20170699 최진성 20170080 김동규 Lab Report

Lab 1. ALU Lab

1. Introduction

ALU(Arithmetic Logic Unit)이란 산술 논리 연산 장치로, CPU/GPU 등에서 실제로 연산을 수행하는 디지털 회로를 의미한다. 주로 산술 연산과 논리 연산을 수행하고, 부가적으로 비교 연산, 보수 연산, 시프트 연산 등도 수행한다.[1] Lab 1에서는 Verilog programming language를 사용하여 ALU module을 구성해 본다. 요구 사항을 모두 만족하기 위해, two's complement binary number와 bit calculation에 대한 내용을 이해하고 있어야 한다. ALU module을 구성해 봄으로써 기본적인 Verilog 문법을 익히고 ALU의 작동원리에 대해 파악한다. 또한 ModelSim 프로그램 사용법과 시뮬레이션 방법을 익혀 추후 진행될 lab들에 필요한 기본적인 능력들을 숙지하였다.

2. Design

Lab 1에서 구현할 ALU module은 2개의 16-bit signed binary numbers를 input으로 받아 4-bit binary number로 설정된 OP에 해당하는 연산을 수행하여 output으로 내보낸다. 문제에서 주어진 16가지 서로 다른 종류의 연산이 가능하도록 하며, 이를 그림으로 나타내면 다음과 같다.

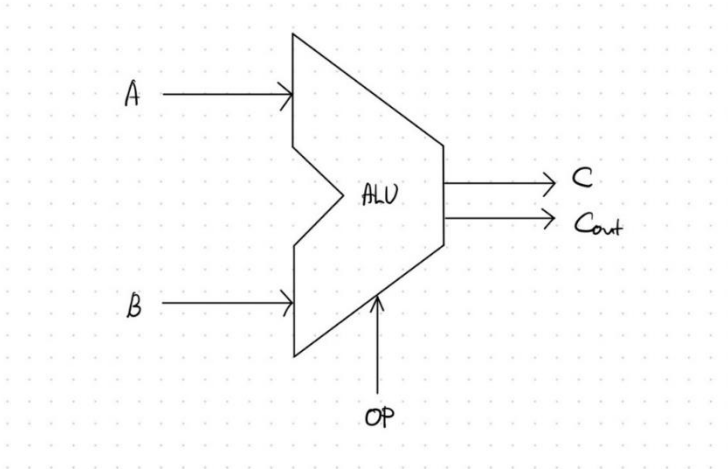

Figure 1. Lab 1에서 구현하는 ALU의 구조

본 lab에서는 addition과 substraction 연산의 경우 overflow를 detect하도록 요구하고 있다. Overflow가 발생할 경우 Cout=1, overflow가 발생하지 않을 경우 Cout=0으로 설정되도록 설계하였다.

[1] http://www.ktword.co.kr/test/view/view.php?m_temp1=374

3. Implementation

해당 ALU는 두개의 16-bit signed binary number input에 대해 4-bit binary number인 OP 값에 따라 각기 다른 연산을 실행한다. Input에 해당하는 A와 B는 wire로, output에 해당하는 C와 Cout은 reg로 설정하였다. OP에 따라 다른 연산을 배정하기 위해 OP값을 활용한 case문으로 코드를 구성하였다. Cout의 initial값은 0으로 설정하였고 addition과 substraction에서 overflow가 발생할 경우 1로 값이 지정되도록 하였다.

