컴퓨터 공학 기초 실험2 보고서

실험제목: Subtractor & Arithmetic Logic Unit

실험일자: 2023년 09월 25일 (월)

제출일자: 2023년 10월 10일 (화)

학 과: 컴퓨터정보공학부

담당교수: 이준환 교수님

실습분반: 월요일 0, 1, 2

학 번: 2020202031

성 명: 김재현

1. 제목 및 목적

A. 제목

Subtractor & Arithmetic Logic Unit (ALU)

B. 목적

이번 실습에서는 operator인 3-bit opcode에 따라 두 숫자의 산술연산(덧셈, 뺄셈 등등)과 논리 연산(AND, OR, XOR, 등등)을 계산하는 디지털 회로인 Arithmetic logic unit(ALU, 산술 논리 장치)를 구현한다.

2. 원리(배경지식)

1) Substractor

기존의 여러 adder에서 opcode의 LSB를 ci로 받고, opcode가 subtraction을 나타내고 있다면 b에 1의 보수를 취해 a와 더하고 ci와 더해준다. Opcoed가 subtraction을 나타낸다는 것은 LSB가 1이라는 뜻이고 이는 ci가 1이라는 뜻이기에 b에 1의 보수를 취한 후 1을 더해주는 것과 같아진다. 따라서 b에 2의 보수를 취하는 것과 같은 효과를 가지는데, a와 b의 2의 보수를 더하는 것은, 곧 a와 b의 뺄셈과 같으므로 하나의 adder와 inverter만 있다면 덧셈과 뺄셈을 모두 수행할 수 있다.

2) Arithmetic Logic Unit (ALU)

A, B, opcode를 입력 받아 Not, And, Or, Xor, Xnor, Add, Sub 등 8개의 연산을 진행하고, 8 to 1 MUX를 통해 Opcode에 따라 단 1개의 연산결과만을 출력한다. 그리고 연산결과, co, c3를 통해 flags를 계산해 출력한다.

3) 4-bit CLA to detect overflow

Overflow는 음수와 음수의 덧셈결과 양수가 나올 때, 양수와 양수의 덧셈결과 음수가 나올 때 발생한다. 그러려면, co = 1, c3 = 0 이거나 co = 0, c3 = 1 이어야 한다.

Co	C3	V
0	0	0
0	1	1
1	0	1
1	1	0

$$V = co'c3 + coc3' = co \oplus c3$$

따라서 overflow flag를 계산하기 위해서는 co, c3에 대한 정보를 알아야 하는데 기존의 CLA는 co만을 출력하므로 약간의 수정을 거쳐 c3도 함께 출력하도록 코드를 수정해준다.

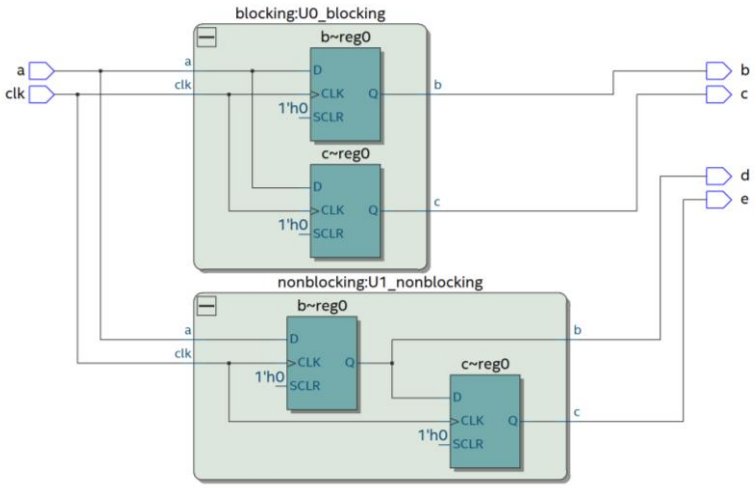
4) 32-bit CLA

4-bit CLA를 8개 이어 붙여 구성한 기존의 32-bit CLA 역시 co만을 출력한다. 따라서 마지막 4-bit는 기존의 4-bit CLA에서 약간의 수정을 거친 4-bit CLA to detect overflow를 사용하여 co, c31을 출력하도록 해준다. 그렇게 된다면 co, c31을 통해 overflow flags를 계산할 수 있게된다.

