

컴퓨터 공학 기초 실험2 보고서

실험제목: Subtractor & Arithmetic Logic Unit(ALU)

실험일자: 2023년 09월 25일 (월)

제출일자: 2023년 10월 11일 (수)

학 과: 컴퓨터공학과

담당교수: 이준환 교수님

실습분반: 월요일 0, 1, 2

학 번: 20222020264

성 명: 최봉규

1. 제목 및 목적

A. 제목

Subtractor & Arithmetic Logic Unit (ALU)

B. 목적

two's complement를 이용하여 감산기를 완성한다. Arithmetic logic을 이해하여 8개의 opcode에 따른 연산 결과를 예상, 측정한다. Alu를 설계하여 n, z, c, v 값 flag를 이해하고 연산중 flag값이 바뀌는 것을 확인한다.

2. 원리(배경지식)

flag는 ALU 내에서 특정 조건이나 상황이 만족되었을 때, 이를 표시해주는 것이다. 이번 실습에서의 flag에서는 n, z, c, v 이 4개의 값을 갖는다. n은 negative, z는 zero, c는 carry, v는 overflow의 뜻을 가지고 있다. 각각은 다음의 상황에서 1의 값을 갖는다.

- N: 연산결과의 sign bit가 1인 경우
- Z: 연산결과가 0인 경우
- C: 연산결과 carry가 발생하는 경우
- V: 연산결과 overflow가 발생한 경우

Carry는 n-bit data에서 MSB에서 1을 넘기는 경우가 생기는 것이다. Overflow는 두 값의 연산 결과, sign bit를 제외한 MSB에서 1을 넘겨 carry out이 발생. 캐리가 발생하면서 sign bit까지 침범하면서 제대로 된 연산 결과가 나오지 않는 경우로 볼 수 있다. 이 overflow는 더해지는 두 값의 부호가 같은 경우에 발생할 수 있다. 예시로, 4'b0101(5₁₀)로 두 양의 정수를 더할 것이다. 연산을 수행했을 때 4'b1010(-6₁₀)의 값을 갖는다. 또 다른 예시로 두 음의 정수를 더할 것이다. 4b'1011(-5₁₀)으로 덧셈연산을 할 경우 5b'(1)0100(+4₁₀)의 값을 갖는다. 이번 케이스는 overflow와 동시에 carry도 발생한 것을 확인할 수 있다. Sign bit까지 포함해서 4bit였던 수가 덧셈을 연산해 보니 bit가 늘었던 것이다. 정리하자면 overflow는 양수 + 양수, 음수 + 음수의 연산에서 sign비트가 침범되면서 발생하는 것이다. Unsigned였다면 부호가 상관없기에 sign비트가 침범된다는 말이 성립이 안 된다. 그렇기 때문에 carry와 overflow의 조건이 MSB에서 1을 넘어서는 걸로 올림수가 나오는 경우로 동일하다.

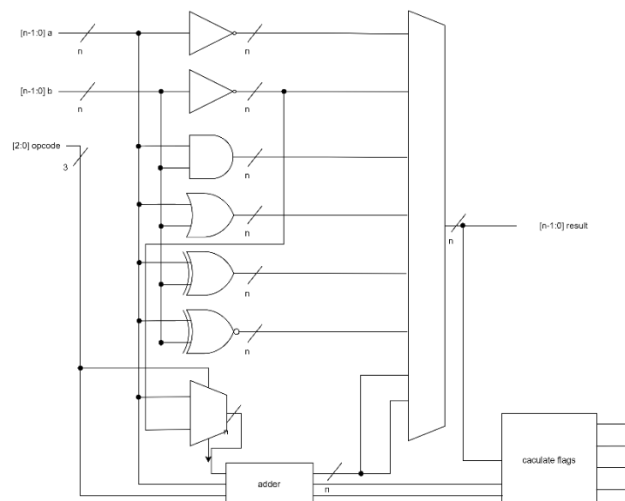
blocking과 non-blocking의 차이는 다음과 같다. 일단 blocking은 '=' 이 기호를 사용한다. LHS는 곧바로 RHS의 값으로 업데이트된다. 즉, '='을 기준으로 왼편의 값은 바로 오른편의 값을 가지게 된다. Scope가 끝나면 곧바로 종료된다. Non-blocking은 '<=' 이 기호를 사용한다. 값을 저장할 필요가 있기 때문에 LHS는 reg type이어야 한다. '<=' 이 기호

