컴퓨터 공학 기초 실험2 보고서

실험제목: Arithmetic Logic Unit

실험일자: 2020년 09월 28일 (월)

제출일자: 2020년 10월 2일 (금)

학 과: 컴퓨터공학과

담당교수: 이준환 교수님

실습분반: 금요일 5,6,7

학 번: 2019202009

성 명: 서여지

1. 제목 및 목적

A. 제목

Arithmetic Logic Unit

B. 목적

adder를 이용한 뺄셈 연산의 원리를 이해하고 실제 ALU제작에 응용한다. Arithmetic Logic Unit의 용도와 구조를 알아본다. ALU에서 사용하는 flag의 의미와 활용을 이해한다. 지금까지 실습에서 제작한 module들을 이용해 4bits ALU를 설계한다. ALU설계에 필요한 multiplexer를 구현한다. 4bits ALU를 이용해 32bits ALU를 제작한다.

2. 원리(배경지식)

1) Subtractor

2진수 체계에서 두 수의 뺄셈계산을 하는 방법에는 2's complement를 이용한 방법이 있다. 2의 보수법이라고 부르는 이 방법을 통해 뺄셈 계산을 하는 방법은, 빼려는 수의 1의 보수를 구한 뒤 1과 함께 더해주는 것이다. 이것을 식으로 나타내면 $A - B = A + B' + 1$ 로 표현할 수 있다. 따라서 뺄셈 계산은 모든 bit를 반전시켜 1의 보수를 구할 수 있는 inverter와, 덧셈을 계산하는 adder를 이용하여 계산할 수 있다. 이번 실습에서 Subtractor를 구현할 때는 adder module의 b port에 ~b를 입력하고, cin에 1을 입력하여 뺄셈 계산이 가능하도록 작성하였다.

2) Arithmetic Logic Unit

Arithmetic Logic Unit은 두 수 A, B에 대한 여러가지 연산의 결과를 모두 계산한 뒤, ALU에 전달된 opcode에 따라 한 가지 값을 선택하여 출력하는 회로이다. 이번 실습에서는 not A, not B, A and B, A or B, A xor B, A xnor B, A add B, A sub B로 총 8가지 계산의 결과를 얻을 수 있는 ALU를 설계하는 것을 목적으로 한다. 8가지 경우의 수를 나타낼 수 있는 최소 bit수는 3bit이기 때문에, 이 ALU가 사용하는 opcode의 크기는 3bit이다.

ALU는 각 연산을 진행하는 module뿐만 아니라 opcode를 s입력값으로 갖는 multiplexer를 포함한다. 이것은 각 module에서 계산된 여러 개의 연산결과 중 입력된 opcode에 해당하는 한가지 결과만을 선택하여 출력하기 위해서이다. 또한 A와 B의 계산이 종료된 후, 상태를 나타내는 status flag를 구하기 위한 회로도 존재한다.

3) Flag

ALU는 연산을 진행한 뒤, 그 연산결과에 맞는 flag를 update한다. 이 flag값을 이용하여 비교연산을 할 수 있다. 이번 실험에서는 C, N, Z, V 로 총 4개의 flag를 나타내는 ALU를

설계한다. flag가 표시되려면 각각의 조건을 만족시켜야 한다. C는 carry flag이다. ALU에서 수행한 연산에서 carry가 발생한 경우 1이 된다. N은 negative flag이고, 연산결과가 음수인 경우, 즉 result의 MSB가 1인 경우 표시된다. Z flag는 zero flag이다. 연산결과가 0인 경우 1로 표시된다. 마지막으로 v flag는 overflow flag이다. 양수와 양수를 더하는 것처럼 같은 부호의 두 수를 더하거나 양수에서 음수를 빼는 것과 같이 다른 부호의 두 수를 빼는 계산을 할 때, 연산 결과가 ALU가 출력하는 result의 크기보다 커져서 올바르게 표현이 불가능한 경우가 발생할 수 있다. 이렇게 overflow가 발생하는 경우 v flag가 1이 되어 표시해준다.

4) carry와 overflow의 차이, 발생여부 확인

carry는 연산 중에 MSB에서 올림수가 발생한 것이다. 반면 overflow는 계산 결과가 너무 커서 주어진 bit 수로 나타낼 수 없는 경우이다. 이 경우 연산결과가 정상적으로 저장되지 못하기 때문에 오류가 발생한다. 2bit의 덧셈을 예를 들어 살펴보면 다음과 같다.

