

컴퓨터 공학 기초 실험2 보고서

실험제목: Subtractor & Arithmetic Logic Unit

실험일자: 2018년 09월 12일 (수)

제출일자: 2018년 09월 18일 (화)

학 과: 컴퓨터공학과

담당교수: 이준환 교수님

실습분반: 수요일 0, 1, 2

학 번: 2015722025

성 명: 정 용 훈

1. 제목 및 목적

A. 제목

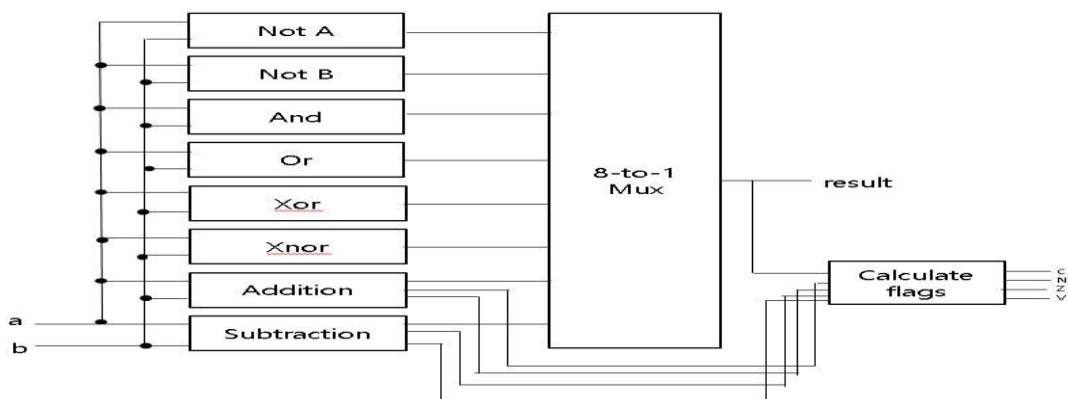
Subtractor & Arithmetic Logic Unit

B. 목적

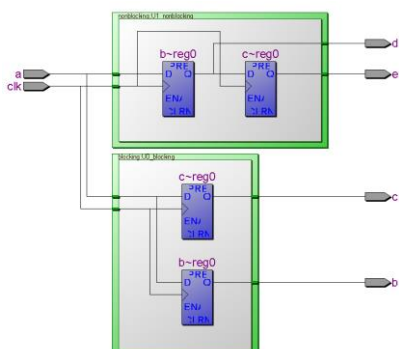
산술 논리 연산 장치(arithmetic logic unit; ALU)는 산술연산, 논리연산 및 Shift를 수행할 수 있는 중앙처리장치 내부의 회로 장치이다. 이번 실험과 과제를 통하여 4-bits ALU와 32-bits ALU를 구현하고 이번 실습을 통해 구현될 4개의 flag에 대해 이해하고 동작을 이해한다.

2. 원리(배경지식)

이번과제인 ALU를 구현하기 위해서는 이번주에 구현하게 될 mux와 더불어 첫 주차부터 구현해왔던 하드웨어들을 모두 사용하게 된다. 이는 ALU가 op code에 따라 수행 가능한 여러 연산 중 한가지의 연산을 선택하여 실행하고 그에 따른 flag가 결정되어 값을 알려 주기 때문이다. 여러 연산 방법 중에 선택하는 역할을 mux가 해주고 여러 연산 중에는 덧셈과 뺄셈이 존재하므로 CLA가 사용된다. CLA를 사용함으로써 CLA의 최상위 두개의 bits를 이용하여 flag값의 overflow를 결정하는 기반이 된다. Flag가 판단 하는 연산으로는 C, N, Z, V 가 있는데 순서대로 C는 carry가 발생하는 경우를 뜻하고 N은 연산 결과의 sign bit가 1이면 실행되며 Z는 연산결과가 0인 경우를 뜻한다 마지막으로 V는 연산결과가 over flow가 발생하면 실행된다. 여기서 over flow와 carry의 차이는 연산 결과자체를 보면 쉽게 이해 할 수 있다. Carry와 over flow는 이번 실습에서 덧셈이나 뺄셈을 통하여 발생하게 된다 tow's complement 체계를 따라 계산하였 때 결과 값이 sign bit의 영향을 받아서 제대로 나오지 않을 경우(ex)양수+양수=음수를 over flow라고 정의하며, carry out은 Co의 값이 올라가며 sign bit의 영향으로 값이 제대로 나오는 경우이다. 직접 계산하고 wave form을 참고하면 쉽게 이해 할 수 있는 원리이다. 아래는 우리가 이번에 구현하게 될 ALU의 이미지이다.