LHS는 scope가 끝나기 전까지 해당 값을 유지한다. 하지만 scope가 끝날 때 해당 scope 위로 가서 ' \leq ' 기호를 사용한 구문에서 LHS는 이제서야 RHS의 값으로 업데이트된다. Blocking은 한 번, non-blocking은 두 번 살펴본다고 정리하면 쉬울 것이다. 같이 첨부된 코드에서 nonblocking module을 살펴보면 일단, a, b, c 세 수 모두 0의 값을 가지고 있다고 가정한다. a가 0의 값을 가지고 있다가 1로 바뀌었다. B의 값은 a의 값으로 업데이트된다. 하지만 c는 b가 업데이트 되기 이전의 값인 0 받게 된다. 그래서 a가 1로 바뀐 사이클에서는 a, b, c 이 세 수는 1, 1, 0의 값을 갖게 된다. 다음 사이클이 되어서야 a, b, c 는 1, 1, 1의 값으로 갖게 된다. blocking module을 살펴보면 일단 앞과 동일한 조건에서 시작한다는 가정하에, 이 module은 a가 1이 되는 시점에 a, b, c 세 수 모두가 1, 1, 1로 업데이트 된다. 이전과 다르게 c는 바뀐 b의 값으로 업데이트 되기 때문이다.

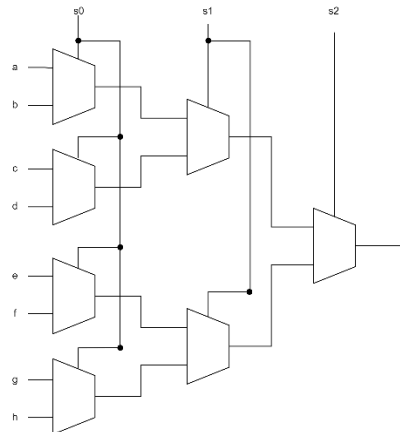
3. 설계 세부사항

입력은 3bit인 opcode와 alu32의 경우 32bit수, alu4의 경우에는 4bit짜리 수를 사용한다. Opcode는 alu에서 연산을 어떤 것을 사용할지 정한다. 000의 경우 첫번째 input의 수를 반전한다. 1의 보수로 만든다. 001의 경우 두번째 input의 수를 1의 보수로 취한다. 010의 경우 두 수를 and연산을 통해 output을 내보낸다. 011의 경우 두 수를 or연산을 통해 output을 내보낸다. 100의 경우에는 두 input 수를 exclusive or 연산을 통해 output을 도출한다. 101의 경우에는 두 input 수를 exclusive nor연산을 통해 output을 내보낸다. 110의 경우에는 두 수를 덧셈 연산한 값을 output에 저장한다. 111의 경우에는 첫 번째 input에서 두 번째 input 수를 감산한 값을 output에 저장한다.

먼저 8개의 연산을 모두 시행한다. 이를 wire bus에 저장을 하고 opcode대로 연산을 수행하기위해 mux를 통해 8개의 값 중 결과값으로 쓸 것을 result 값에 저장한다. result값을 이용하여 n, z, c, v의 flag값을 판별한다.