2진수 표현	10진수 표현	carry	overflow
01 + 10 = 11	1 - 2 = -1	0	0
01 + 01 = 10	1 + 1 = -2	0	1
01 + 11 = (1)00	1 - 1 = 0	1	0
11 + 10 = (1)01	-1 - 2 = 1	1	1

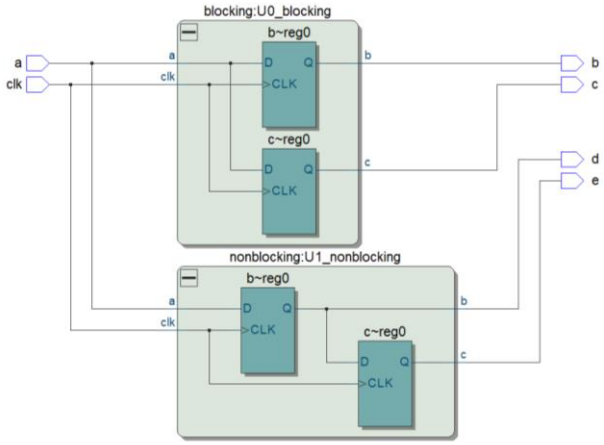
overflow가 발생한 경우 계산 결과가 정상적으로 저장되지 않았음을 확인할 수 있다. 이러한 현상은 수의 표현에서 MSB가 부호를 나타내기 때문에 발생한다

carry의 발생 여부는 carry out의 값을 통해 알 수 있다. overflow의 발생 여부는 최상위 2bit에서 carry의 발생여부를 xor연산하면 알 수 있다. 두 bit 모두 carry가 발생하지 않았다면 bit 수를 넘기지 않으므로 overflow가 일어나지 않고, 두 bit 모두 carry가 발생한 경우 연산한 두 수 모두 MSB가 1인 음수였고, 계산의 결과도 마찬가지로 음수로 나타난다는 의미이기 때문에 overflow가 일어나지 않았음을 알 수 있다. 반면 MSB에서만 carry가 발생한 경우 두 음수를 더한 값이 양수가 되었으므로 overflow임을 알 수 있다. 마지막으로 MSB는 carry가 발생하지 않고 MSB의 다음 bit에서 carry가 발생했다면 두 양수를 더한 값이 음수가 된 것이므로 이 경우에도 overflow가 발생했다는 것을 알 수 있다.

5) blocking, non-blocking

always문 내부에서 assignment를 표현할 때 '='를 이용하여 blocking assignment를 표현한다. 해당 실행문이 실행완료 되기 전까지, 다음 실행문은 실행되지 않는다. 이러한 동작을 one pass라고 한다.

반면 non-blocking assignment는 '<='를 이용하여 표현한다. assign 명령어의 오른쪽 항(RHS)이 먼저 update되고, 왼쪽 항(LHS)은 모듈 내의 모든 오른쪽 항이 update 된 이후에 update된다. 따라서 이전 명령어에 의해 실행하려는 문장의 오른쪽 항의 값이 변경되어도 영향을 받지 않는다.



주어진 코드를 컴파일하고, RTL Viewer를 확인해보면 그 차이를 분명하게 할 수 있다. blocking을 사용한 module은 b와 c가 결국 같은 값으로 저장되지만, non-blocking module은 d와 e가 서로 다른 값으로 저장될 수 있다.

3. 설계 세부사항

1. 4-bits ALU

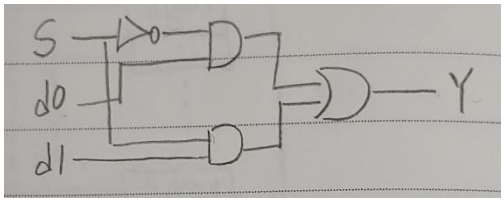
ALU를 크게 세 가지 부분으로 나누어 설계할 수 있다. A와 B입력에 대한 출력 값을 연산하는 부분, multiplexer를 이용해 opcode에 해당하는 result를 결정하는 부분과 flag를 계산하는 부분으로 분리해서 설계한다. 실습에서 설계한 4bit ALU는 not A, not B, A and B, A or B, A xor B, A xnor B, A add B, A sub B의 8가지 연산을 하므로 각각을 연산하는 gate module을 instance하여 사용한다. multiplexer는 8가지 입력 중 하나의 입력을 선택하는 8 to 1 mux형태로 설계한다. 8 to 1 mux는 다시 기본이 되는 2 to 1 mux 여러 개로 구성된다. 2 to 1 mux의 진리표와 k-map, Boolean equation은 다음과 같다.

