# 컴퓨터 공학 기초 실험2 보고서

## 실험제목: Subtractor & Arithmetic Logic Unit

실험일자: 2022년 10월 5일 (수)

제출일자: 2022년 10월 11일 (화)

학 과:컴퓨터정보공학부

담당교수: 공진흥 교수님

실습분반: 수요일 0, 1, 2

학 번: 2019202021

성 명: 정 성 엽

# 1. 제목 및 목적

## A. 제목

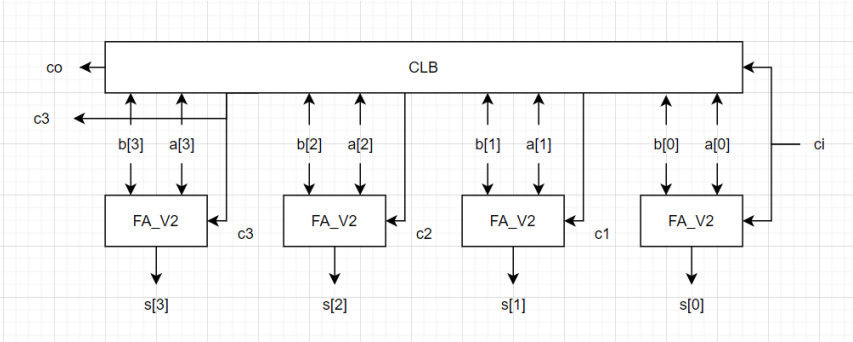
Subtractor & Arithmetic Logic Unit

## B. 목적

2의 보수 표현을 이용해서 이전에 만들었던 CLA를 활용하고 8 to 1 mux의 원리를 이해하고 ALU에 들어갈 연산을 미리 만들어 Verilog를 통해 구현한다. 계산 결과르 이용해 flags를 계산하도록 Cal_flag 모듈을 제작하여 flag 역할을 하도록 한다.

# 2. 원리(배경지식)

## A. cla4_ov




이전에 만들어보았던 CLA4 모듈과 다르게 이번 실험에서 만든 CLA4_ov 모듈은 Overflow를 검출하기 위해 상위 2개의 Carry를 출력하여 output으로 가져오도록 한다.

## B. Flags

### i. Flags

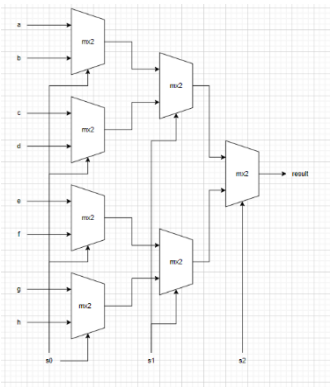
- N(negative): 음수를 표시하는 flag로 음수면 1로 나타낸다.
- Z(zero) : 만약 모든 bit가 0이면 1로 나타낸다.
- C(carry): Carry out이 발생한 경우 1을 나타내며 보통 adder에서 carry out으로 구분한다.
- V(overflow): overflow가 발생하면 1로 나타낸다.

### ii. Carry Flag와 Overflow Flag

- Carry Flag는 Unsigned 연산 또는 signed 연산에서 모두 가능하고 발생하며, 다만 Unsigned 연산에서 나온 값은 MSB가 부호표현을 하지 않기 때문에 Carry out 연산값을 가지고 더 큰 범위의 숫자를 표현할 수 있다.
- Overflow는 unsigned가 아닌 signed값으르 계산 할 때만 발생한다. MSB가 부호표현을 하므로 2의 보수로 표현된 수를 연산한다면 overflow 발생시 제일 좌측 자릿수가 인식할 수 없다. 예를 들어 3bits의 $111_2$과 $101_2$을 덧셈 연산을 할 때 $1100_2$이 되는 데 unsigned는 상관 없지만 signed에서는 (-1) +(-3) = -4가 아닌 0의 값이 계산된다 이는 3bit 연산

에서 4bit의 결과 값이 나오면서 제일 왼측의 1을 인식 못하는 것이다. 위와 같은 오류를 overflow 발생을 뜻하며 이는 signed 값을 표현할 때 중요하다. 이번 실험에서 CLA에서 co값만 가져오는 것이 아닌 c3를 가져오는 이유도 이 둘이 xor일 때 signed 2의 보수에서는 다른 부호가 나오므로 overflow로 판별한다.