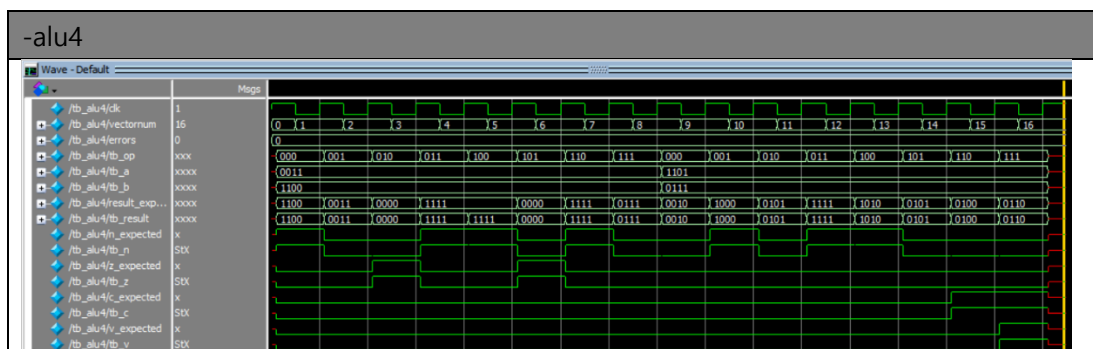
구현을 하면 이런식으로 만들어진다. N-bit Inverter gate 2개와 n-bit and gate 1개, n-bit OR gate 1개, n-bit xor gate 1개, n-bit xnor gate 1개, n-bit 2-to-1 mux 1개 n-bit 8-to-1 mux 1개, adder를 구현하는 것과 flag를 계산하는 module로 이루어져 있다. Adder는 cla를 이용해서 구현하고 flag를 연산할 때에는 삼항연산자를 이용하여 구현한다. 8-to-1 mux는 2-to-1 mux를 7개를 이용해서 구현한다. 7개는 다음과 같이 연결한다.



S0, s1, s2는 각 opcode에 의해 선택되며 결과값이 y에 값으로 결정된다.

4. 설계 검증 및 실험 결과

A. 시뮬레이션 결과



두 수 $a = 4'b0011$ (d3), $b = 4'b1100$ (d-4)

| Input opcode | Operation | Expected value | Measured value |
|--------------|-----------|---|---|
| 000 | Not a | result = 4'b1100 {n, z, c, v} = {1, 0, 0, 0} | result = 4'b1100 {n, z, c, v} = {1, 0, 0, 0} |
| 001 | Not b | result = 4'b0011 {n, z, c, v} = {0, 0, 0, 0} | result = 4'b0011 {n, z, c, v} = {0, 0, 0, 0} |
| 010 | And | result = 4'b0000 {n, z, c, v} = {0, 1, 0, 0} | result = 4'b0000 {n, z, c, v} = {0, 1, 0, 0} |

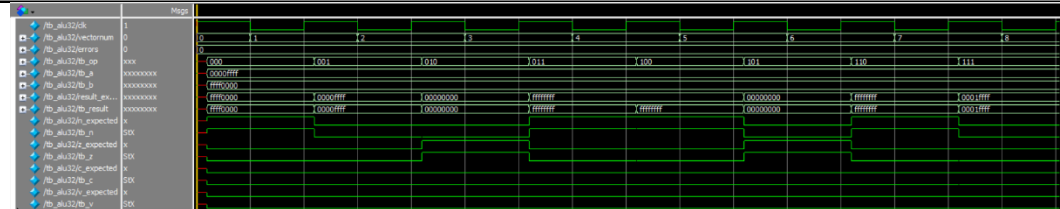
| | | | |
|-----|---------------|---|---|
| 011 | Or | result = 4'b1111 {n, z, c, v} = {1, 0, 0, 0} | result = 4'b1111 {n, z, c, v} = {1, 0, 0, 0} |
| 100 | Exclusive Or | result = 4'b1111 {n, z, c, v} = {1, 0, 0, 0} | result = 4'b1111 {n, z, c, v} = {1, 0, 0, 0} |
| 101 | Exclusive Nor | result = 4'b0000 {n, z, c, v} = {0, 1, 0, 0} | result = 4'b0000 {n, z, c, v} = {0, 1, 0, 0} |
| 110 | Addition | result = 4'b1111 {n, z, c, v} = {1, 0, 0, 0} | result = 4'b1111 {n, z, c, v} = {1, 0, 0, 0} |
| 111 | Subtraction | result = 4'b0111 {n, z, c, v} = {0, 0, 0, 0} | result = 4'b0111 {n, z, c, v} = {0, 0, 0, 0} |