5) 4개의 flags 중 carry와 overflow의 차이점

Carry는 연산 결과 carry가 발생하는지를 판별하는 flag이고, overflow는 연산 결과 표현 가능한 수의 범위를 초과했는지를 판별하는 flag이다. 따라서 carry와 overflow는 8개의 opcode 중 addition, substraction 연산에서만 발생함을 알 수 있다. 두 개의 피연산자의 연산 결과, MSB의 연산으로 carry out이 발생한다면 carry가 1이 된다. 두 피연산자가 모두 음수라면 덧셈 연산 결과 양수가 나오거나, 모두 양수일 때 덧셈 연산 결과 음수가 나온다면, 표현가능한 범위를 초과했다는 의미이므로 overflow가 1이 되는 것이다.

6) Blocking과 non-blocking의 차이점



Blocking은 절차지향언어처럼 위에서부터 라인별로 차례로 대입이 실행되고, non-blocking은 모든 대입이 동시에 일어난다. 따라서 blocking의 경우 b=a, c=b가 차례로 실행되어 b, c에 모두 a가 대입이 되지만, non-blocking의 경우 b=a와 c=b가 동시에 실행되어 b에는 a값이, c에는 원래 b 값이 대입된다.

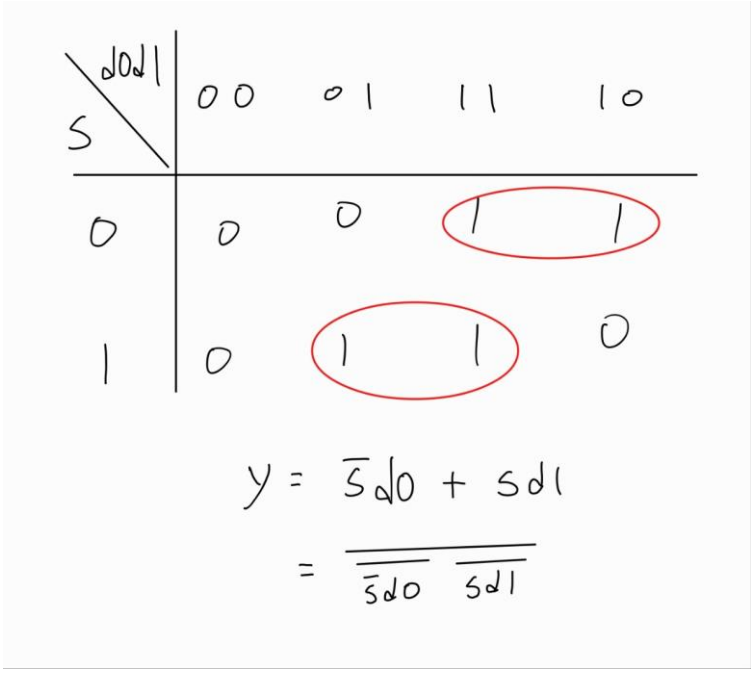
3. 설계 세부사항

1) cla4_ov

기존의 cla4에서 output port로 c3를 추가해준 module이다. 기존의 cla4는 clb4에서 계산돼 출력되는 c3를 오로지 MSB bit 연산에만 사용했지만 cla4에서는 그와 동시에 output으로 출력하여 flags를 계산하는데 c3를 사용할 수 있도록 한다.

2) mx2

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



진리표와 카르노 맵을 활용하여 2to1 mux의 논리식을 도출하고 이와 동일하게 베릴로그 코드를 작성했습니다.

3) mx2_4bits

4bits 입력을 2개 받아 그 중 1개만을 출력하는 4비트짜리 mx2를 구현하기 위해 d0

와 d1의 각 비트를 1bit mx2 instance 4개에 입력했습니다. 따라서 s의 값에 따라 각 1bit mx2에서 출력하는 값들이 곧 d0, d1 둘 중 하나의 값이 된다.

4) mx8

8개의 입력을 2개씩 짝지어서 mx2에 입력해서 출력된 결과값 4개를 다시 한번 mx2에 입력하고, 마지막으로 나온 결과값 2개를 mx2에 한번 더 입력하여 나온 값을 출력합니다. 총 세 번의 단계를 거치기 때문에 mx8에 입력되는 s값은 3bits이어야 합니다.

5) mx8_4bits

4bits 입력을 2개 받아 그 중 1개만을 출력하는 4비트짜리 mx2를 구현하기 위해 d0와 d1의 각 비트를 1bit mx4 instance 4개에 입력했습니다. 따라서 s의 값에 따라 각 1bit mx4에서 출력하는 값들이 곧 d0, d1 둘 중 하나의 값이 된다.