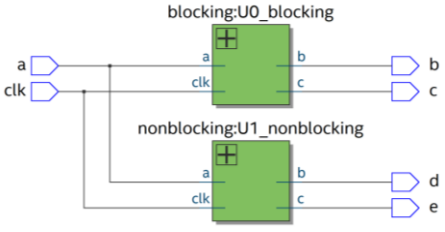
C. 8-to-1 MUX

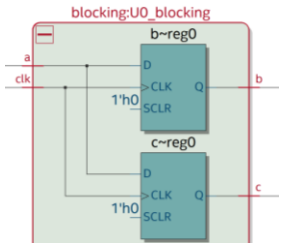


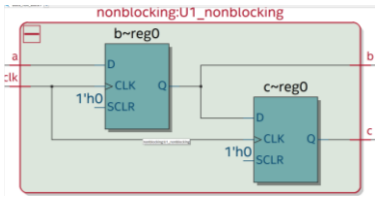
| opcode (s2,s1,s0) | | | result |
|---|---|---|---|
| 0 | 0 | 0 | a |
| 0 | 0 | 1 | b |
| 0 | 1 | 0 | c |
| 0 | 1 | 1 | d |
| 1 | 0 | 0 | e |
| 1 | 0 | 1 | f |
| 1 | 1 | 0 | g |
| 1 | 1 | 1 | h |

Opcode에 따라 result의 결과가 다르게 나오는데 opcode의 조합이 8개이고 결과 또한 8개이다. 기존 2-to-1 MUX를 7개 사용하여 8-to-1 MUX를 구현할 수 있다.

D. Blocking과 non blocking의 차이




i. Blocking

'='기호를 사용하여 값을 대입하는데 이때 '=' 기호는 해당 연산이 모두 끝난 이후 작동한다. 그래서 결국 posedge에 a=b=c가 된다. 따라서 2개의 D-flipflop에 a값만 입력된다.

ii. non-blocking

'<=' 기호를 사용하여 값을 대입하는데 이때 해당 연산을 다음 연산과 병렬적으

로 수행하여 posedge에는 b=a, c=b가 된다. 따라서 2개의 D-flipflop에 a, b의 값이 각각 입력된다.

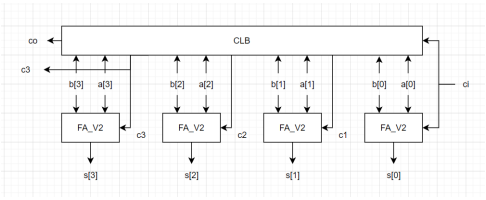
## 3. 설계 세부사항

### A. 8-to-1 MUX

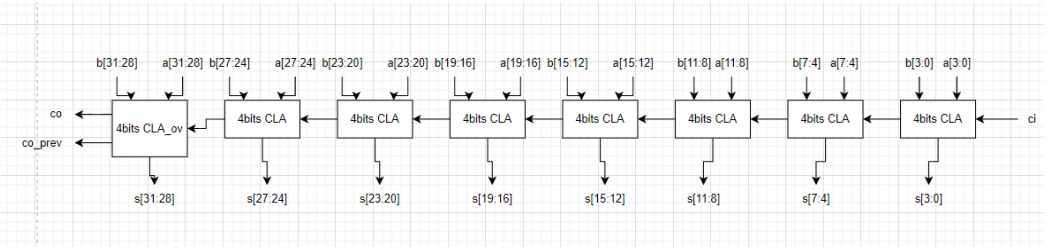
OPCODE에 따라 mux의 동작이 다르게 작동한다. 동작의 종류와 해당 OPCODE는 아래 표와 같다.