s	d0	d1	y
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

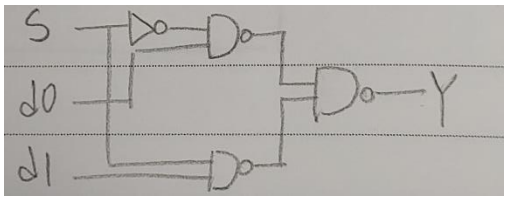
s \ d0d1	00	01	11	10
0	0	0	1	1
1	0	1	1	0

$$Y = \bar{S}d0 + Sd1$$

이 식을 and gate와 or gate를 이용해 구현하면 다음과 같다.



드모르간 법칙을 이용해 nand gate를 이용한 회로로 바꾼다.



설계한 2 to 1 mux를 이용하여 8 to 1 mux를 만든다. 8개의 input을 4개의 2 to 1 mux에 각각 연결하고, s값은 s[0]을 연결한다. 4개의 mux에서 나오는 출력을 다시 2개의 2 to 1 mux에 연결하고, s값으로 s[1]을 입력한다. 두 개의 mux의 출력 값은 s[2]를 s값으로 하는 2 to 1 mux의 입력 값으로 넣어준다.

마지막으로 flag를 계산하는 module을 제작한다. 이 실험에서 사용하는 4개의 flag는 각각 다음의 방법으로 구할 수 있다.

C: add 혹은 sub 연산이 실행된 경우 adder의 연산 결과에서 carry out값을 출력한다.
N: mux의 output인 result의 MSB가 1인 경우 1로 표시된다.
Z: result가 0인 경우 1로 표시된다.
V: add 혹은 sub 연산이 실행된 경우 adder의 연산결과에서 carry out과 carry_prev 값을 xor한 결과를 출력한다.

이때 v flag를 계산하기 위해서 alu에 사용되는 adder는 carry out과 함께 carry_prev를 출력할 수 있도록 수정한다.

2. 32-bits ALU

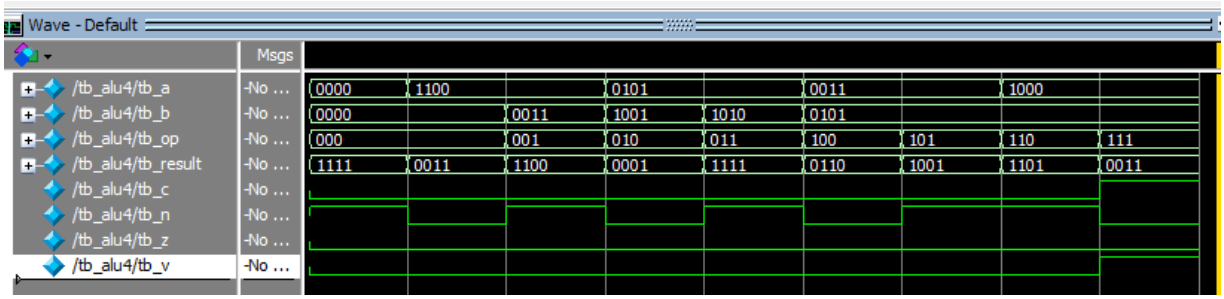
4bit ALU에서 사용했던 module의 input과 output을 32bit로 수정하여 새로운 module을 생성한다. 새로 생성한 module을 이용해 4bits ALU와 같은 방법으로 32bits ALU를 설계한다.

