컴퓨터 공학 기초 실험2 보고서

실험제목: Subtractor & Arithmetic Logic Unit (ALU)

실험일자: 2023년 10월 04일 (수)

제출일자: 2023년 10월 04일 (수)

학 과: 컴퓨터정보공학부

담당교수: 이준환 교수님

실습분반: 수요일 0, 1, 2

학 번: 2022202075

성 명: 우나륜

1. 제목 및 목적

A. 제목

Subtractor & Arithmetic Logic Unit (ALU)

B. 목적

2's compliment를 사용하여 subtraction을 수행할 수 있으며 하나의 adder에 addition과 subtraction을 구현할 수 있다. Arithmetic Logic Unit에 필요한 flag를 operator를 사용하여 설계하고 Verilog로 subtractor와 ALU를 구현할 수 있다.

2. 원리(배경지식)

A. 2's compliment

2's complement(2의 보수)란 어떠한 2진수의 모든 0과 1을 반전시키고 1을 더해 음수를 얻는 방법이다. 또한, 어떠한 수와 어떠한 수의 2의 보수를 더하면 결과값은 0이 되어야 한다. 이때, 어떠한 수의 2의 보수의 MSB는 1이 되고, 이는 음수를 나타낸다. 예를 들어, 4-bit 00112 (+3)의 보수를 구하자면, 먼저 0과 1의 값을 모두 반전시킨다.

B. Subtraction

Verilog로 subtractor(뺄셈기)를 구현하기 위해 2의 보수를 사용한다. 예를 들어 두 수 A와 B를 빼는 식은 A-B로 나타낼 수 있다. 이때, Subtractor는 A + (-B)로 식을 바꾸어 계산을 수행한다. 따라서 A에서 B를 빼는 것보다 A에서 B의 보수를 더하는 방식으로 뺄셈 기능을 수행한다. 결과적으로 subtractor는 1개의 adder와 1개의 inverter를 사용하여 덧셈과 뺄셈을 연산을 모두 수행하게 된다.

C. ALU

Arithmetic Logic Unit (ALU)란 산술 논리 장치로, 두 수의 덧셈, 뺄셈 등 산술연산과 AND, OR, XOR 등 논리 연산을 계산하는 디지털 회로이다. 본 과제에서는 아래의 표1 3bit opcode에 따라 연산을 수행한다.

Opcode	Operation
3'b000	Not A
3'b001	Not B
3'b010	AND
3'b011	OR
3'b100	Exclusive OR
3'b101	Exclusive NOR

3'b110	Addition
3'b111	Subtraction

표1. 3-bit opcode

D. Flag

ALU내에서 특정 조건과 상황이 만족되는지 확인하기 위해 flag를 사용한다. 본 실습에서는 4가지의 flag를 사용하는데, N (negative), Z (zero), C (carry), V (overflow)가 있다. Negative는 연산 결과의 sign bit인 MSB가 1인 경우를 말한다. 만약 연산 결과값의 MSB가 1이면 음수이므로 N이 1로 세팅된다. Zero는 0을 나타내고 연산결과가 0일 때 1로 세팅된다. Carry는 연산 결과에서 carry가 발생하는 경우를 말한다. 만약 carry가 발생하였다면 carry에 해당하는 비트가 1로 세팅된다. Overflow는 연산 결과의 범위가 표현 범위를 넘어 표현할 수 없는 경우를 말하며, 이때 1로 세팅된다.

Carry는 최상위 비트인 MSB에서 그 위의 비트로 자리 올림이 발생하는 것을 의미하는데, 이때 캐리 그 자체로는 오류 발생과 관련이 없다. 하지만, overflow는 연산 결과 값이 주어진 비트의 자리수로 표현할 수 있는 범위를 넘은 것을 의미하며 수의 범위를 넘어 오류로 간주된다.