아래는 blocking과 non-blocking의 관련된 설명이다. Blocking은 부호로 '=' 을 사용하며 non-blocking은 '<=' 을 사용한다. 각 문법의 의미를 쉽게 말로 해석하면 blocking은 값의 처리가 한번에 이루어지며 non-blocking의 경우는 값의 처리가 순차적으로 처리 된다고 하면 이해하기 쉽다. 아래의 이미지는 주어진 코드를 컴파일 하였을 때 RTL Viewer를



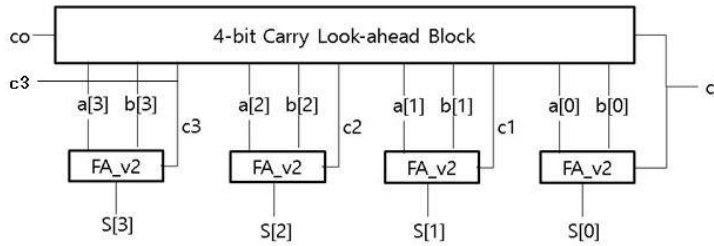
갈무리하면 나타낼 수 있는 이미지이다. 이미지를 보았을 때 blocking은 rising edge가 한번 일어나게 될 때 output의 값이 한번에 처리되는 것을 볼 수 있고, non-blocking의 경우 값의 처리가 순차적으로 일어날 수 있도록 설계되어 있는 것을 볼 수 있다.

쉽게 말해 주어진 코드를 예로 들면 아래와 같은 blocking 코드의 경우 결과 값으로 $b=a$, $c=a$ 의 값을 가질 수 있다. 그에 비해 non-blocking의 결과 값은 $b=a$, $c=b$ 의 값을 가지게 된다.

```
→always@(posedge clk)
→begin
→b=a;
→c=b;
→end
→endmodule
```

```
→always@(posedge clk)
→begin
→b<=a;
→c<=b;
→end
→endmodule
```

3. 설계 세부사항



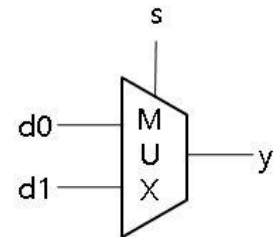
ALU는 1주차부터 지금까지 설계한 하드웨어와 더불어 mux를 구현하고 각 연산에 맞는 flag에 관련된 하드웨어를 구현하여 연결해주면 됩니다. 첫 번째로 flag의

overflow를 검출하기 위하여 먼저 설계한 CLA의 최상위 bits 두개 Co와 C3를 이용하여 overflow를 검출한다. 쉽게 아래 표와 같이 정리 할 수 있다.

Co	C3	overflow
0	0	0
0	1	1
1	0	1
1	1	0

즉 식으로 표현 하면 $v = Co \oplus C3$ 로 표현 할 수 있다.

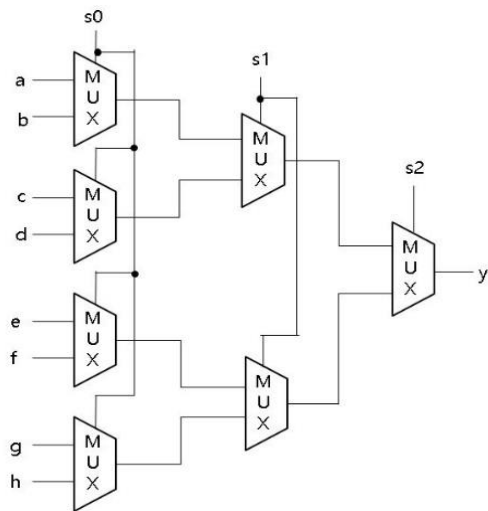
그 다음으로는 Mux를 설계하게 된다. Mux의 역할은 input의 값을 여러가지 넣었을 때 그 중 한 개를 뽑아서 출력하는 역할을 해준다. 즉 ALU는 여러가지 연산이 복합적으로 있는 하드웨어 이므로 여러 연산 중 원하는 연산을 선택하기 위하여 설계하는 것이 목적이다. 우선 2-to-1 Mux의 진리표를 보면 아래표와 같다.