Addition과 substraction 연산에서 overflow가 발생하는 경우는 A, B, C의 msb(most significant bit)를 확인해보면 알 수 있다. Signed binary number의 경우 msb가 부호를 결정하므로 A[15], B[15], C[15]의 관계를 확인하면 언제 overflow가 발생하는지 알 수 있다. Addition 연산의 경우 (양수)+(양수)=(음수)인 경우와 (음수)+(음수)=(양수)인 경우 overflow가 발생하는 것을 알 수 있다. 즉, A[15]=0, B[15]=0, C[15]=1 (~A[15] & ~B[15] & C[15]) 일 때와 A[15]=1, B[15]=1, C[15]=0 (A[15] & B[15] & ~C[15]) 일 때 Cout=1이 되도록 설정하였다. Substraction 연산의 경우 (양수)-(음수)=(음수)인 경우와 (음수)-(양수)=(양수)인 경우 overflow가 발생하는 것을 알 수 있다. 즉, A[15]=0, B[15]=1, C[15]=1 (~A[15] & B[15] & C[15]) 일 때와, A[15]=1, B[15]=0, C[15]=0 (A[15] & ~B[15] & ~C[15]) 일 때 Cout=1이 되도록 설정하였다.

문제에서 구현하도록 요구하는 연산들은 대부분 verilog내에 predefined operation으로 정의되어 있기에 이를 이용하여 16가지 연산들을 처리하였다. 이 때 Arithmetic Right Shift의 경우, 이동한 만큼 MSB와 같은 값으로 채워줘야 함으로 Bitwise shift를 한 칸 수행한 뒤에 결과값의 MSB인 C[15]와 초기값의 MSB인 A[15]를 일치시키는 방식으로 재설정하였다.

4. Evaluation

완성된 코드를 컴파일한 이후 주어진 ALU_TB.v testbench 파일을 사용하여 시뮬레이션 결과를 확인하였다. 그 결과 모든 50 개 테스트에 대해서 다음과 같이 50 개의 정상적인 결과값이 나온다는 사실을 확인하였다.

```
# Passed = 50, Failed = 0
# ** Note: $finish    : C:/modelsim_project/lab1/ALU_TB.v(55)
#    Time: 50 ns  Iteration: 0  Instance: /ALU_TB
```

Figure 2. 주어진 Testbench 파일로 실행한 시뮬레이션 결과

추가적으로 기존 testbench 에는 없는 Subtraction 에서의 overflow dectection 이 정상적으로 작동하는지 확인하기 위하여 주어진 testbench 코드를 다음과 같이 overflow 가 일어나는 상황으로 바꾸어 테스트하였고, 이 역시 정상적으로 통과하는 결과를 얻을 수 있었다.

```
Test("Sub-7", 16'h8000, 16'h0001, 16'h7fff, 1);
//Original Line: Test("Sub-7", 0, 16'hffff, 1, 0);
# TEST                Sub-7 :
# PASSED
```

Figure 2. 수정한 Testbench 파일과 해당 시뮬레이션 결과

이를 통해 구현한 ALU의 16가지 동작들이 모두 정상적으로 작동하고, overflow dectection 역시 addition과 subtraction 상황에서 모두 정상적으로 작동한다는 것을 확인하였다.

5. Discussion

초반 verilog의 문법과 코딩 방식이 익숙하지 않아 초반 코딩에 비교적 많은 시간을 소비하였고, ModelSim의 경우 기존에 쓰던 VScode 등과 코딩, 디버깅, 컴파일, 그리고 러닝 환경에 차이가 있어 익숙해지는 과정이 필요하였다. 이 때 참고 자료로 주어진 Verilog 설명 자료와 인터넷을 참고하였고, 문법과 코딩 환경이 익숙해진 이후에는 ALU 모듈을 구성하는 과정에 큰 어려움이 없었다.

6. Conclusion

가장 기본적인 연산장치인 ALU를 verilog로 구현함으로서 ALU의 작동 방식과 기본적인 verilog programming langauge의 특성 및 문법을 이해할 수 있었다. 이번 과제를 통해 앞으로 EE312 수업 동안 마주하게 될 verilog의 코딩에 익숙해질 수 있었고, 작성한 verilog 코드를 컴파일하고 실행하는 ModelSim이 어떻게 동작하는지, test bench 파일과 함께 시뮬레이션 및 결과값을 어떻게 확인하는지를 직접 해보며 알게 되었다.