4비트 2진수를 예로 들어 설명하면 다음과 같다. 우선, 4비트로 나타낼 수 있는 수의 범위는 -8부터 7까지이다. 1 + (-1) = 0을 2진수로 계산하면, 0001 + 1111 = 10000으로 MBS에서 그 위의 비트로 자리 올림이 발생한다. 이것을 carry가 발생하였다고 하고, 이때의 carry는 무시되고 하위 4비트는 올바른 답인 0을 나타낸다. 다음으로 1 + 7 = 8을 2진수로 계산하면, 0001 + 0111 = 1000이 된다. 캐리는 발생하지 않았지만 결과값이 10진수로 -8이 되고 원하는 결과값인 +8이 계산되지 않았다. 또한, +8은 4비트 2진수로 나타낼 수 없는 값이므로 이 경우는 오버플로우가 발생한다.

E. Modification of CLA

Overflow를 찾기 위해 4-bit CLA에서 carry out과 carry out이전의 c3을 출력하도록 수정하였다.

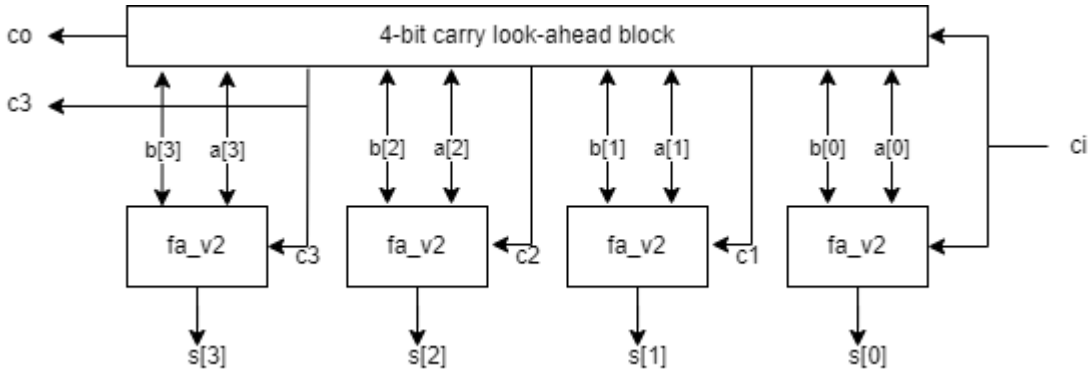

그림1. Modification of CLA

32-bit CLA 또한, 8번째 cla4를 위에서 작성한 4-bit CLA to detect overflow를 사용하여 Carry의 최상위 두 개의 bit를 출력하였다.

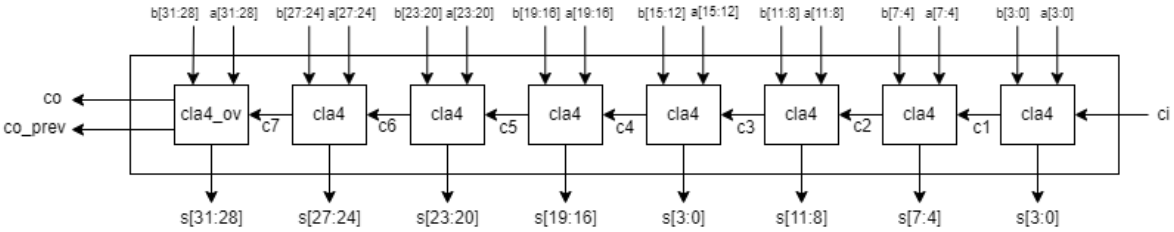

그림2. 32-bit CLA

F. Verilog에서 blocking과 non-blocking

Verilog에서 always 구문과 함께 blocking assignment와 non-blocking assignment를 사용한다. Blocking assignment는 '='를 사용하며 맨 윗줄의 assignment가 수행되고 그 다음의 assignment가 수행되는 순차적인 흐름을 가진다. 주로 combinational 회로에서 주로 활용된다. Non-blocking assignment는 '<='를 사용하며 대입문들은 동시에 수행되고 주로 sequential 회로에서 활용된다.

