000	A + B	Signed Addition

FuncCode	Operation	Comment
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
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

위와 같이 대응되게 ALU를 디자인한다. 본 Lab을 통해 목표하는 것은 다음과 같다.

- ALU를 Verilog로 구현함으로써 Verilog의 모듈화 및 always, assignment, case, parameter 등의 기능에 대해 익힌다.
- ALU 단위에 대해 이해함으로써 앞으로 ALU를 잘 활용한다.

Design

우선 ALU를 모듈화하기 위해 ALU라는 모듈이 존재한다. ALU의 input, output은 다음과 같다.

- **input**
  - A : 크기가 data_width인 vector로, operand에 해당한다.
  - B : 크기가 data_width인 vector로, operand에 해당한다.
  - FuncCode : 크기가 4인 vector로, opcode에 해당한다.
- **output**
  - C : 크기가 data_width인 vector
  - OverflowFlag : 오버플로우 여부를 나타내는 비트.

FuncCode에 따라 다른 연산으로 이동하는 것은 간단히 case 문으로 구현하였다. 0000~1111까지의 FuncCode가 정확히 어떤 연산자를 의미하는지 모호하기 위해 alu_func.v에서 제시되어 있는 매크로를 활용하였다. case 문 내의 assign은 non-block assignment(`<=`)를 원칙으로 했다.

Add, Sub의 경우 Overflow를 따로 계산했다. A, B, C의 MSB 값에 따라 Overflow가 1이 될지 0이 될지 여부가 결정된다.

data_width는 parameter이기 때문에 호출할 때 변화를 줄 수 있다. 기본값은 16이다.

Implementation

우선 alu.v에서 always 구문을 이용했고, 안에 case 문을 통해 특정 FuncCode에 따라 C를 서로 다른 방법으로 연산하도록 했다. 기본적으로 연산은 나온 것처럼 똑같이 하면 되며, 예외로 xor 문의 경우 ^ 연산자를 사용하면 된다. XNOR 연산의 경우 Verilog에서는 `~^`, `^~` 연산이 모두 가능하다.

오버플로우의 경우, MSB가 sign을 가리키는 것인 만큼, add, sub에 따라 오버플로우가 되는 조건은 다음과 같다. a, b, c를 각각 A, B, C의 sign bit라 했을 때

ADD :

`ADD :` $(a,b,c) = (0,0,1)$ or $(a,b,c) = (1,1,0)$

`SUB :` $(a,b,c) = (0,1,0)$ or $(a,b,c) = (1,0,0)$

이다. ADD의 경우, A와 B의 부호가 같아야 overflow가 발생하고 C는 A, B와 달라야 한다. SUB의 경우 A, B의 부호가 달라야 overflow가 발생하고 C는 B와 부호가 같아야 한다. 다른 연산을 할 때는 OF가 0이 되어야 하므로 case 문 진입 전에 OF에 0을 대입한다. 그러지 않으면 마지막 sub 연산 이후 OF 값이 계속 유지되기 때문이다.

다음으로 주의할 점은 >>>의 경우 signed 연산이기 때문에 바로 사용하면 제대로 해결되지 않는다. 이 문제의 경우 `$signed(A) >>> B` 라고 쓰면 해결된다.

```
`include "alu_func.v"
```

```
always @(*) begin
  OverflowFlag <= 0;
  case (FuncCode)
    `FUNC_ADD : begin
      C = A + B;
      OverflowFlag <= A[data_width - 1] & B[data_width - 1] &
~C[data_width - 1] | ~A[data_width - 1] & ~B[data_width - 1] & C[data_width - 1];
    end
```

```
    `FUNC_SUB : begin
      C = A - B;
      OverflowFlag <= ~A[data_width - 1] & B[data_width - 1] &
C[data_width - 1] | A[data_width - 1] & ~B[data_width - 1] & ~C[data_width - 1];
    end
    `FUNC_ID : C <= A;
    `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;
    `FUNC_LLS : C <= A << 1;
    `FUNC_LRS : C <= A >> 1;
    `FUNC_ALS : C <= A <<< 1;
    `FUNC_ARS : C <= $signed(A) >>> 1;
    `FUNC_TCP : C <= ~A + 1;
    `FUNC_ZERO : C <= 0;
    default : C <= 0;
  endcase
end
```

alu_tb.v Testbench에서 26번째 줄 모듈을 실행하는 코드는 다음과 같이 작성하면 된다.

```
ALU #(16) alu(
  .A(A),
  .B(B),
  .FuncCode(FuncCode),
  .C(C),
  .OverflowFlag(OverflowFlag)
);
```

Discussion

지금까지 FuncCode, A, B를 input으로 갖고 결과값인 C와 조건 코드 역할을 하는 OverflowFlag를 output으로 갖는 ALU Module을 Verilog로 구현했다. 주안점은 다음과 같다.

- case 문과 연산자를 이용해 구현했기에 구현 난이도는 쉬우나, 얼마나 최적화되어 있을지 여부는 불확실하다. ALU는 굉장히 많이 쓰이는 유닛으로 최대한 최적화해야 하나, gate 개수, wire 개수 등의 complexity가 최소한의 개수를 가질지는 불명확하다.
- Arithmetic Right Shift 연산의 경우 $signed를 사용해 해결했는데, 근본적으로 어떤 원리로 해결되는지에 대해 탐구해 보면 좋을 것 같다.

추가로, Modelsim GUI를 이용하지 않고 Visual Studio Code 환경만을 이용해 시뮬레이션을 구동하였다. Circuit 모형이나 Waveform이 필요할 때는 Modelsim GUI를 사용해야 하지만, 그렇지 않다면 Command Line Mode를 이용해 쉽게 시뮬레이션을 구동할 수 있다. 우선

`vlog` 명령어를 이용해 컴파일하고, `vsim -c <module name>`으로 시뮬레이션을 돌린다. run을 하고 싶으면 이후 vsim REPL에서 `run` 명령어를 사용하면 된다. 이렇게 하면 추가로 Modelsim GUI 없이 Testbench를 구동해 ouput, 컴파일 결과 및 오류 등을 열람할 수 있다.

Conclusion

Arithmetic Logic Unit을 Verilog로 구현해 보았다. Testbench 결과 42개의 pass를 얻을 수 있었다. 가장 기본적인 연산 유닛인 만큼 의미가 있었고, Verilog의 Behavioural Modeling을 직접 사용해 보는 것에 있었어도 가치가 컸다. 뿐만 아니라 이전에는 잘 모르고 썼던 HDL Verilog의 기본적인 원리와 syntax를 조금 더 알고 쓸 수 있었다.