| Opcode | Operation |
|---|---|
| 000 | not a |
| 001 | not b |
| 010 | and |
| 011 | or |
| 100 | xor |
| 101 | xnor |
| 110 | add |
| 111 | sub |

### B. 4bits CLA & 32bits CLA

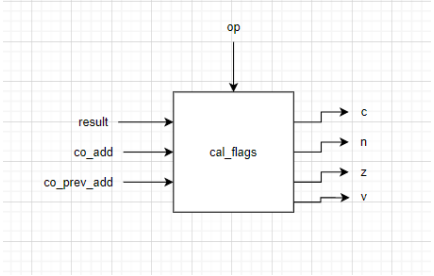



원리 때 설명했던 4bits CLA를 구현하고 이를 마지막에 사용하여 32bits CLA를 구현한다.




위 회로처럼 마지막에 4bits CLA_ov를 사용하여 co와 co_prev를 출력할 수 있도록 한다.

### C. calculate flags




Opcode, result, co_add, co_prev_sub를 모두 입력받고 해당 값에 따라 c,n,z,v를 세팅한다.

i. C(Carry)

- 만약 opcode 왼쪽 두자리가 11이라면 co_add 값으로 세팅하고 아니라면 0으로 세팅한다.

ii. N(Negative)

- MSB 비트가 1이라면 N 값을 1로 세팅하고 아니라면 0으로 세팅한다.

iii. Z(Zero)

- Result의 모든 비트가 0이면 Z를 1로 세팅하고 아니라면 0으로 세팅한다.

iv. V(Overflow)

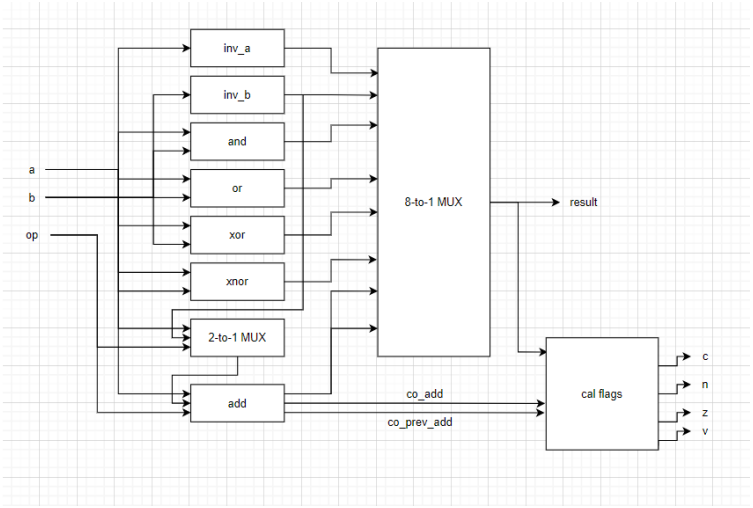
- 먼저 Opcode의 왼쪽 두자리가 11인지 확인하고, 맞다면 co_add^co_prev_add의 연산 값을 V에 세팅하고 아니라면 0을 세팅한다.

D. Arithmetic Logic Unit(ALU)

| Port | Name | Bandwidth(bits) |
|---|---|---|
| input | a | 4/32 |
| | b | 4/32 |
| | op | 3 |
| output | result | 4/32 |
| | c | 1 |
| | n | 1 |
| | v | 1 |
| | z | 1 |

| 구분 | 이름 |
|---|---|
| Top module | alu4 / alu32 |
| Sub module | _inv_4 / _inv_32 |
| | _and2_4 / _and2_32 |
| | _or2_4 / _or2_32 |
| | _xor2_4 / _xor2_32 |
| | _xnor2_4 / _xnor2_32 |
| | mx2_4 / mx2_32 |
| | cla4_ov / cla32_ov |
| | mx8_4 / mx8_32 |
| | cal_flags4 / cal_flags32 |

Arithmetic Logic Unit의 input, output 의 비트 크기는 필요에 따라 4bits 또는 32bits로 설정하고 Module 또한 4bits 또는 32bits 단위로 만들어 4bits ALU와 32bits ALU를 구현한다.