G. Self-checking testbench with testvectors

본 과제에서는 디지털 논리 시간에 배운 self-checking testbench with testvectors 기법을 사용하여 검증을 수행하였다. 기존의 testbech와 다르게, .tv 파일 내에 대입하고자 하는 값을 작성해두고 testbench module에서 값 생산 및 비교를 수행할 수 있다.

3. 설계 세부사항

A. 4-bit ALU

4-bit Arithmetic Logic Unit를 구현하기 위해 4 bits 계산이 가능한 inverter, AND gate, OR gate, XOR gate, XNOR gate와 이를 통한 기본 4-bit 연산이 가능한 기능, CNCV를 검사할 수 있는 기능, 2 MUX, 8 MUX, 4-bit 산술 논리가 가능한 기능을 구현하였다. 4-bit 연산이 가능한 연산기는 이전 과제에서 설계한 4-bit Carry Look-ahead Adder에서 N, Z, C, V를 검사할 수 있게 수정하였다. 수정한 CLA는 위의 그림 1과 같다. 또한, opcode에 따라 기능을 바꾸는 4-bit 8-to-1 Multiplexer를 구현하기 위해 가장 간단힌 1-bit 2-to-1 Multiplexer부터, 4-bit 2-to-1 Multiplexer, 1-bit 8-to-1 Multiplexer를 생성하였다.

Flag들의 값을 계산하는 cal_flags4 module은 Carry를 계산하기 위해 input으로 co_add를 입력받고 opcode의 op[2:1]의 값이 11과 같으면 co_add의 값을, 그렇지 않으면 0의 값을 가진다. Negative는 결과값의 MSB가 1이면 1의 값을, 그렇지 않으면 0의 값을 가진다. Zero는 결과값이 모두 0이면 1, 그렇지 않으면 0이 된다. Overflow는 opcocde의 op[2:1]이 11이 아니면, 0이 되고 그 반대의 경우 co_add ^ c3_add의 값을 가진다.

4-bit ALU는 위 표1에서 볼 수 있듯이, 총 8개의 기능을 가진다. 이때, 입력값 opcode의 값에 따라 8가지 중 1가지의 기능만 수행할 수 있어야 하는데, 이를 위해 4-bit 8-to-1 MUX를 구현하였다.

B. 32-bit ALU

32-bit Arithmetic Logic Unit은 32 bits 계산이 가능한 inverter, AND gate, OR gate, XOR gate, XNOR gate와 이를 통한 기본 32-bit 연산이 가능한 기능, CNCV를 검사할 수 있는 기능, 2 MUX, 8 MUX, 32-bit 산술 논리가 가능한 기능이 필요하다. 앞서 구현한 4-bit ALU와 유사하게 구현하였다.

4. 설계 검증 및 실험 결과

A. 시뮬레이션 결과

1) 4-bit ALU

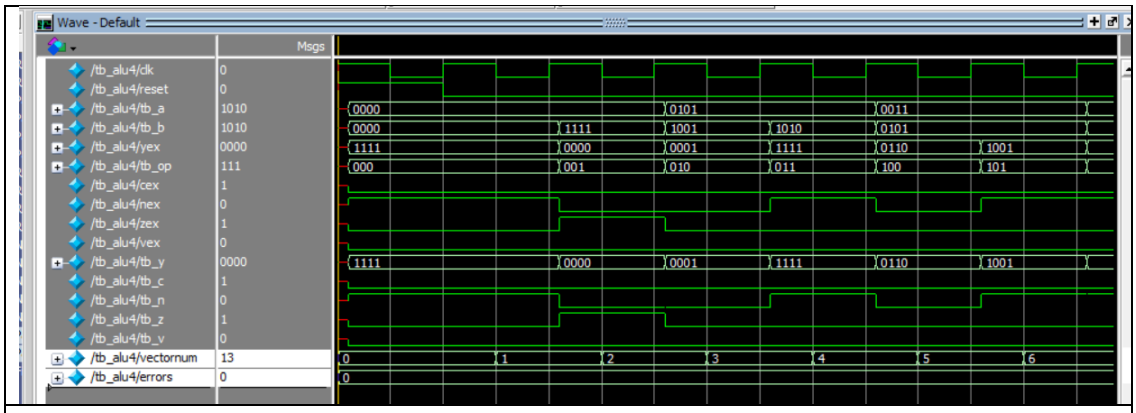

그림3. tb_alu4 Logical operation (binary display)