Input			Output
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

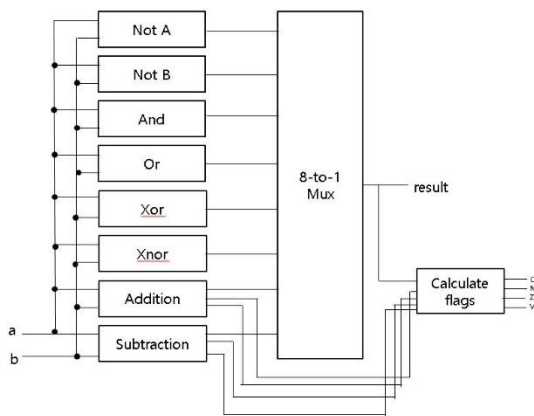
2-to-1 Mux는 s가 0일 때 d0의 값을 출력하게 되고, s가 1일 때, d1의 값을 출력하게 만들면 된다. 이번 설계에서 사용하게 될 Mux는 8-to-1 Mux로 2-to-1 Mux를 이용하여 4-2-1구조 즉 토너먼트식으로 연결하여 만들면 된다. 8-to-1의 Mux를 만드는 이유는 설계하게 될 ALU의 연산이 8개이기 때문이다. 또한 설계한 Mux에게 어떠한 연산을 해

야 할지 알려주기 위한 code가 있는데 이것을 Opcode라 칭한다. 8-to-1 Mux와 연산에 관련된 Op코드는 아래 그림과 표처럼 정리 할 수 있다.



Opcode			operation
S2	S1	S0	
0	0	0	Not A
0	0	1	Not B
0	1	0	And
0	1	1	Or
1	0	0	Exclusive Or(XOR)
1	0	1	Exclusive Nor(XNOR)
1	1	0	Addition
1	1	1	Subtraction

표에서 보는것과 같이 8개의 연산을 구분 할 수 있다. 결과적으로 ALU의 구조부터 보자면 아래와 같은 형태를 하고있다.



연산의 종류를 보면 A의 부정, B의 부정, And연산, Or연산, XOR연산, XNOR연산, 더하기, 빼기 의 연산이 있다. 이제 Flag와 관련된 연산이 시작된다. 8가지의 연산을 통하여 나온 결과를 가지 Flag가 연산을 하여 4가지의 상태를 구분하여 알려준다 4가지 상태로는 C, N, Z, V가 있고 의미하는 바로는 C는 carry, N은 sign bit의 판단, Z는 연산결과값이 0인지 판단, V는 연산의 결과에서 overflow가 일어났

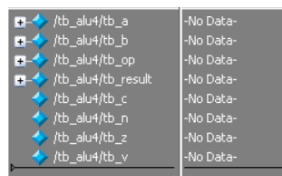
는지를 판단해 준다. 위와 같은 연산을 해주는 하드웨어를 Calculate flags라고 부르게 된다. 결과적으로 ALU의 output의 종류는 연산 결과인 result와 연산 결과에 대한 판단을 해주는 C, N, Z, V가 있다는 것으로 보면 이해하기 쉽다.

4. 설계 검증 및 실험 결과

A. 시뮬레이션 결과

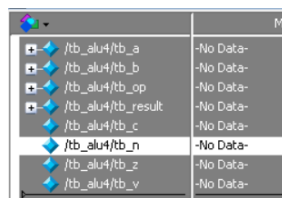
ALU4

제공된 코드

	<pre># view structure # .main_pane.structure.interior.cs.body.struct # view signals # .main_pane.objects.interior.cs.body.tree # run -all # 000 failed # 001 failed # 010 failed # 011 failed # 100 failed # 101 failed # 110 failed # 111 failed</pre>
---	---

a, b의 input 값은 같은 값으로 일정하게 들어가는 것을 볼 수 있다. 연산을 결정하는 op code가 계속 바뀌는데 연산이 바뀌므로 결과 값도 바뀌는 것을 함께 볼 수 있다. 연산 결과에 따라 flag에서 상태를 점검해주고 상태에 맞게 output이 나오는 것을 확인 할 수 있다. 두번째 연산에 관련하여 설명하면 op code 가 001이다. Op code가 001이면 Not B연산을 하게 되는데 이에 input 0110 인 b의 값이 1001로 바뀌어 결과가 나오는 것을 확인 할 수 있고 연산결과에 대하여 sign bit가 1 이므로 flag의 n이 1로 올라가는 것을 확인 할 수 있다. 이와 마찬가지로 나머지 연산들도 op code와 비교하여 연산의 결과에 따라 flag의 값이 바뀌는 것을 확인 할 수 있다. 또한 self-checking을 사용하였을 때 예상 값과 다르기 때문에 failed가 뜨는 것을 확인 할 수 있다.