6) cal_flags4

flags 계산은 삼항 연산자를 활용했다. carry는 오직 덧셈, 뺄셈 연산에서만 발생하므로 opcode 상위 두 비트가 11이 아니면 0을, 11이면 덧셈, 뺄셈 연산 결과 나온 carryout을 반환합니다. MSB가 부호를 나타내므로 negative에는 최상위 비트를 반환한다. 연산결과가 0이라면 zero에 1을 반환한다. overflow 역시 오직 덧셈, 뺄셈에서만 발생한다. 따라서 opcode 상위 두 비트가 11이 아니면 0을 반환하고, 원리(배경지식)에서 설명했듯 overflow는 $V = co'c3 + coc3' = co \oplus c3$ 라는 논리식을 가지기 때문에 opcode 상위 두 비트가 11이면 $co \oplus c3$을 반환한다.

4. 설계 검증 및 실험 결과

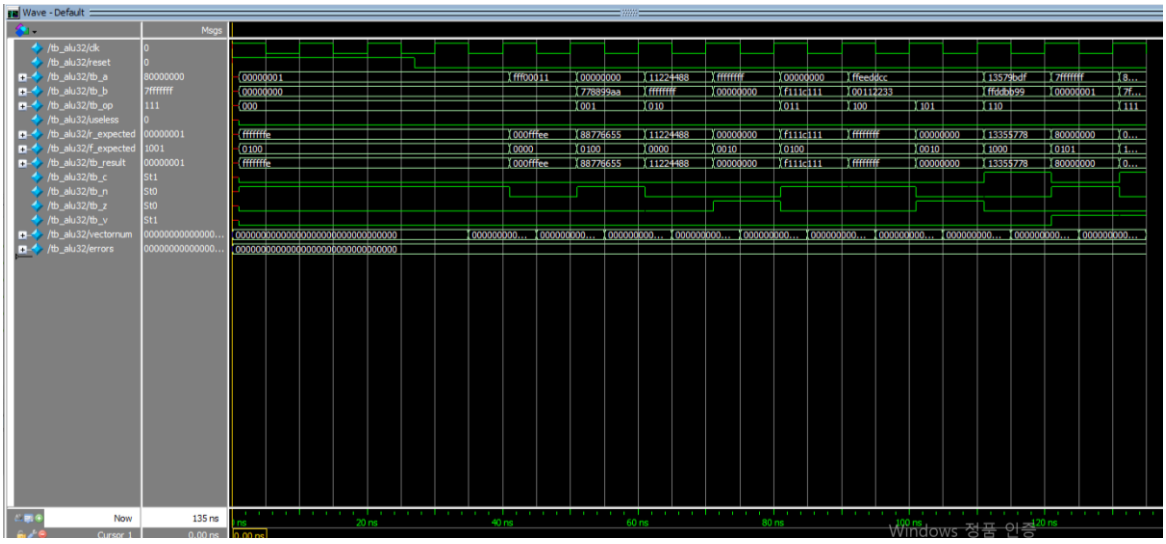
A. 시뮬레이션 결과

1) alu4

```
#
# add wave *
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
#         17 tests completed with          0 errors
# ** Note: $finish    : C:/Users/kk200/verilog/week4. ALU/alu4/tb_alu4.v(49)
#    Time: 195 ns  Iteration: 1  Instance: /tb_alu4
# 1
# Break in Module tb_alu4 at C:/Users/kk200/verilog/week4. ALU/alu4/tb_alu4.v line 49
```

example.tv 파일에 a_b_opcode_expected result_expected flags 형식으로 입력했다. tb_a, tb_b, op, r_expected, f_expected 에서 example.tv에 저장된 data들을 읽어들인다. tb_a, tb_b, op를 alu4의 instance의 input으로 입력하고, tb_result, tb_c, tb_n, tb_z, tb_v를 output으로 지정하여 tb_result, {tb_c, tb_n, tb_z, tb_v}를 r_expected, f_expected와 비교한다. 연산 결과 값과 예상 값이 다를 경우 오류 메시지를 출력하고 몇 개의 에러가 있었는지 출력하도록 testbench를 설계했는데, 오류 메시지가 출력되지 않고, 0개의 에러가 존재한다는 메시지가 출력된 것으로 보아, alu4 module이 잘 설계되었음을 알 수 있다.