두 수 a = 4'b1101('d-3) b = 4'b0111('d7)

| Input opcode | Operation | Expected value | Measured value |
|--------------|---------------|---|---|
| 000 | Not a | result = 4'b0010 {n, z, c, v} = {0, 0, 0, 0} | result = 4'b0010 {n, z, c, v} = {0, 0, 0, 0} |
| 001 | Not b | result = 4'b1000 {n, z, c, v} = {1, 0, 0, 0} | result = 4'b1000 {n, z, c, v} = {1, 0, 0, 0} |
| 010 | And | result = 4'b0101 {n, z, c, v} = {0, 0, 0, 0} | result = 4'b0101 {n, z, c, v} = {0, 0, 0, 0} |
| 011 | Or | result = 4'b1111 {n, z, c, v} = {1, 0, 0, 0} | result = 4'b1111 {n, z, c, v} = {1, 0, 0, 0} |
| 100 | Exclusive Or | result = 4'b1010 {n, z, c, v} = {1, 0, 0, 0} | result = 4'b1010 {n, z, c, v} = {1, 0, 0, 0} |
| 101 | Exclusive Nor | result = 4'b0101 {n, z, c, v} = {0, 0, 0, 0} | result = 4'b0101 {n, z, c, v} = {0, 0, 0, 0} |
| 110 | Addition | result = 4'b0100 {n, z, c, v} = {0, 0, 1, 0} | result = 4'b0100 {n, z, c, v} = {0, 0, 1, 0} |

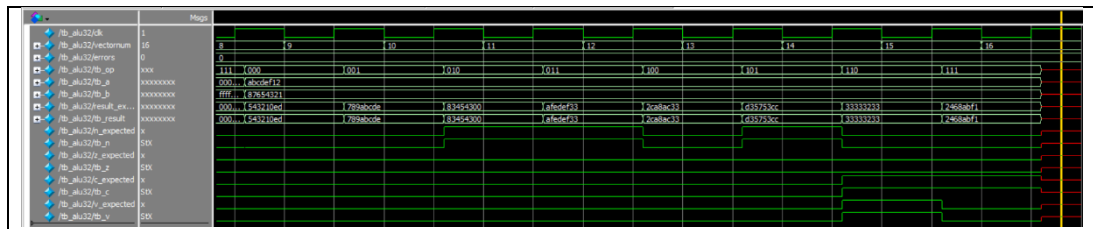
| | | | |
|-----|-------------|---|---|
| 111 | Subtraction | result = 4'b0110 {n, z, c, v} = {0, 0, 1, 1} | result = 4'b0110 {n, z, c, v} = {0, 0, 1, 1} |
|-----|-------------|---|---|

-alu32



두 수 a = 32'h0000_ffff b = 32'hffff_0000

| Input opcode | Operation | Expected value | Measured value |
|--------------|---------------|---|---|
| 000 | Not a | result = 'hffff_0000 {n, z, c, v} = {1, 0, 0, 0} | result = 'hffff_0000 {n, z, c, v} = {1, 0, 0, 0} |
| 001 | Not b | result = 'hffff_0000 {n, z, c, v} = {0, 0, 0, 0} | result = 'hffff_0000 {n, z, c, v} = {0, 0, 0, 0} |
| 010 | And | result = 'hffff_0000 {n, z, c, v} = {0, 1, 0, 0} | result = 'hffff_0000 {n, z, c, v} = {0, 1, 0, 0} |
| 011 | Or | result = 'hffff_0000 {n, z, c, v} = {1, 0, 0, 0} | result = 'hffff_0000 {n, z, c, v} = {1, 0, 0, 0} |
| 100 | Exclusive Or | result = 'hffff_0000 {n, z, c, v} = {1, 0, 0, 0} | result = 'hffff_0000 {n, z, c, v} = {1, 0, 0, 0} |
| 101 | Exclusive Nor | result = 'hffff_0000 {n, z, c, v} = {0, 1, 0, 0} | result = 'hffff_0000 {n, z, c, v} = {0, 1, 0, 0} |
| 110 | Addition | result = 'hffff_0000 {n, z, c, v} = {1, 0, 0, 0} | result = 'hffff_0000 {n, z, c, v} = {1, 0, 0, 0} |
| 111 | Subtraction | result = 'hffff_0000 {n, z, c, v} = {0, 0, 0, 0} | result = 'hffff_0000 {n, z, c, v} = {0, 0, 0, 0} |