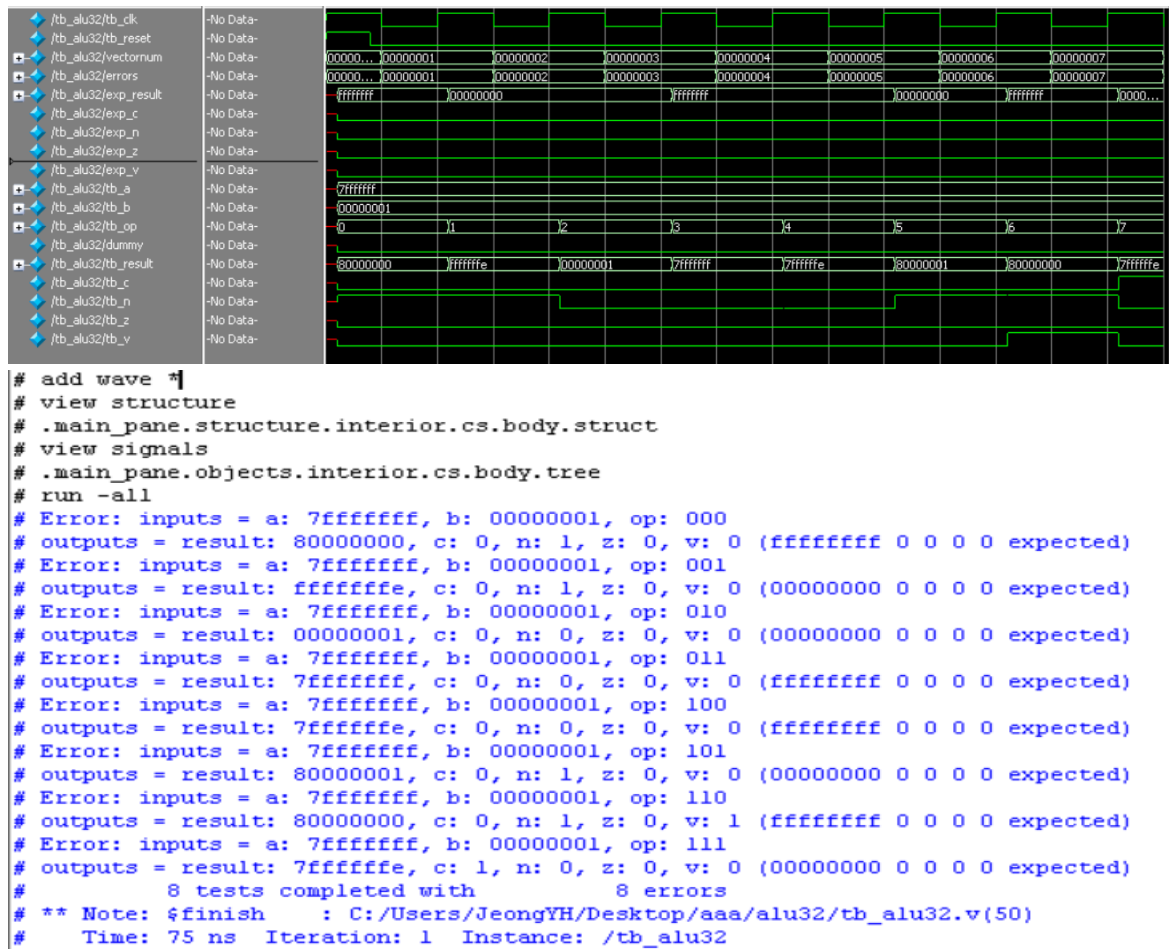
Ppt 코드

	<pre># view structure # .main_pane.structure.interior.cs.body.struct # view signals # .main_pane.objects.interior.cs.body.tree # run -all # Break in Module tb_alu4 at C:/Users/JeongYH/Desktop/aaa/alu4/tb_alu4.v line 31 # Simulation Breakpoint: Break in Module tb_alu4 at C:/Users/JeongYH/Desktop/aaa/alu4/tb_alu4.v line 31 # MACRO ./alu4_run_msim_rtl_verilog.do PAUSED at line 25</pre>
---	---