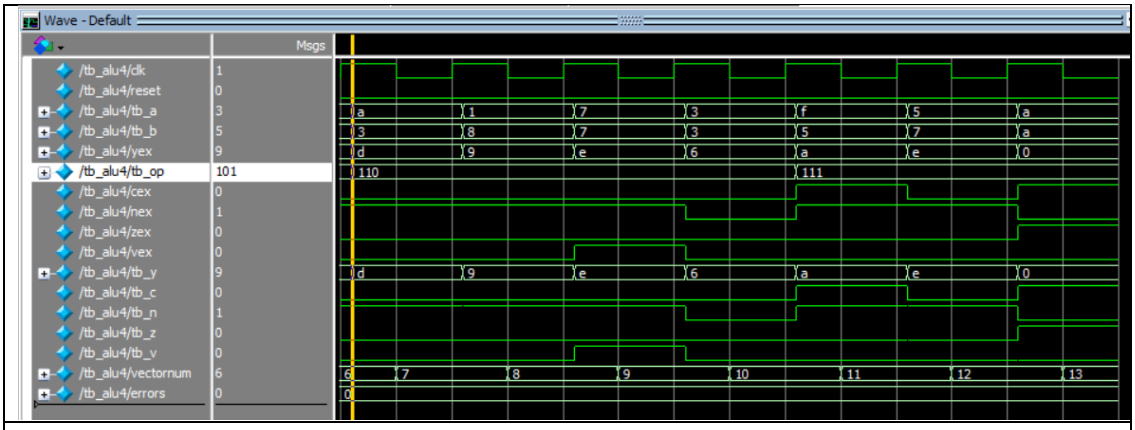

그림4. tb_alu4 Arithmetic operation (hexadecimal display)

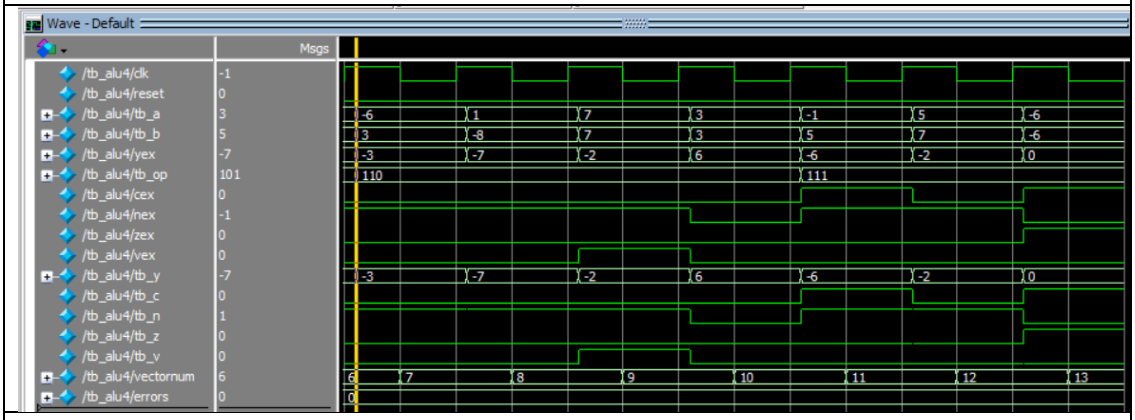

그림5. tb_alu4 Arithmetic operation (decimal display)

위의 그림3 ~ 그림5는 ALU4의 testbench 결과화면이다. Testbench는 self-checking testbench with testvectors 기법을 적용하여 검증하였으며, yex, cex, nex, zex, vex는 예상값을 의미하고, tb_y, tb_c, tb_n, tb_z, tb_v는 계산된 값을 의미한다. 그림3은 opcode 000 ~ 101까지의 연산결과를 binary를 통해 나타낸 것이다. 그림4와 그림5는 opcode 110, 111의 연산결과를 각각 hexadecimal, decimal로 나타낸 것이다. 위 값에서 맨 아래의 errors는 예상값과 계산된 값이 다른 경우의 횟수를 나타내는데, 0인 것을 보아 오류가 나타나지 않은 것을 알 수 있다. 또한, clock의 rising edge에 input이 입력되고, clk=0일 때 예상값과 결과값의 비교가 이루어진다.