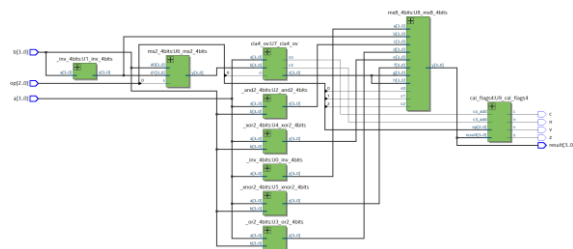


두 수 a = 32'habcd_ef12 b = 32'h8765_4321

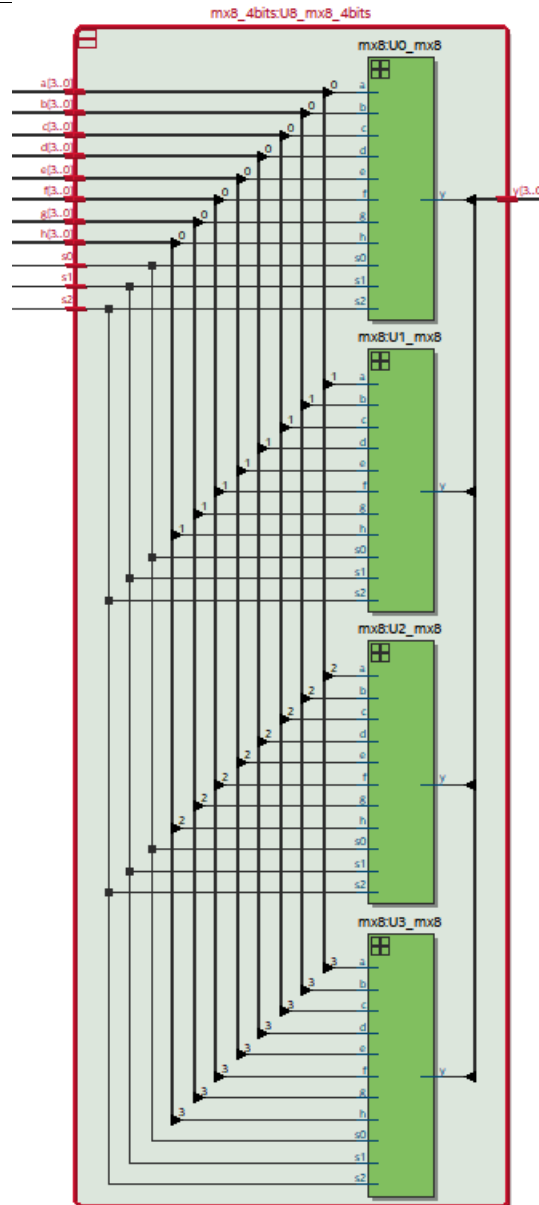
| Input opcode | Operation | Expected value | Measured value |
|--------------|---------------|---|---|
| 000 | Not a | result = 'h5432_10ed {n, z, c, v} = {0, 0, 0, 0} | result = 'h5432_10ed {n, z, c, v} = {0, 0, 0, 0} |
| 001 | Not b | result = 'h789a_bcde {n, z, c, v} = {0, 0, 0, 0} | result = 'h789a_bcde {n, z, c, v} = {0, 0, 0, 0} |
| 010 | And | result = 'h8345_4300 {n, z, c, v} = {1, 0, 0, 0} | result = 'h8345_4300 {n, z, c, v} = {1, 0, 0, 0} |
| 011 | Or | result = 'hafed_ef33 {n, z, c, v} = {1, 0, 0, 0} | result = 'hafed_ef33 {n, z, c, v} = {1, 0, 0, 0} |
| 100 | Exclusive Or | result = 'h2ca8_ac33 {n, z, c, v} = {0, 0, 0, 0} | result = 'h2ca8_ac33 {n, z, c, v} = {0, 0, 0, 0} |
| 101 | Exclusive Nor | result = 'hd357_53cc {n, z, c, v} = {1, 0, 0, 0} | result = 'hd357_53cc {n, z, c, v} = {1, 0, 0, 0} |
| 110 | Addition | result = 'h3333_3233 {n, z, c, v} = {0, 0, 1, 1} | result = 'h3333_3233 {n, z, c, v} = {0, 0, 1, 1} |
| 111 | Subtraction | result = 'h2468_abf1 {n, z, c, v} = {0, 0, 1, 0} | result = 'h2468_abf1 {n, z, c, v} = {0, 0, 1, 0} |