# 4. 설계 검증 및 실험 결과

## A. 시뮬레이션 결과

### i. 4-bits ALU

```
Transcript
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
# 14 tests completed with  0 errors
# ** Note: $stop    : C:/2-2/COMEX/week5/alu4/tb_alu4.v(51)
#    Time: 165 ns  Iteration: 1  Instance: /tb_alu4
# Break in Module tb_alu4 at C:/2-2/COMEX/week5/alu4/tb_alu4.v line 51
```

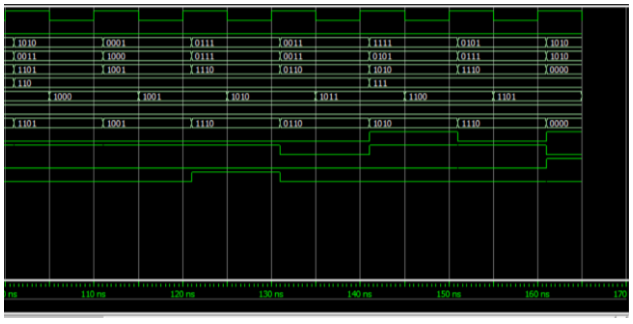
Vector 파일을 통해 testbench에 테스트하고 Self-checking을 통해 display 된 것이 없으므로 이상이 없다.

- Opcode 000부터 101까지

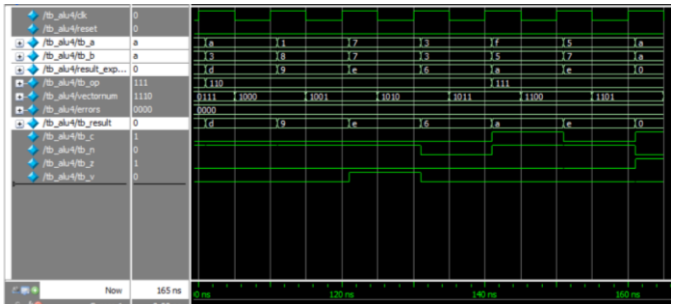
1) Opcode가 000일 때 not a 연산이 되어 0000이 1111로 1100이 0011로 된 것을 확인했다.
2) Opcode가 001일 때 not b 연산이 되어 0011이 1100이 된 것을 확인했다.
3) Opcode가 010일 때 and 연산이 되어 0101과 1001이 0001이 된 것을 확인했다.
4) Opcode가 011일 때 or 연산이 되어 0101과 1010이 1111이 된 것을 확인했다.
5) Opcode가 100일 때 xor 연산이 되어 0011과 0101이 0110이 된 것을 확인했다.
6) Opcode가 101일 때 xnor 연산이 되어 0011과 0101이 1001이 된 것을 확인했다.

- Opcode 110부터 111까지

1) binary display

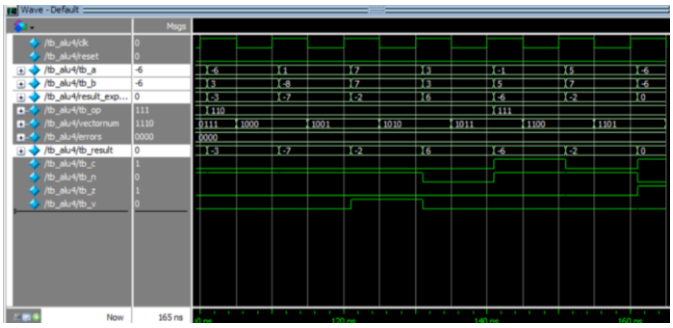


2) hexadecimal display



- 110연산이든 111연산이든 모두 add 연산이 되어 부호와 상관없이 더한 값이 binary display든 hexadecimal display에서 모두 더한 값으로 잘 출력됨을 확인했다.