2) 32-bit ALU

아래의 그림 6은 ALU32의 testbench waveform이다. ALU32의 testbench도 self-checking testbench with testvectors 기법을 적용하여 검증하였고, 변수가 나타내는 값도 ALU4와 동일하다. Clock의 rising edge에서 input의 값이 대입되고 계산결과 값과 예상값은 clk=0일 때 비교된다. Testbench의 결과값이 올바르게 계산되었고 error의 값이 0인 것으로 보아 알맞게 계산이 되었는 것을 확인할 수 있다.

그림6. tb_alu32 waveform

B. 합성(synthesis) 결과

1) 4 bits ALU

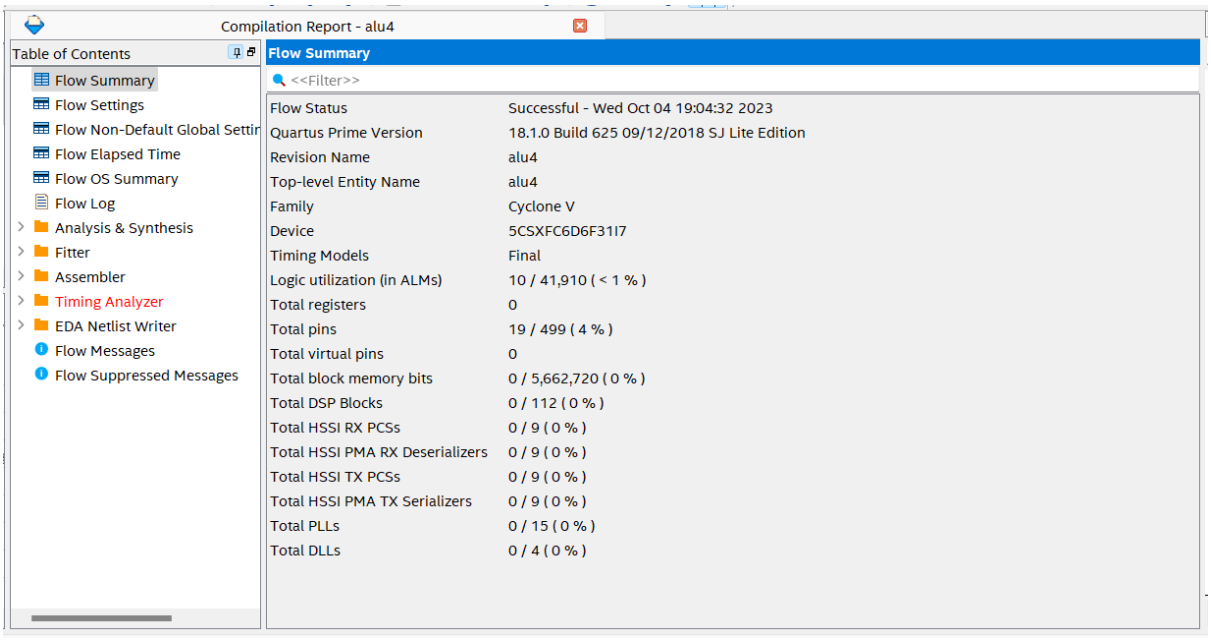

그림7. 4-bit ALU flow summary

위의 그림은 ALU4의 flow summary이다. 사용된 logic utilization은 10으로 1%미만이고, total register는 0, total pins는 19로 4% 사용되었다.