B. 합성(synthesis) 결과

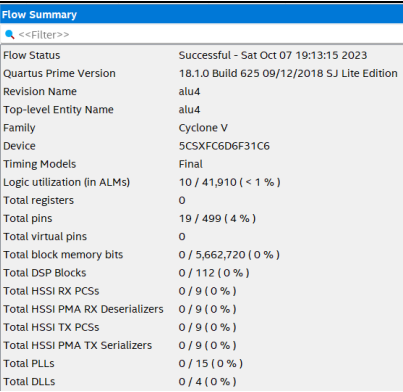
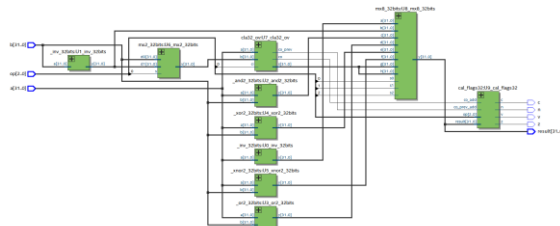
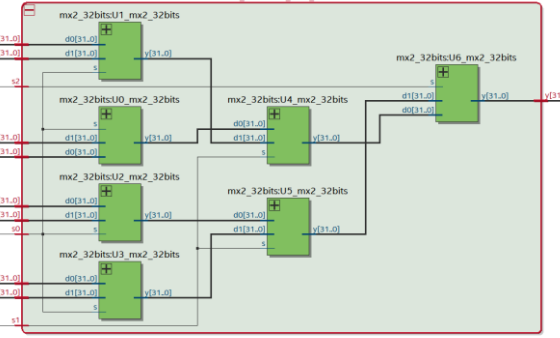
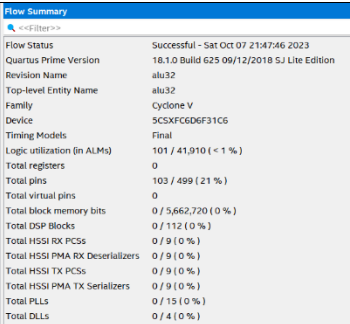
-alu4



설계에서 보인 것과 마찬가지로 크게 10개의 모듈로 이루어져 있다.



4bit 8-to-1 mux가 잘 연결됨.