4. 설계 검증 및 실험 결과

A. 시뮬레이션 결과

1. 4 bits ALU

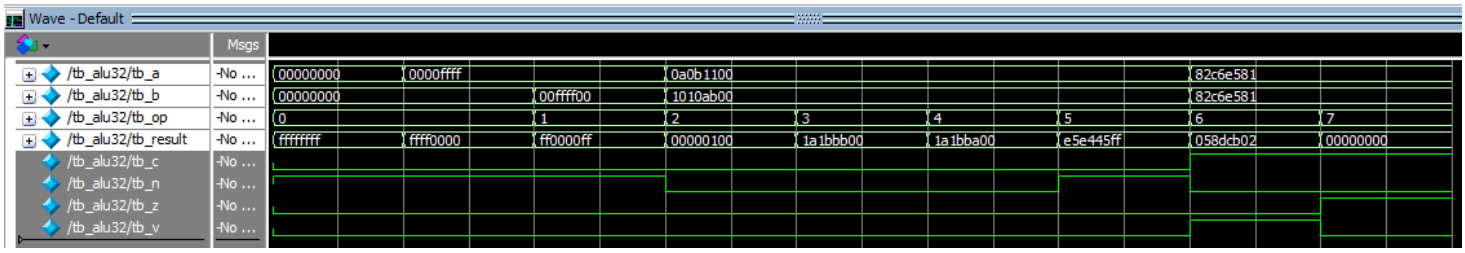
testbench는 강의자료에 제시된 값을 이용하여 작성하였고, add와 sub명령어를 이용하는 경우를 추가하였다.



0ns	~10ns	~20ns	~30ns	~40ns	~50ns	~60ns	~70ns	~80ns	~90ns
tb_a	0000	1100		0101		0011		1000	
tb_b	0000		0011	1001	1010	0101			
tb_op	000		001	010	011	100	101	110	111
tb_result	1111	0011	1100	0001	1111	0110	1001	1101	0011
tb_c	0								1
tb_n	1	0	1	0	1	0	1		0
tb_z	0								
tb_v	0								1

op의 값에 따라 지정된 연산결과가 출력되는 것을 확인할 수 있다. flag도 적절하게 표시되었다. result의 MSB가 1일 때 n flag의 값이 1이 되고, 그 자리에서 carry가 발생한 마지막 case에서 c flag도 1이 되었다. 계산 결과가 0이 되는 경우는 없기 때문에 z flag는 항상 0이고, 1000 – 0101 연산으로 값의 범위를 벗어난 결과가 나타난 80~90ns 구간에서 v flag가 정상적으로 출력되었다.

2. 32bits ALU



0ns	~10ns	~20ns	~30ns	~40ns	~50ns	~60ns	~70ns	~80ns	~90ns
tb_a	00000000	0000ffff		0a0b1100				82c6e581	
tb_b	00000000		00ffff00	1010ab00				82c6e581	

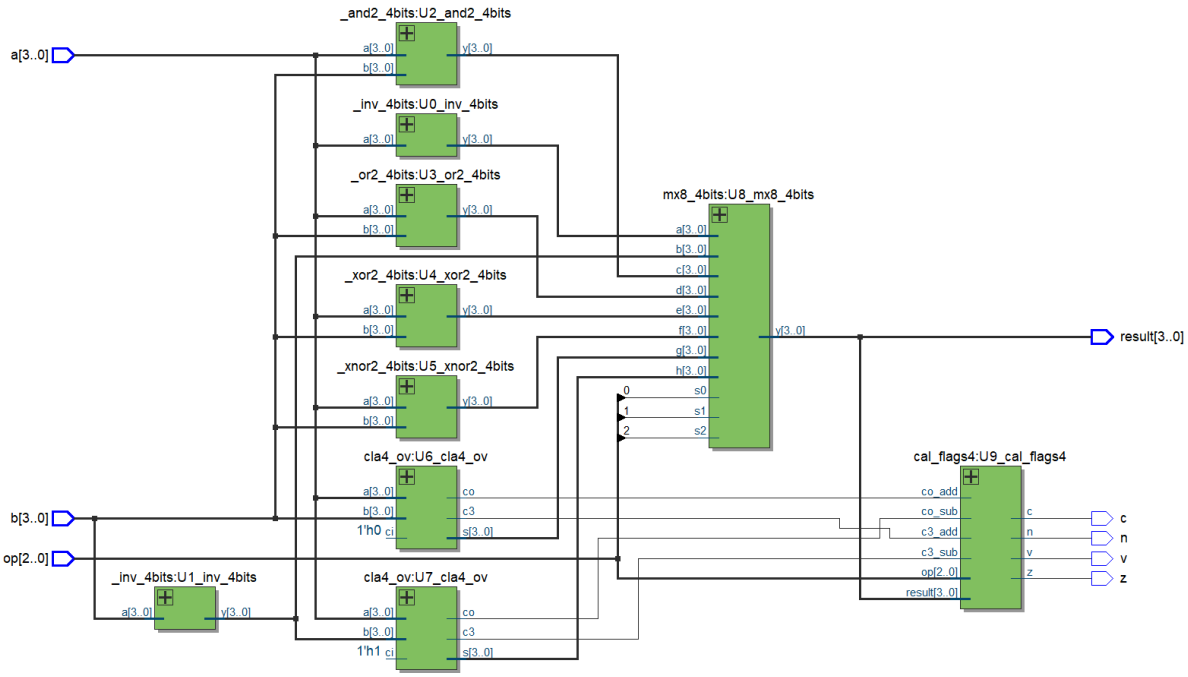
tb_op	0		1	2	3	4	5	6	7
tb_result	ffffffff	ffff0000	ff0000ff	00000100	1a1bbb00	1a1bba00	e5e445ff	058dcb02	00000000
tb_c	0							1	
tb_n	1			0			1	0	
tb_z	0								1
tb_v	0							1	0