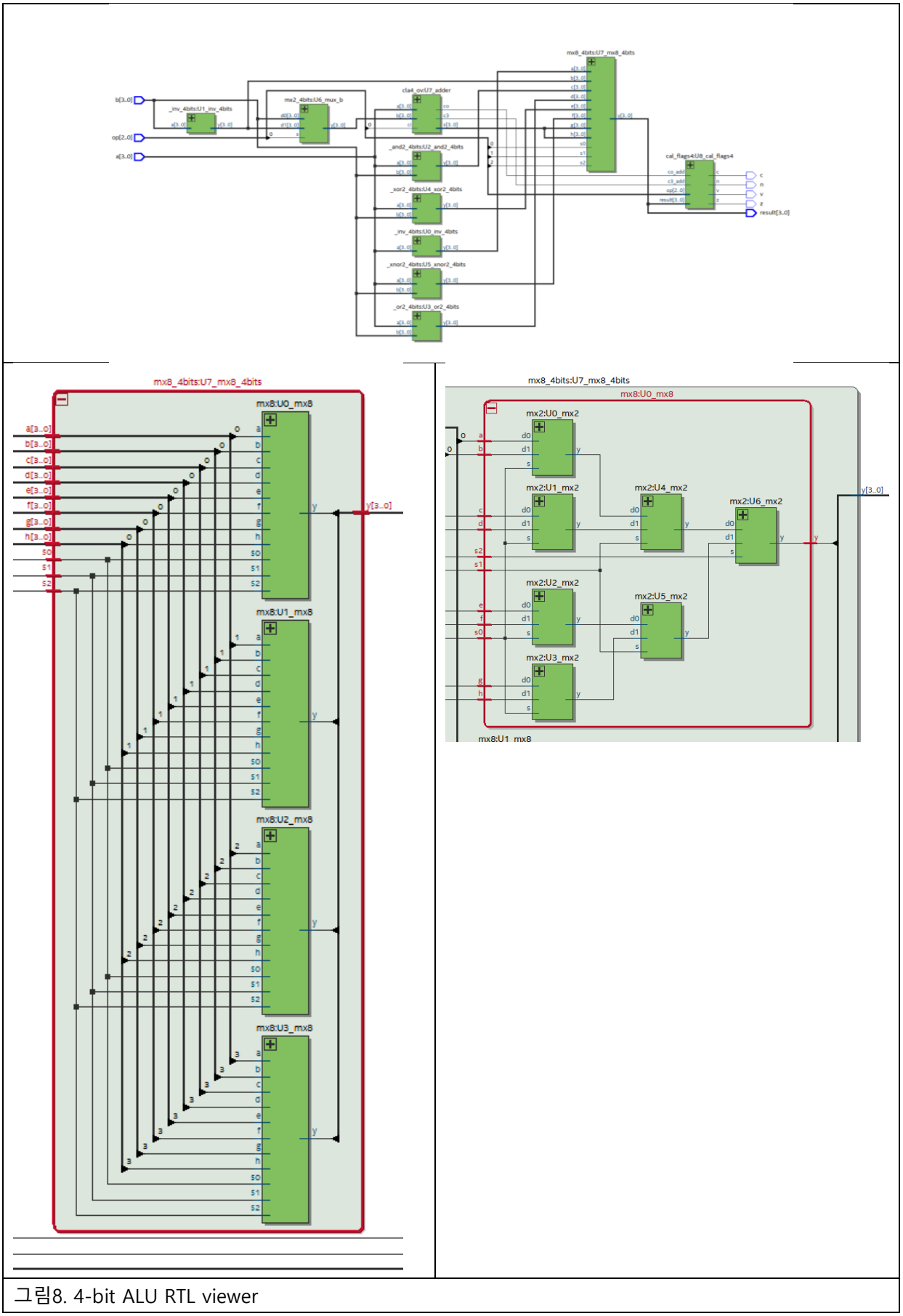

그림8. 4-bit ALU RTL viewer

그림8은 ALU4의 RTL viewer이다. 4-bit inverter for input b, 4 bits 2-to-1 MUX, cla4, 4 bits AND, 4 bits OR, 4 bits XOR, 4 bits inverter, 4 bits XNOR, cal_flags module이 사용되었다. mx8_4bits module에는 4개의 1 bit 8-to-1 MUX가 사용되었고, 하나의 1 bit 8-to-1 MUX module에는 7개의 1 bit 2-to-1 MUX가 사용되었다.