3) decimal display

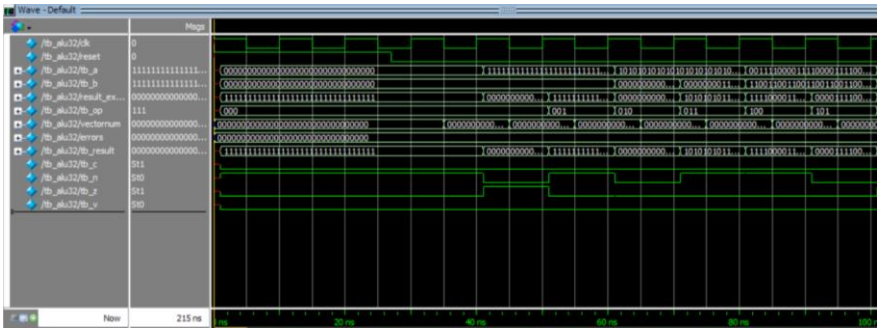


- hexadecimal display와 다르게 decimal display에서는 signed 연산으로 계산하여 양수의 연산과 음수의 연산 모두 잘 되었음을 확인할 수 있다.

ii. 32-bits ALU


Vector 파일을 통해 testbench에 테스트할 값을 넣고 Self-checking을 통해 결과가 이상 없이 display가 나타나지 않는 것을 확인했다.

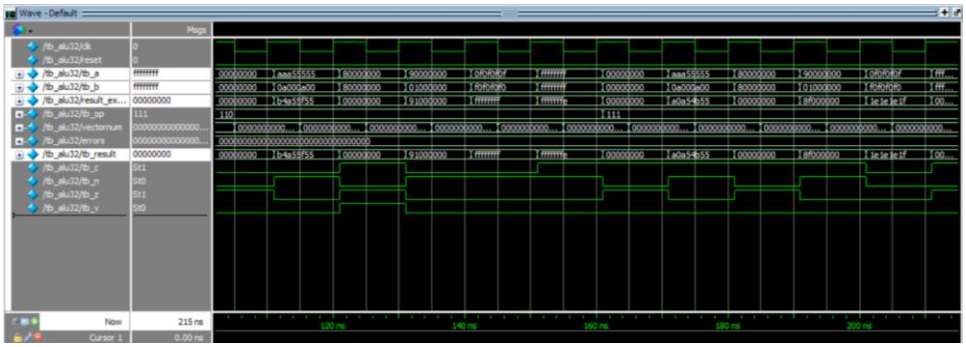
- Opcode 000부터 101까지



Inv, And, Or, Xor, Xnor 연산 모두 4bit Alu 연산 때와 마찬가지로 정상적으로 잘 진행되었고 예측한 값과 같음을 확인했다.

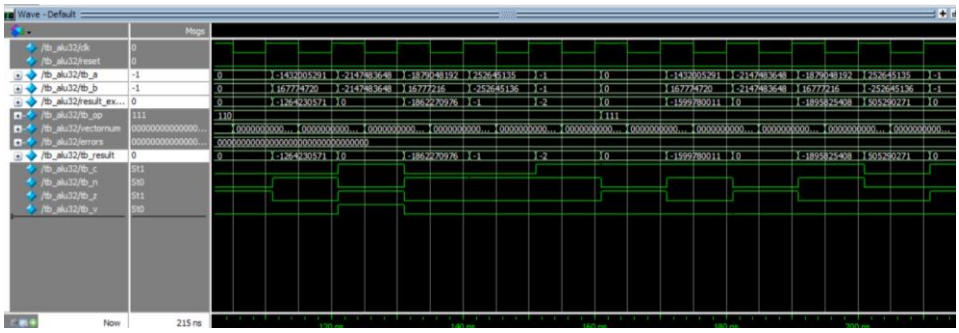
-Opcode 110부터 111까지

1) hexadecimal display



덧셈 연산과 뺄셈 연산은 hexadecimal display에서 확인했을 때 연산의 결과 값이 정상적으로 잘 나왔음을 확인했다.