이 wave form 또한 4-bits ALU에 관련된 것이다 연산 결과 중 한가지를 예로 들어보겠다. 마지막 연산인 op code 101 은 Xnor의 연산을 하며 두 input의 bit를 대칭 시켜 비교 하였을 때 비교한 두 input이 같으면 1의 값을 다르면 0의 값을 반환한다 그리하여 wave form에서 보는 것과 같이 0011, 0101을 input으로 넣은 결과 1001이라는 결과 값이 나오게 된다. 연산과정에서 sign bit가 1 이 되었으므로 flag값은 n을 1로 반환하게 된다. 또한 self-checking을 하였을 때 예상 값과 결과 값이 같으므로 위처럼 failed이 뜨지 않는 것을 볼 수 있다.

ALU32

제공된 코드

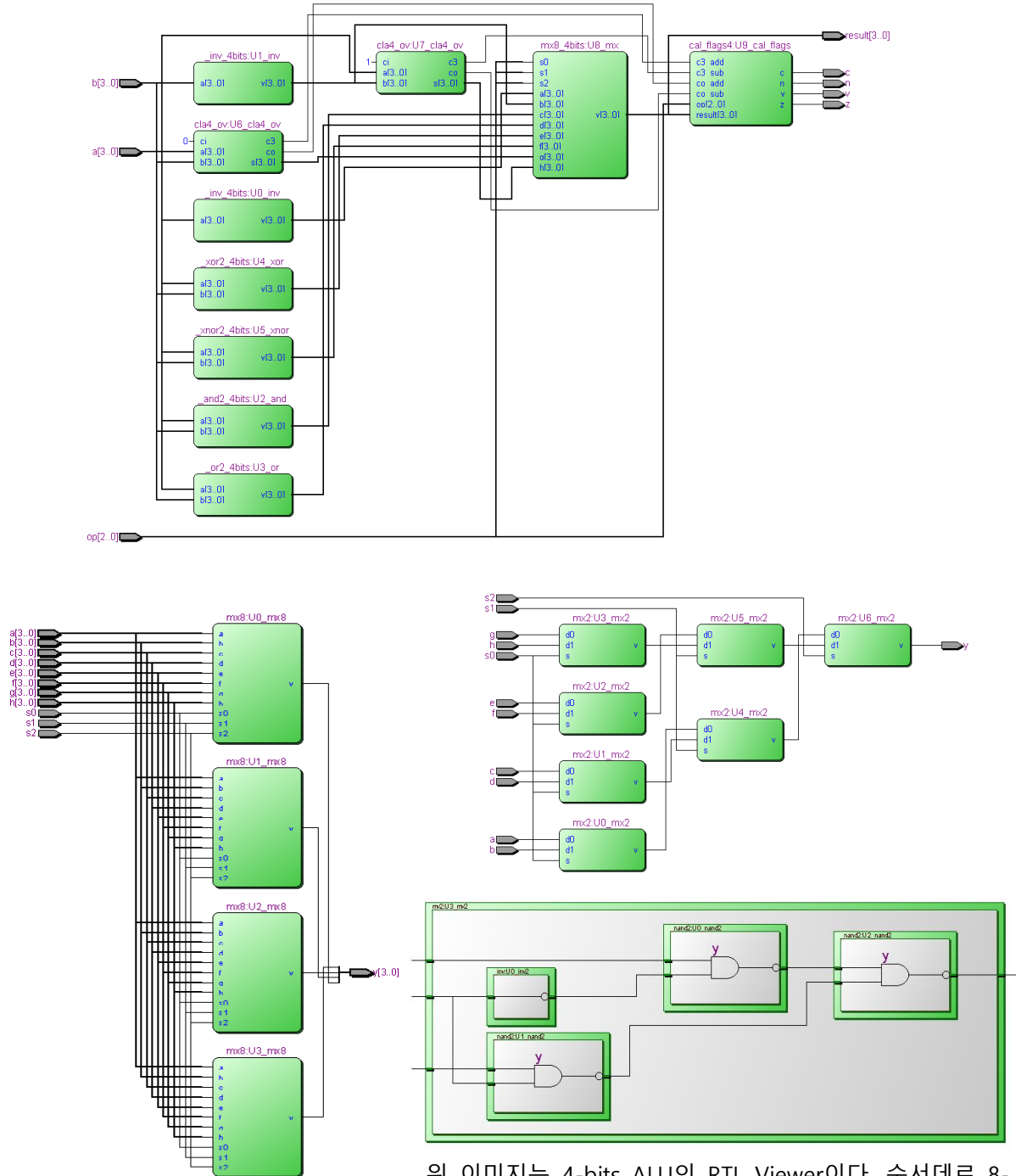


Waveform의 결과를 보면 vectornum이 하나씩 증가 하는 것을 볼 수 있고 clock이 rising일 때 마다 일정한 값이 들어가고 결과가 바뀌는 것을 볼 수 있다. 그에 따라 결과 값에 관련하여 flag의 값이 변하는 것을 아래에서 확인 할 수 있다. Clock이 들어간 것과 bits수가 차이 나는 것 이외에는 alu4의 동작과 유사하다고 보면 쉽다. 위에 32-bits는 self-checking 방법을 사용하지 않고 test vector를 사용하여 검증한 결과이다. 예상 값과 다르기 때문에 에러가 나는 것을 확인 할 수 있다.

B. 합성(synthesis) 결과

4-bits ALU

RTL Viewer