2) 32 bits ALU

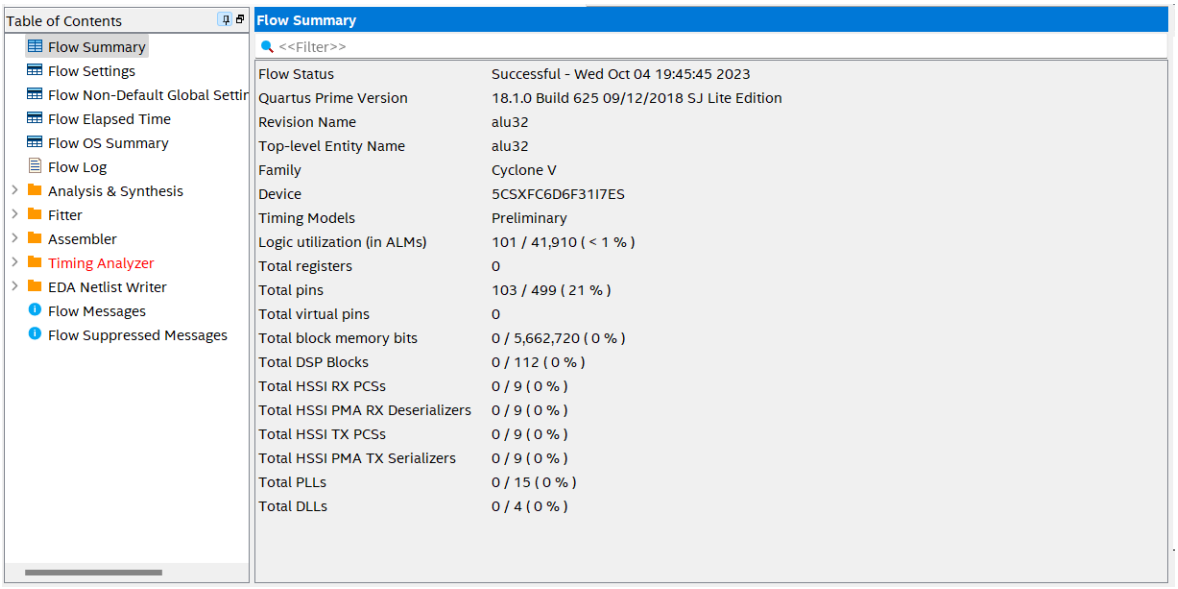

그림9. 32-bit ALU flow summary

그림9는 ALU32의 flow summary이다. 사용된 logic utilization은 101로 1%미만이고, total register는 0, total pins는 103으로 21%가 사용되었다.