2) decimal display

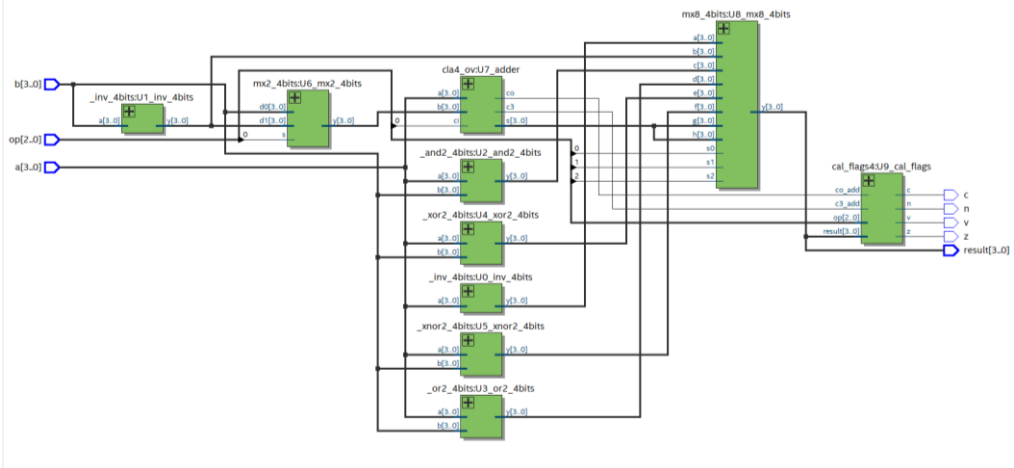


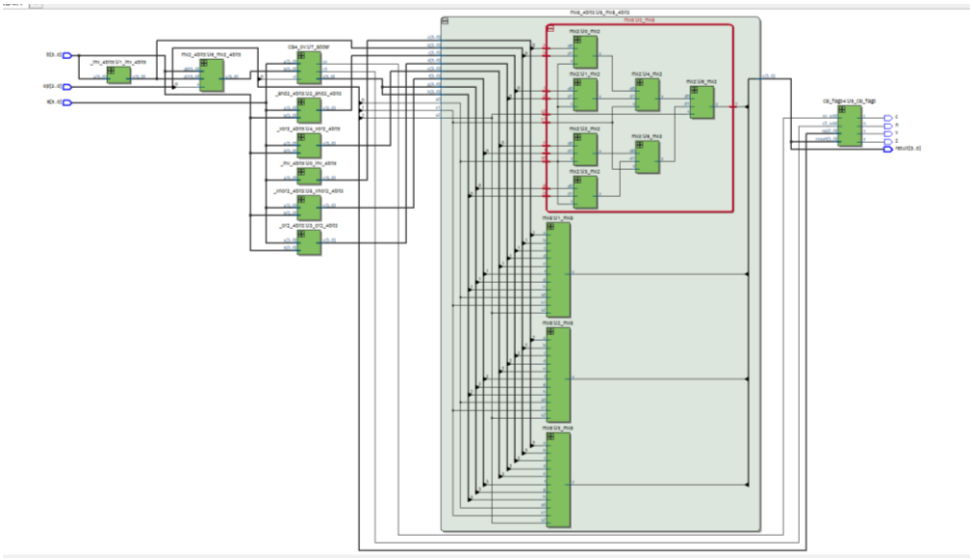
덧셈 연산과 뺄셈 연산을 decimal display에서 확인했으며 이때 또한 연산이 잘 이뤄졌음을 확인했으며 이 때는 overflow의 flag를 확인했을 때 정상적으로 overflow 발생시 set가 되었음을 확인하였다.