32bit의 경우에도 연산결과가 op입력값에 따라 적절하게 출력된 것을 확인할 수 있다.

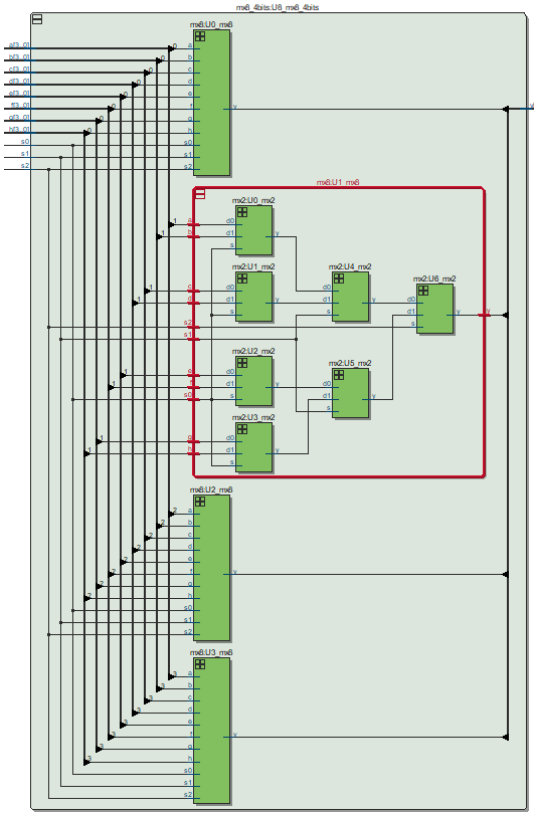
B. 합성(synthesis) 결과.

1. 4bit ALU

4bits ALU의 RTL Viewer는 다음과 같이 나타났다.



a와 b의 입력값이 8종류의 4bits gate를 거쳐 연산이 되고, 그 결과 값이 하나의 8 to 1 mux module의 입력으로 들어가 op값에 따라 선택되어 result로 출력되는 것을 볼 수 있다. flag를 계산하는 모듈은 add를 계산하는 adder와 sub를 계산하는 adder에서 carry와 carry_prev를 받고, 8 to 1 mux의 출력 result를 받아 flag를 계산한다.