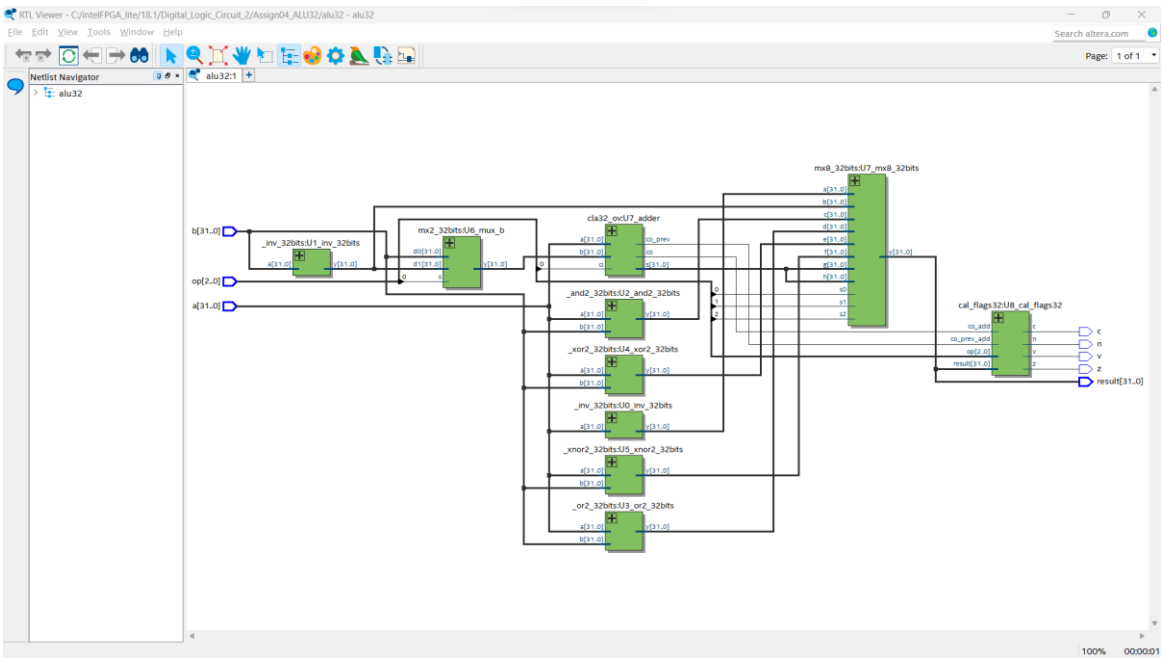

그림10. 32-bit ALU RTL viewer

그림10은 ALU32의 RTL viewer이다. 32 bits inverter를 사용하여 input b의 inverse를 계산하고, 32 bits 2-to-1 MUX를 통해 덧셈을 위한 b 또는 뺄셈을 위한 -b가 cla32로 입력될지 선택된다. ALU32에 필요한 add와 sub 기능을 cla32를 통해 계산하고, 32 bits AND, 32 bits XOR, 32 bits inverter, 32 bits XNOR, 32 bits OR의 기능이 구현되어있다. 그리고 입력된 opcode에 따라 결과값을 출력하기 위해 32-bit 8-to-1 MUX가 사용되었다.

5. 고찰 및 결론

A. 고찰

1. Testvector를 저장한 문서를 불러올 수 없는 에러가 발생하였다. 이는 .tv파일을 simulation > modelsim 파일 내에 저장하니 해결되었다.
2. 저장된 testvector의 값들이 제대로 옮겨지지 않았다. testvector문서를 hexadecimal로 작성하고, readmemh를 통해 불러왔는데, 값이 제대로 불러와지지 않고 이상한 값들이 추가되거나 아예 다른 값이 되어버렸다. 이후에 base를 hexadecimal에서 binary로 바꾸니 해결되었다.

B. 결론

해당 실험을 통하여 새롭게 알게 된 사항이나 느낀 점 그리고 실험결과에 대한 응용을 작성한다. (1문단 이상)

이번 과제에서 처음으로 testvector를 사용하여 testbench를 구현하는데 많은 어려움을 겪었다. Modelsim 폴더에 파일을 저장해야 하는 오류는 오류 메시지를 통해 바로 해결하였지만, 이외의 오류를 처리하는데 많은 시간이 소요되었다. 특히, hexadecimal을 불러오는 reamemh의 기능이 제대로 수행되지 않아 이 부분에서 가장 힘들었다. 이후에 readmemb를 사용하여 오류를 해결하였지만 여전히 왜 그러한 오류가 발생하였는지 알 수 없어 아쉬웠다.

Verilog로 간단한 3-bit opcode를 사용하는 ALU4, ALU32를 구현하였다. 이를 통해 이후에 opcode의 bit수를 늘려 더 많은 기능을 수행하고 더 큰 비트의 연산을 할 수 있는 ALU를 구현할 수 있을 것 같다.

6. 참고문헌

[1] 캐리와 오버플로우 / https://blog.naver.com/hjyang0/183698525

[2] testbench / 이준환교수님, 광운대학교 / 2023년