## B. 합성(synthesis) 결과

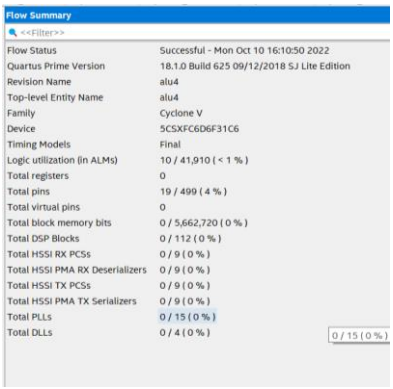
### i. 4-bits ALU

- RTL viewer





- Flow summary

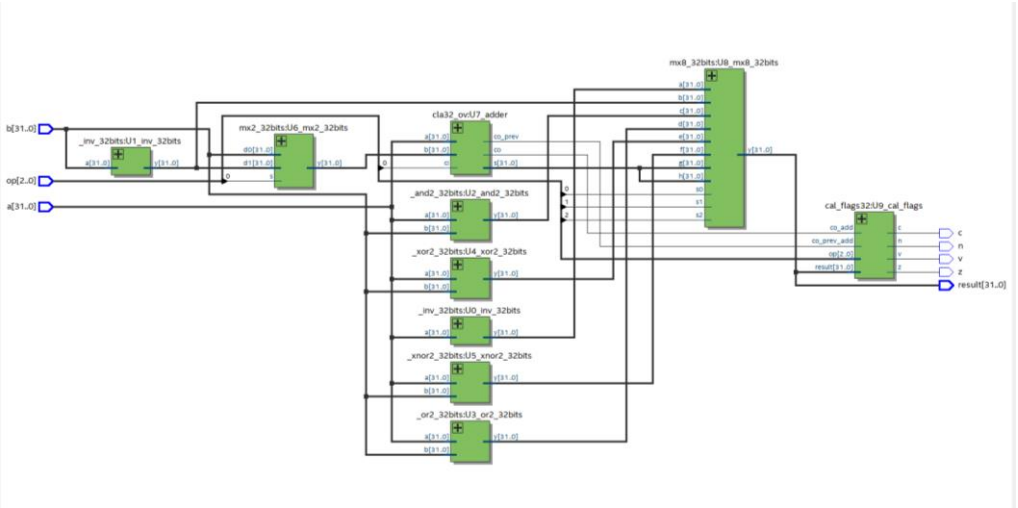


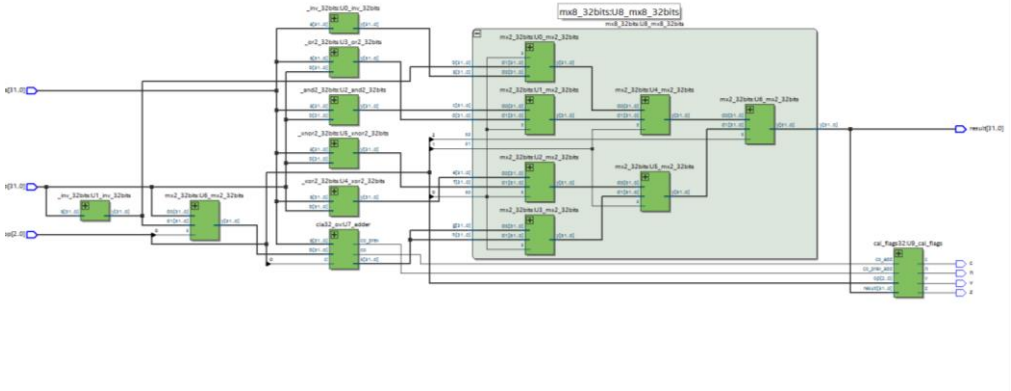
| Flow Summary | |
|---|---|
| Flow Status | Successful - Mon Oct 10 16:10:50 2022 |
| Quartus Prime Version | 18.1.0 Build 625 09/12/2018 SJ Lite Edition |
| Revision Name | alu4 |
| Top-level Entity Name | alu4 |
| Family | Cyclone V |
| Device | 5CSXFC6D6F31C6 |
| Timing Models | Final |
| Logic utilization (in ALMs) | 10 / 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 % ) |

Opcode에 따라 gate의 연산이 달라지고 add 또는 sub 연산을 하였을 때 그에 따른 flag를 계산하고 n, z, c, v로 표현하였다. 이때 mux를 직접 gate 단위로 구현하여 진행하였기에 내부에 gate들이 매우 많이 존재하는 것을 확인할 수 있다.