2) alu32

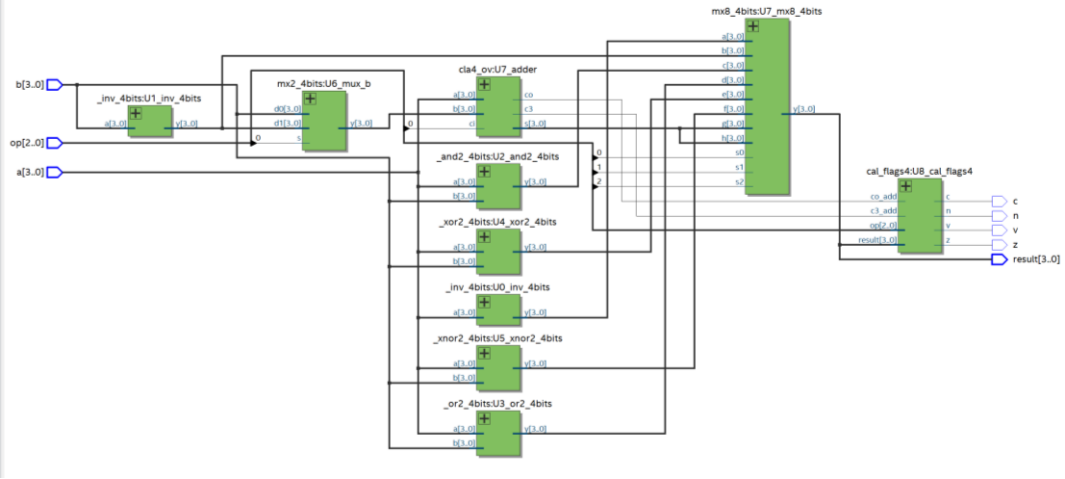



```
# add wave *
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
#         11 tests completed with          0 errors
# ** Note: $finish    : C:/Users/kk200/verilog/week4. ALU/alu32/tb_alu32.v(50)
#    Time: 135 ns  Iteration: 1  Instance: /tb_alu32
# 1
# Break in Module tb_alu32 at C:/Users/kk200/verilog/week4. ALU/alu32/tb_alu32.v line 50
```

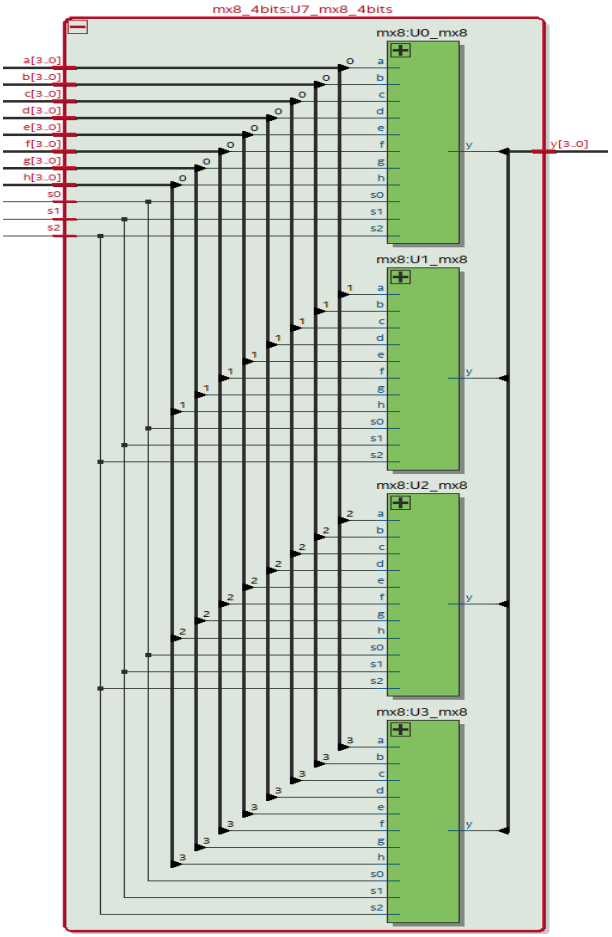
tb_alu32에서는 $readmemh를 사용하여 example.tv 파일을 16진수로 읽어들였기 때문에, opcode를 16진수로 해석할 수 있도록 reg useless를 선언하여 testvectors에서 reg로 값들을 대입시킬 때 opcode 앞에 useless를 붙여줬습니다. 그 외에는 tb_alu4와 동일하게 테스트를 진행했고, 테스트 결과 오류 메시지가 출력되지 않고, 0개의 에러가 존재한다는 메시지가 출력된 것으로 보아, alu4 module이 잘 설계되었음을 알 수 있습니다.

