CSED311 Lab 1 : ALU

20170684 강덕형
20170735 이승준

Introduction

설계 및 구현 목표

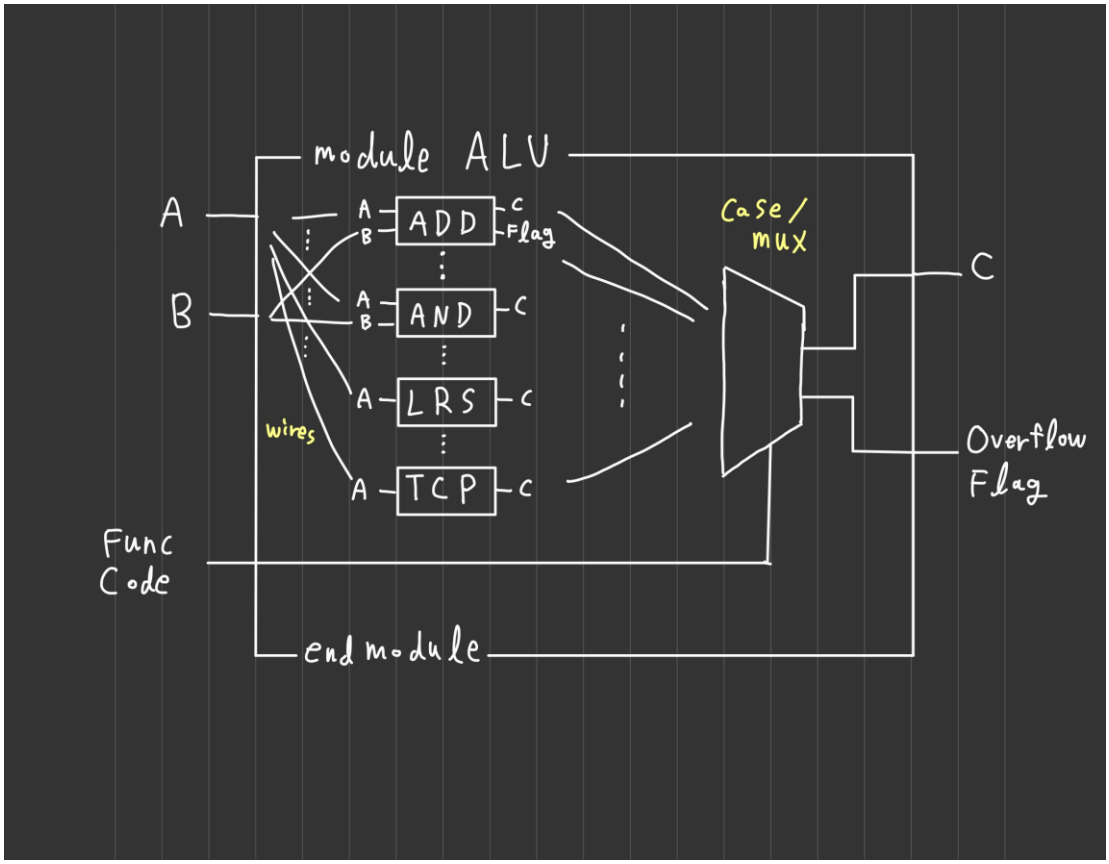
이번 Lab 과제는 ALU(Arithmetic Logic Unit)을 설계 및 구현하는 것이 목표이다. ALU는 산술연산과 논리연산을 담당하는 회로이다. ALU 모듈은 CPU(Central Processing Unit)의 서브모듈로 활용되며, CPU 의 계산 과정에서 사용된다.

학습목표

이번 Lab 과제를 통해 HDL(Hardware Description Language)의 한 종류인 Verilog를 이해한다. CPU의 하위 모듈이자 기초적인 회로인 ALU를 구현하며, 산술연산과 논리연산을 이해하고 verilog modularization을 익힌다.

Design

1. ALU Specification 의 각 연산마다, 하나의 모듈을 설계한다.
2. 각 연산 모듈은, ALU 로부터 input 을 받아 담당하는 연산을 수행해 output 을 계산한다.
3. wire 를 이용해, 각 연산 모듈들의 입력값에 ALU 의 입력값을 전달한다.
4. 각 연산 모듈들의 output은 MUX(multiplexer)에 연결되어 있고 concurrent하게 연산을 수행중이다.
4. MUX 는 FuncCode 에 해당하는 연산 모듈만의 output 을, ALU 의 output 에 assign 한다.