ii. 32-bits ALU

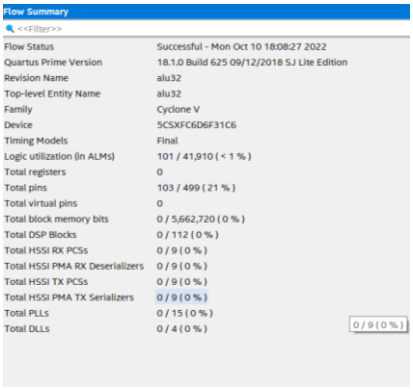
- RTL viewer






이전 4bits ALU와 달리 mx8이 매우 간소화 된 것을 볼 수 있는데 이는 구현할 때 Conditional Operator를 사용했기 때문이다.

- Flow summary



| Flow Summary | |
|---|---|
| Flow Status | Successful - Mon Oct 10 18:08:27 2022 |
| Quartus Prime Version | 18.1.0 Build 625 09/12/2018 SJ Lite Edition |
| Revision Name | alu32 |
| Top-level Entity Name | alu32 |
| Family | Cyclone V |
| Device | 5CSXFC6D6F31C6 |
| Timing Models | Final |
| Logic utilization (in ALMs) | 101 / 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 % ) |

실험 예시에서 나온 Flow summary 보다 더 적은 Logic Utilization을 사용한 것을 확인할 수 있는데, 예시보다 더 적게 사용한 것으로 보아 더 효율적으로 제작하였거나 예시가 더 많은 logic을 사용했음을 볼 수 있다.

## 5. 고찰 및 결론

### A. 고찰

지난번과 동일하게 wire을 설정할 때 w0를 wo로 적어서 컴파일 오류가 발생하는 일이 있었다. 또한 아직 C언어에 익숙해서 인지 module 부분이 함수라 생각하고 작성하면 ';' 을 빼먹는 일이 있다.

이번 실험은 전에 사용한 2-to-1 mux를 이용하여 8-to-1 mux를 구현하는 것으로 opcode에 따라 다른 연산이 이루어질 수 있도록 했으며 flags를 통해 연산 결과에 대한 분석이 가능하도록 했다. 이는 어셈블리 언어에서도 사용한다.

매번 각각의 경우를 미리 계산하여 모듈을 만드는 것이 힘든 과정이었으나 삼항연사자를 이용하여 2-to-1 mux를 구현하는 것은 매우 쉽고 빠르게 처리가 가능하게 되었다. 이

는 cal-flag 모듈에서 c, v를 구현할 때도 사용하여 11x의 opcode를 가진 경우와 아닌 경우를 나누었다.

디지털논리회로2 시간에 배우는 self-checking 기법은 미리 값을 계산하여 후에 testbench로 waveform과 수를 출력할 때 미리 계산한 값과 같은 지 비교하여 display한다. 또한 testvectors를 통해 값을 쉽게 넣을 수 있었다.

### B. 결론

이번 실험에서는 opcode에 따른 ALU module의 실행을 했으며 alu module에 출력되는 종류는 이번 실험과 동일할 수도 있고 후에 선택에 따라 다른 연산을 할 수 있도록 만들 수도 있다. 또한 adder에서 나온 sum의 결과를 2번 넣어서 ADD 연산 뿐만 아니라 SUB 연산도 만들 수 있음을 확인하였다.

또한 Adder에서 나온 co값과 co_prev 값을 통해 calculate flags에서 V값을 세팅할 수 있으며 삼항 연산자를 통해 if문을 만들어 조건문을 완성하였다.

마지막으로 testbench에서도 selfchecking을 진행하고 이를 실행할 때 testvectors를 이용한 기법을 적용하여 보다 많고 쉽게 그리고 결과값과 예상값을 비교 할 수 있는 testbench를 작성할 수 있었다.

## 6. 참고문헌

공영호 교수님/디지털논리회로2/광운대학교 컴퓨터정보공학부/2022

공진흥 교수님/컴퓨터공학기초실험2/광운대학교 컴퓨터정보공학부/2022