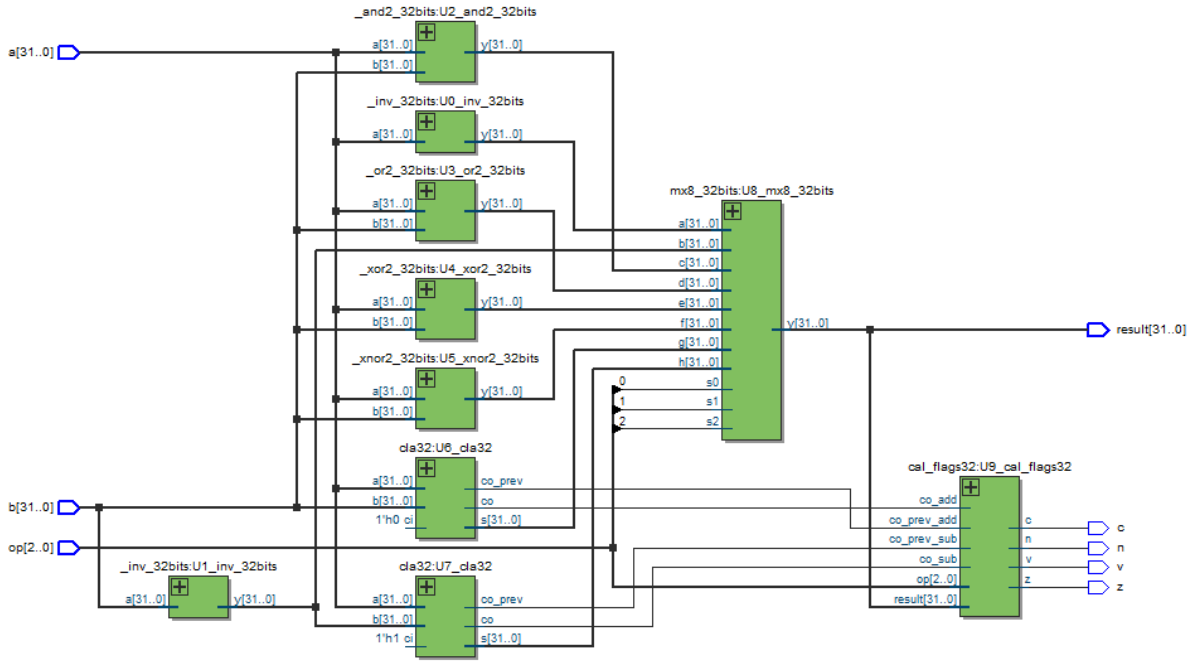
4 bits 8 to 1 mux의 내부에는 4 개의 1 bit 8 to 1 mux module이 있고, 각각의 mux module은 여러 개의 2 to 1 mux module로 이루어져 있음을 알 수 있다.

Flow Summary	
Flow Status	Successful - Thu Oct 01 15:04:39 2020
Quartus Prime Version	15.1.0 Build 185 10/21/2015 SJ Lite Edition
Revision Name	alu4
Top-level Entity Name	alu4
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	12 / 41,910 (< 1 %)
Total registers	0
Total pins	19 / 499 (4 %)
Total virtual pins	0
Total block memory bits	0 / 5,662,720 (0 %)
Total DSP Blocks	0 / 112 (0 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	0 / 4 (0 %)

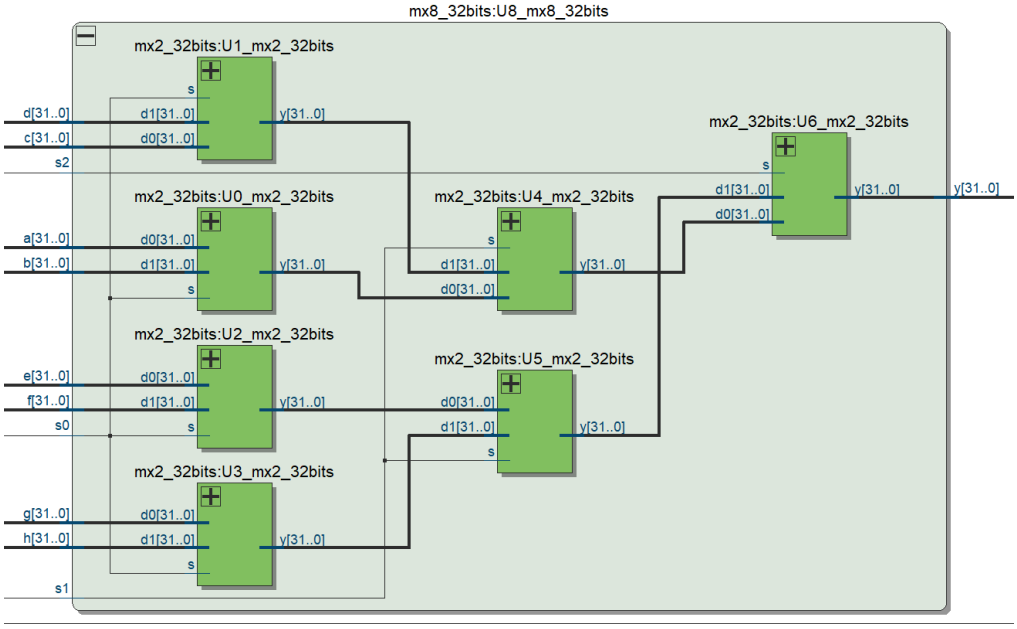
Flow Summary는 위와 같다. input a,b에 각각 4개, op에 3개의 pin이 사용되었고, output에는 result에 4개, c,n,v,z flag에 하나씩 사용되어 총 19개의 pin이 사용되었다.