B. 합성(synthesis) 결과

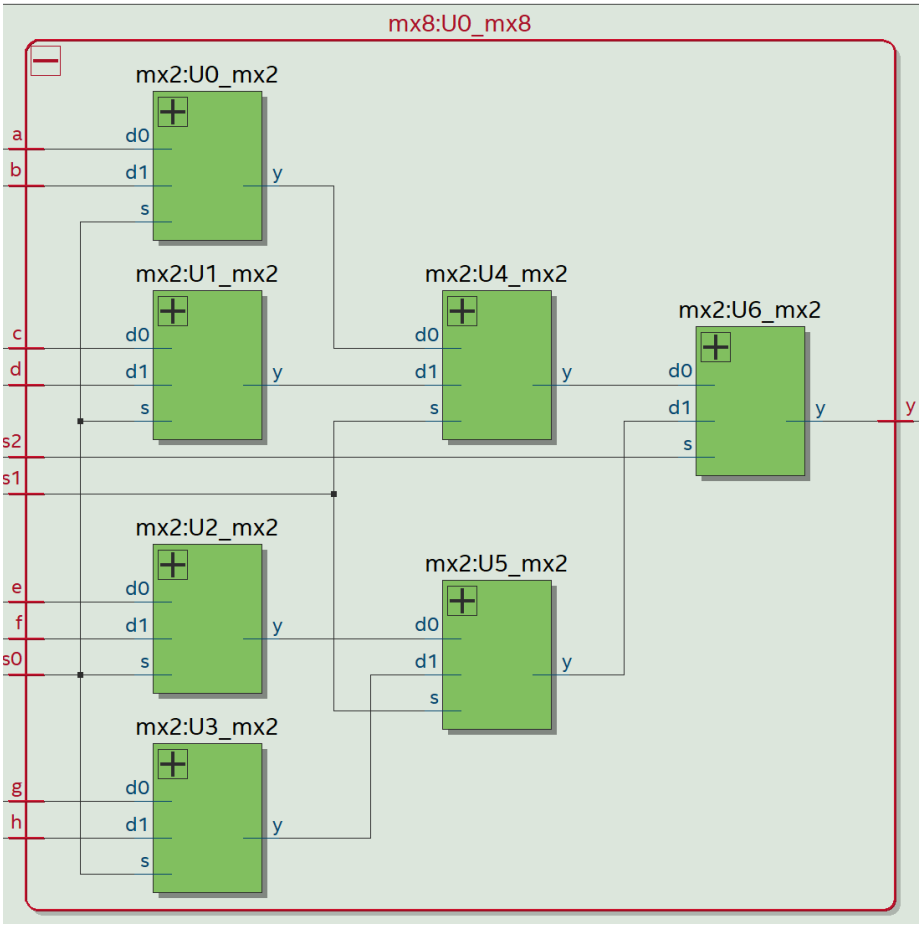
1) alu4



4bits data a와 b가 각 연산기에 입력되어 모든 연산이 동시다발적으로 진행된다. op값에 따라 mx8-4bits에서 진행된 모든 연산 중 한 개의 연산만을 result로 출력한다. 또한 op 값과 상위 2개의 carry에 따라 cal_flags에서 c, n, v, z 모든 flags를 업데이트하여 출력한다.



mx8_4bits는 4bits 데이터를 각 비트들을 mx8로 instance 4개에 넣어서 한 개의 데이터만을 출력하는 것을 확인할 수 있습니다.



mx8은 mx2를 사용해 세 단계에 걸쳐 8개의 데이터 중 한 개의 데이터만을 출력한다.