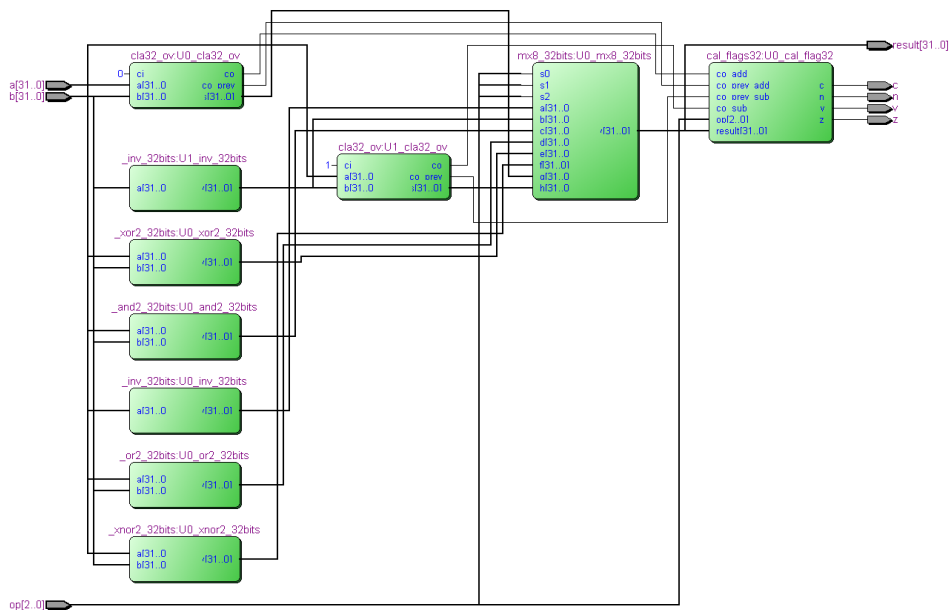
위 이미지는 4-bits ALU의 RTL Viewer이다. 순서대로 8-to-1 mux의 모습이 보이고 이어서 2-to-1 mux의 내부까지 볼 수 있다. 특히 연산이 CLA를 거치게 되면 flag에서 추가적으로 인풋 값을 받아 flag의 값을 결정하는 것을 볼 수 있다.

Flow summary

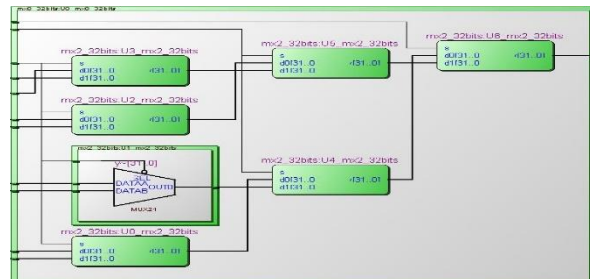
Flow Summary	
Flow Status	Successful - Sat Sep 15 16:08:40 2018
Quartus II 64-Bit Version	13.0.0 Build 156 04/24/2013 SJ Web Edition
Revision Name	alu4
Top-level Entity Name	alu4
Family	Cyclone II
Device	EP2C70F896C6
Timing Models	Final
Total logic elements	42 / 68,416 (< 1 %)
Total combinational functions	42 / 68,416 (< 1 %)
Dedicated logic registers	0 / 68,416 (0 %)
Total registers	0
Total pins	19 / 622 (3 %)
Total virtual pins	0
Total memory bits	0 / 1,152,000 (0 %)
Embedded Multiplier 9-bit elements	0 / 300 (0 %)
Total PLLs	0 / 4 (0 %)