[fig 1] The Diagram of ALU design

Implementation

우리는 수행하는 연산이 유사한 모듈들을 하나의 파일에 구현하였다. 이에 프로젝트는 연산 모듈들이 담겨 있는 module_*.v 파일, 그리고 연산 모듈들을 이용해 ALU 모듈을 구현한 alu.v 로 이루어져 있다.

module_add_sub.v = {ADDModule, SUBModule)
ADDModule: 2개의 16-bit signed input을 받아 Signed Addition, Overflow detection을 수행한다.
SUBModule: 2개의 16-bit signed input을 받아 Signed Addition, Overflow detection을 수행한다.
Overflow detection 은 operand 들의 MSB 와, output 의 MSB 를 이용해 계산했다.

module_bitwise.v = {NOTModule, ANDModule, ORModule, NANDModule, NORModule, XORModule, XNORModule}
NOTModule: 1 개의 16-bit input을 받아, not 연산을 수행한다.
그 외: 2 개의 16-bit input을 받아, AND, OR, NAND, NOR, XOR, XNOR 연산을 각각 수행한다.

module_shift.v = {LLSModule, LRSModule, ALSModule, ARSModule}

1 개의 16-bit input 을 받아 LLS, LRS, ALS, ARS 연산을 수행한다.
이 때 ARS 연산의 경우 verilog에서 제공하는 ARS operator (>>>) 를 input에 바로 적용시키면 MSB 를 보존하지 않았다. 이를 해결하기 위해 input을 signed integer로 명시하여 ARS operator를 적용했다.

module_others.v = {IDModule, TCPModule, ZEROModule}

IDModule: 1 개의 16-bit input을 받아, Identity(A) 연산을 수행한다.
TCPModule: 1 개의 16-bit input을 받아, 2의 보수 연산을 수행한다.
ZEROModule: 0 을 output 으로 전달한다.

alu.v:

위의 module_*.v 파일들의 module 들을 instantiation 한다. 그 후, ALU 모듈의 input 을 각 연산모듈의 인스턴스들의 input 에 연결한다. 각 연산모듈의 연산값을 받아올 wire 들을 ALU 모듈 내에 선언하고, 연산모듈의 output 들에 wire 들을 연결한다. 그리고 always statement 를 이용해 ALU 모듈의 input 에 해당하는 A 또는 B 또는 FuncCode 가 변할 때마다 FuncCode 를 확인한다. Case statement 를 통해, 각 FuncCode 에 해당하는 연산모듈의 output wire 를 ALU 모듈의 output 에 연결한다.

Discussion

Module design was changed

본래 우리는 각 연산마다 모듈을 구현하는 것이 아니라, 각 file 마다 하나의 모듈을 구현하는 것을 계획했다. 예를 들어, module_bitwise 에서 case (FuncCode) 를 이용해, FuncCode 에 해당하는 bitwise 연산을 수행해 out 에 연결해주는 방식으로 디자인했다. 그러나 test bench 시뮬레이션 결과, n 번째 task 의 ALU output 값이 (n+1) 번째 output 에 출력이 되는 문제가 발생했다.
이 문제를 해결하기 위해 여러 방안을 시도해보았으나, 결국 동일한 디자인에서는 해결하지 못했다. 이에 지금처럼 각 연산마다 하나의 모듈을 만들어 연산값을 assign 하는 디자인으로 선회하였다.
해당 문제에 원인에 대해 우리는 한 가지 가설을 추측하였다. 과거 디자인의 경우 ALU 모듈에서 always-case문 한번, 연산 모듈에서도 always-case문을 한번씩 사용하였다. 이러한 과정에서 두 always문의 sensitivity list는 동일하다. 따라서 sensitivity list에 변화가 발생하면, 두 always문은 동시에 실행되게 된다. 이 때 첫 번째 always문의 경우 연산 모듈의 결과를 ALU 모듈의 결과에 assign하는 역할을 수행하는 반면, 두 번째 always문의 경우 연산 모듈의 결과를 update한다.하지만,

둘 중 무엇이 먼저 실행이 되는지 deterministic 하지 않기 때문에, ALU 모듈이 연산모듈의 결과를 가져올 때 연산모듈의 결과가 최신인지 보장할 수 없다. 이에 test bench 에서 ALU 모듈의 결과 값 순서가 혼동되는 문제가 발생했다고 추측한다.

Use of `$signed` in ARS module

$signed 가 없던 최초 구현에서는, verilog operator >>> 가 >> 와 동일하게 작동했다. 시뮬레이션 결과 verilog 는 $signed 키워드가 없으면, 기본적으로 상수를 unsigned 로 인식하는 것으로 보여진다. 이에 우리는 ARS module 의 operand 에 $signed 키워드를 붙여주어, >>> 가 signed operand 를 shift 하게 하도록 구현했다.

Conclusion

우리는 본 과제에서 ALU 를 설계 및 구현했다. 또한 ALU 에서 수행하는 산술연산과 논리연산을 이해했다. 구현 과정에서 verilog 의 기초 문법 및 작동 원리를 익혔다. 세세한 modularization 을 통해 ALU 를 구현하며, ALU 의 작동 원리를 전반적으로 이해할 수 있었다.