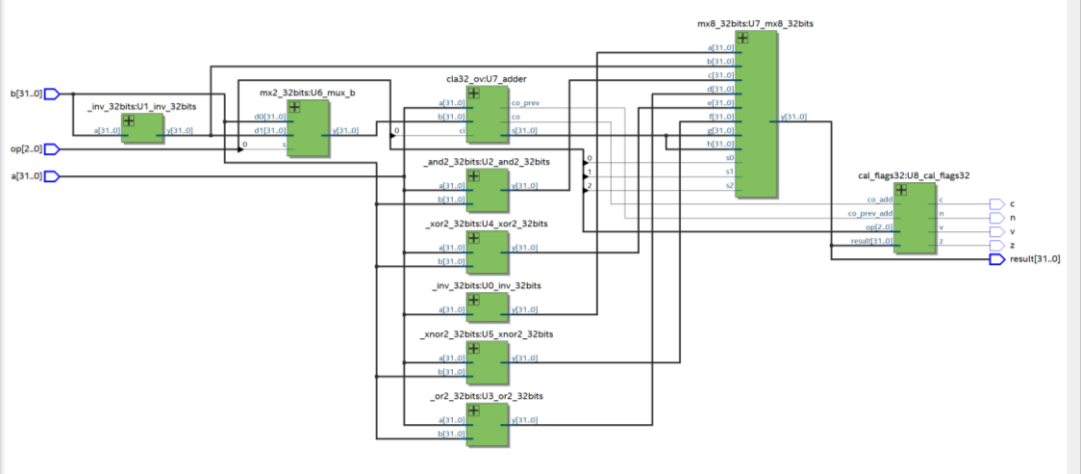
Flow Summary

<<Filter>>

Flow Status	Successful - Mon Oct 09 21:51:49 2023
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)	9 / 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 %)

total pins of alu4 = a 4bits + b 4bits + op 3bits + results 4bits + flags 4bits = 19

2) aul32



32bits data a와 b가 각 연산기에 입력되어 모든 연산이 동시다발적으로 진행된다. op값에 따라 mx8-32bits에서 진행된 모든 연산 중 한 개의 연산만을 result로 출력한다. 또한 op 값과 상위 2개의 carry에 따라 cal_flags에서 c, n, v, z 모든 flags를 업데이트하여 출력한다.

Flow Summary

<<Filter>>

Flow Status	Successful - Tue Oct 10 00:28:36 2023
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)	97 / 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 %)

total pins of alu4 = a 32bits + b 32bits + op 3bits + results 32bits + flags 4bits = 103

alu4에 비해 alu32가 bits가 8배이고 그에 따라 logic utilization이 10배 가까이 월등히 높음을 알 수 있습니다.

5. 고찰 및 결론

A. 고찰

testbench를 설계할 때 tb_alu4에서는 $readmemb를 사용했기 때문에 각 비트를 직접 계산하여 예상 값을 도출하기가 쉬웠는데 tb_alu32에서는 $readmemh를 사용하여 16진수로 값을 읽어들여서 예상 값을 도출하는데 더 많은 노력이 필요했고, opcode가 3비트이기 때문에 16진수로 읽어들이려면 opcode 앞에 1비트가 추가로 필요했기에, useless register를 선언하여 opcode 앞에 붙여줌으로서 "example.tv" 파일에서 data를 읽어들일 때 opcode를 16진수로 잘 읽어들일 수 있었습니다. 또한 testvectors를 선언할 때 useless register를 고려하지 않고 testvectors를 103비트로 선언하였는데 그로인해 input a의 MSB가 0으로 자동처리되어 input 값이 변하는 오류를 범했는데, 이 또한 testvectors를 104bits로 수정함으로써 해결했습니다.

B. 결론

input 값 a, b, op에 따라 모든 연산이 잘 이루어지고 있음을 testvectors를 활용한 testbench로 증명했다. 또한, blocking과 nonblocking에 대해서 알아보며 직접 구동해본 결과 잘 돌아가는 모습을 볼 수 있었다. 또한 이번 실습을 통해 어셈블리프로그래밍 수업시간에 막연히 받아들이며 넘어갔던 부분들을 각각의 flag의 작동방식에 대해서 세세하게 이해할 수 있었다.

tv파일에 input 값들과 예상 값들을 미리 적어두고 이 값들을 clk이 rising할 때 대입하고 falling할 때 실제 값들을 예상 값들과 비교하여 값이 다르다면 출력하도록 testbench를 구성함으로써 올바른 값이 출력됐는지 여부를 테스트하는 법을 체득할 수 있었다.

6. 참고문헌

이준환 교수님/컴퓨터공학기초실험2/광운대학교(컴퓨터정보공학부)/2023

이준환 교수님/디지털논리회로2/광운대학교(컴퓨터정보공학부)/2023