32-bits ALU

RTL Viewer



RTL Viewer를 확인해보면 이전에 설계한 4-bits ALU와 큰 차이가 없는 것을 확인 할 수 있다. 정말 단순하게 생각하며 비트수의 차이만 있고 동작 자체는 동일 하다고 생각하면 된다.



Flow summary

Flow Summary	
Flow Status	Successful - Sat Sep 15 16:21:18 2018
Quartus II 64-Bit Version	13.0.0 Build 156 04/24/2013 SJ Web Edition
Revision Name	alu32
Top-level Entity Name	alu32
Family	Cyclone II
Device	EP2C70F896C6
Timing Models	Final
Total logic elements	266 / 68,416 (< 1 %)
Total combinational functions	266 / 68,416 (< 1 %)
Dedicated logic registers	0 / 68,416 (0 %)
Total registers	0
Total pins	103 / 622 (17 %)
Total virtual pins	0
Total memory bits	0 / 1,152,000 (0 %)
Embedded Multiplier 9-bit elements	0 / 300 (0 %)
Total PLLs	0 / 4 (0 %)

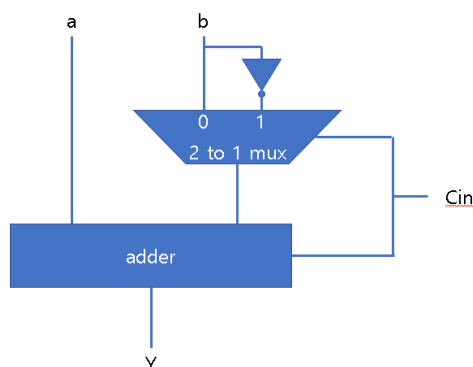
5. 고찰 및 결론

A. 고찰

제공된 test bench가 wave form을 통하여 봤을 때 동작을 하지 않는 문제가 있었습니다. 이유로는 함께 제공 되었던 example.tv 파일을 project에 넣어두지 않아서 문제가 발생하게 되었습니다. 이는 프로젝트에 제공된 파일 넣으면서 해결 하였습니다. 또한 비슷한 방식으로 input의 값이 다른 test bench를 작성하고 같은 방식으로 프로그램을 실행 했지만 wave form에서의 결과 값이 제대로 나오지 않는 것을 확인 했습니다. 그 이유로는 example.tv 파일의 경로를 설정을 해줘야 하는데 설정하지 않은 것이 문제 였습니다. 해결 방법으로는 미리 제공된 코드와 비교를 해보면서 다른 부분을 살펴보고 경로를 재 설정해주면서 문제를 해결하게 되었습니다.

B. 결론

이번 과제를 통해서 기본적인 ALU의 동작이 어떻게 이루어지는지 좀 더 자세하게 알게 되었습니다. 특히 미리 설계하였던 CLA의 최상위 비트를 이용하여 over flow를 검출하는 방법을 새롭게 알게 되었습니다. 설계한 ALU에서는 adder를 두개 사용하여 덧셈과 뺄셈을 따로 하였습니다. 이를 좀더 효율적으로 사용하기 위해서는 adder를 하나만 사용하여 두개의 연산을 할 수 있는 방법을 찾아 보았고 ppt를 참고해서 보았을 때 Cin의 값이 0 일때는 addition에 관련된 연산을 하였고 1일 경우에는 Subtraction에 관련되어 연산하는 것을 보았습니다. 이는 우리가 설계한 2-to-1 mux를 사용하여 Cin의 값을 선택 할 수 있도록 하고 Cin의 값에 따라 알맞은 연산을 할 수 있도록 설계하면 된다는 결론을 도출할 수 있었습니다.



6. 참고문헌

ALU의 대한 개념 / <https://ko.wikipedia.org/wiki/ALU>

김영민 / Subtractor and ALU / 광운대학교 / 2018

blocking 과 non-blocking / <https://blog.naver.com/zzbksk/220885007983>