| | |
|---|---|
|  <p>Flow Summary</p> <p>Flow Status: Successful - Sat Oct 07 19:13:15 2023</p> <p>Quartus Prime Version: 18.1.0 Build 625 09/12/2018 SJ Lite Edition</p> <p>Revision Name: alu4</p> <p>Top-level Entity Name: alu4</p> <p>Family: Cyclone V</p> <p>Device: 5CSXFC6D6F31C6</p> <p>Timing Models: Final</p> <p>Logic utilization (in ALMs): 10 / 41,910 (< 1 %)</p> <p>Total registers: 0</p> <p>Total pins: 19 / 499 (4 %)</p> <p>Total virtual pins: 0</p> <p>Total block memory bits: 0 / 5,662,720 (0 %)</p> <p>Total DSP Blocks: 0 / 112 (0 %)</p> <p>Total HSSI RX PCSs: 0 / 9 (0 %)</p> <p>Total HSSI PMA RX Deserializers: 0 / 9 (0 %)</p> <p>Total HSSI TX PCSs: 0 / 9 (0 %)</p> <p>Total HSSI PMA TX Serializers: 0 / 9 (0 %)</p> <p>Total PLLs: 0 / 15 (0 %)</p> <p>Total DLLs: 0 / 4 (0 %)</p> | <p>Top module의 이름은 alu4로 Top module 설정은 잘된 것으로 보인다.</p> <p>Flow status에서 successful로 나온 것으로 해당 설계의 syntax error는 없는 것으로 보인다.</p> |
| -alu32 | |
|  | <p>앞서 보던 것과 마찬가지로 설계 에서 보인 것과 마찬가지로 크게 10개의 모듈로 이루어져 있다.</p> |
|  | <p>32bit 8-to-1 mux가 잘 연결됨.</p> |
|  <p>Flow Summary</p> <p>Flow Status: Successful - Sat Oct 07 21:47:46 2023</p> <p>Quartus Prime Version: 18.1.0 Build 625 09/12/2018 SJ Lite Edition</p> <p>Revision Name: alu32</p> <p>Top-level Entity Name: alu32</p> <p>Family: Cyclone V</p> <p>Device: 5CSXFC6D6F31C6</p> <p>Timing Models: Final</p> <p>Logic utilization (in ALMs): 101 / 41,910 (< 1 %)</p> <p>Total registers: 0</p> <p>Total pins: 103 / 499 (21 %)</p> <p>Total virtual pins: 0</p> <p>Total block memory bits: 0 / 5,662,720 (0 %)</p> <p>Total DSP Blocks: 0 / 112 (0 %)</p> <p>Total HSSI RX PCSs: 0 / 9 (0 %)</p> <p>Total HSSI PMA RX Deserializers: 0 / 9 (0 %)</p> <p>Total HSSI TX PCSs: 0 / 9 (0 %)</p> <p>Total HSSI PMA TX Serializers: 0 / 9 (0 %)</p> <p>Total PLLs: 0 / 15 (0 %)</p> <p>Total DLLs: 0 / 4 (0 %)</p> | <p>Top module의 이름은 alu4로 Top module 설정은 잘된 것으로 보인다.</p> <p>Flow status에서 successful로 나온 것으로 해당 설계의 syntax error 는 없는 것으로 보인다.</p> |

5. 고찰 및 결론

A. 고찰

추가해야 되는 파일에 fa_v2가 있었다. 해당 파일은 cla4를 구현하면서 만들게 된 파일인데 fulladder이며 sum값을 나오게 하는 module이다. 해당 module를 구현할 때 halfadder인 ha module를 이용하여 구현하는데, 이 때문에 quatars에서 확인하는 Logic utilization의 값의 차이가 있는 것으로 보인다. Sum 값을 구할 때 halfadder를 거치지 않

고 만들어 놓았던 2input xor gate 2개를 이용하여 구하려 했으나 Logic utilization의 수는 달라지지 않았다. 예시로 보여준 것과 다른 수가 나와서 똑같이 맞추려고 했으나, 바뀌지는 않았다. 이 수의 차이는 아마 구현 방식의 차이일 뿐이라고 추측된다.

Testvector로는 testbench를 돌리는 것은 처음 사용해서 벡터 내용을 추출해야 되는데, 해당 파일을 어떻게 추가해야 되는지 방법을 찾는데 오래 걸렸다. 그 방법은 testbench를 돌리기 전 v 파일을 추가할 때 같이 input을 넣을 파일을 추가하면 해결되는 것이었다. 연산이 되고 예상 값을 계산할 때 32bit의 값은 너무 커서 계산하기 너무 복잡해서 최대한 단순한 수로 연산을 하기 위해 input값을 선정하게 되었다.

B. 결론

설계를 하면서 논리연산 opcode가 000, ..., 101 까지는 flag값이 바뀌면 안 될 것 같다는 생각이 들었다. 애초에 논리연산일 경우에는 해당 flag값을 무시하면 되는 것도 맞지만, 이에 대한 예외처리 같이 생각해야 될지를 고민하게 되었다. Overflow의 조건을 co와 co_prev의 xor연산으로 설정했다. N-bit의 수가 있었을 때 $x_{n-1}y_{n-1}s_{n-1}' + x_{n-2}'y_{n-2}'s_{n-2} = co_{n-1}'co_{n-2} \wedge co_{n-1}co_{n-2}'$ 가 overflow의 조건으로 co_prev의 값이 필요했다. 디지털논리회로2 시간에서 배웠던 이론을 다시 한번 보게 되었던 실습이라고 생각한다.

6. 참고문헌

이준환 교수님/디지털논리회로2/광운대학교(컴퓨터정보공학부)/2023

이준환 교수님/컴퓨터공학기초실험2/광운대학교(컴퓨터정보공학부)/2023