2. 32bit ALU

32bits ALU의 RTL Viewer는 다음과 같이 나타났다. 4bits ALU과 같은 구조를 갖는다.



8 to 1 mux의 구조는 4bits ALU에서 사용한 것과 다르게 각 bit가 다른 MUX를 이용하지 않는다. 32개의 모든 bit가 지정된 mux를 이용하는 것을 확인할 수 있다.



Flow Summary	
Flow Status	Successful - Thu Oct 01 14:44:24 2020
Quartus Prime Version	15.1.0 Build 185 10/21/2015 SJ Lite Edition
Revision Name	alu32
Top-level Entity Name	alu32
Family	Cyclone V
Device	5CSXFC6D6F31C6
Timing Models	Final
Logic utilization (in ALMs)	126 / 41,910 (< 1 %)
Total registers	0
Total pins	103 / 499 (21 %)
Total virtual pins	0
Total block memory bits	0 / 5,662,720 (0 %)
Total DSP Blocks	0 / 112 (0 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 15 (0 %)
Total DLLs	0 / 4 (0 %)

Flow Summary는 위와 같다. input a,b에 각각 32개, op에 3개의 pin이 사용되었고, output인 result에 32개, c,n,v,z flag에 하나씩 사용되어 총 103개의 pin이 사용되었다.

5. 고찰 및 결론

A. 고찰

v flag와 c flag를 혼동하여 ALU 설계에 어려움을 가졌었다. 직접 예를 들며 v flag와 c flag가 표시되거나 되지 않는 경우의 수를 찾아보며 둘의 차이점을 알아내었다. 디지털 논리 회로 과목만이 아니라 어셈블리프로그래밍 과목에서도 접해본 flag인데, 이에 대해 더욱 깊이 이해하게 되었다. 수업시간에 배웠던 blocking, non-blocking assignment를 RTL viewer의 회로를 통해 확인하는 것이 개념을 이해하는 것에 큰 도움이 되었다.

B. 결론

이번 실습을 통해 Arithmetic Logic Unit의 의미와 역할, 구조를 알게 되었다. 다양한 종류의 module들을 각기 다른 level에서 활용하여 사용하는 과정이 조금 복잡하게 느껴지기도 했다. 회로의 전체적인 설계를 마친 후 직접 코드를 작성하는 과정에서 다시 설계도를 찾아보고, 컴파일 과정에서 오류가 발생한 것을 수정하며 더욱 많이 배울 수 있었던 것 같다. 완성한 회로에 대해 적절한 testbench를 작성하는 것이 어려웠다. 무작위로 숫자를 선택할 경우 확인되지 않는 경우가 존재했기 때문이다. 여러 개의 testcase를 작성하며 적절한 경우를 선택하는 방법으로 testbench를 작성하였다.

C. 하나의 adder를 이용하여 addition과 subtraction을 하는 방법

이번 실습에서 구현한 ALU에서는 addition와 subtraction을 모두 adder를 이용하여 계산하였다. 그러나 실제로 사용한 instance는 별개의 instance였다. addition을 수행하는 경우 ci에 0을, subtraction을 수행하는 경우 1을 넣어서 사용하였다. 하나의 adder로 두 가지 연산을 수행하려면 연산의 종류를 의미하는 op에 따라 변화하도록 만들어야 한다. add의 opcode는 110이고, sub의 opcode는 111이므로, 입력해야 할 ci의 값은 opcode의 마지막 bit와 같다. adder에 입력될 b는 2 to 1 mux를 이용해서 add연산일 경우 b를, sub일 경우 ~b를 adder에 연결하도록 설계하면 하나의 adder만을 이용하여 addition과 subtraction을 구할 수 있을 것이다.

6. 참고문헌

이준환교수님, 디지털논리회로2 강의자료, 광운대학교 컴퓨터정보공학과,2020

이준환교수님, 컴퓨터공학기초실험2 강의자료, 광운대학교 컴퓨터정보공학과,2020